The AMF Architecture in a Multiple User Interface Generation Process

Kinan Samaan, Franck Tarpin-Bernard
Laboratoire ICTT, Lyon
21, Av. Jean Capelle, 69621 Villeurbanne cedex - FRANCE
kinan@ictt.insa-lyon.fr, franck.tarpin-bernard@insa-lyon.fr

ABSTRACT
In the context of Multiple User Interface (MUI) generation, this paper presents the AMF architecture on which a method relies for the adaptation of interactive applications to the specific characteristics of a targeted context. In our model-based approach, we use a library of task patterns and interaction patterns to adapt the interaction model of the application.

For the description of AMF architecture, we use an XML file that ensures the link between the tasks model and the functional core of the application. An engine parses and processes the file to run the application.

Keywords
Multiple User Interface, design patterns, model-based, AMF, XML based language.

INTRODUCTION
Everyday, new platforms are emerging with new characteristics and new interaction capabilities. The traditional classification (PC, mobile phone, PDA, interactive TV…) is not sufficient to generate MUI for interactive applications. We must not propose the same interface for a PDA with a small coloured screen and a keyboard and another one with a larger screen and a stick.

These last years, many researches have been led on Multiple User Interface (MUI) generation processes. The model-based approaches seem to be the most promising. In classical software engineering, models like MVC [4] have been exploited for a long time. Recently, user models or task models have been introduced to help for the generation of MUI. Whereas these models can be easily described with XML, it is not the case with MVC. According to us, it is very important to be able to also model with a portable file the interaction model.

Indeed, the interaction model is one of the most important models to consider because, on the one hand, it manages the interaction between the user and the application, and on the other hand, it ensures the link with the functional core of the application. This model was often neglected in the steps of MUI generation.

In this paper, we present the basis of a new approach that integrates the interaction model and the platform model into the design and generation processes.

The adaptation process forces the designer to clarify/explain the links between the task model, the interface model and interaction model. For the description and the adaptation of the interaction model, our approach relies on the AMF architecture (Agent with Multiple Facets), which has been created in 1997 for modelling common interactive applications [12]. Indeed, AMF presents the following advantages that will be deeply discussed:

- The multi-facets concept is very interesting especially for multiple presentation definitions.
- The XML description of AMF models allows the definition of an abstract interaction model and patterns of interaction.
- A run-time engine is able to execute an AMF model and allows switching dynamically from a specific model or sub-model to another.

RELATED WORKS
To design and implement MUI [9] interactive software, the first approaches were based on description languages like UIML [1] and XIML (RedWhale) [15]. These languages organize adaptation processes in two levels or steps: an abstract level and a concrete level corresponding to the implementation in HTML, Java or WM. If this approach certainly represents a progress, we think that the proposed abstractions are still insufficiently generic. Mainly it imposes a particular style of interaction, i.e. this abstract level specifies, for example, a button that will be concretised under different forms (aspect, position, etc.), but mandatory imposes the use of button while neglecting the other forms of interaction more adapted for a given platform as the vocal recognition or use of physical button. In this way, abstract level is portability oriented and not plasticity [9] oriented.

Newer approaches try to define a component-based framework that will allow runtime migratable user interfaces, which are independent of the target software platform, the target device and the interaction modalities [5]. In these frameworks, the user interfaces are merely considered as a presentation of a single service or of more functionally grouped services. These kinds of solutions are more powerful than the language-based ones but, as

they do not use task models, they are not able to filter the functions that cannot be used in a specific context.

Other approaches mainly focus on the task model [7][11]. They filter a generic task model in order to define an abstract user interface and later build a concrete user interface. The Abstract user interface is described in terms of Abstract Interaction Objects (AIOs) [14] that are latter transformed into Concrete Interaction Objects (CIOs) once a specific target has been selected. Calvary et al. defined a unifying reference framework for multi-target user interfaces [2]. This framework tries to give a global view of the multiples approaches on MUI.

The improvement is important. However, these methods do not explicitly define an interaction model. As a consequence, they are very efficient for modelling basic interactions but are limited for modelling more sophisticated ones like "drag and drop". Currently they are dedicated to graphical interaction and need to be extended to manage multimodal and multi-style interactions.

THE GENERAL APPROACH OF MUI GENERATION

To allow a more important variation at the interaction style level as well as at the implementation level, it is necessary to introduce a richer and generic description and to replace the language-based approach by an architecture-based approach [10]. In the AMF approach, we propose to start with a task model and to map it to an architecture-based abstract interaction model expressed in AMF, then to concretise this one in relation to the characteristics of the working platform. Once the concrete interaction have been chosen, the degrees of freedom available allow an ultimate adaptation to the user and the environment.

Our approach consists in organizing the MUI generation process in 4 phases (Fig 1):

- Abstract application definition phase,
- Interaction styles selection phase,
- Concrete interface generation phase,
- Final adaptation phase.

The first phase consists in modelling the generic task model and the abstract interface model, and defines the links between these two models and the abstract interaction model of the application. The designer builds these models once for all.

The second phase aims at dynamically generating the components of the interface that are adapted to the target. This phase is activated when a target (that is a triple < user, environment, platform > [13]) is running the software. The process consists in transforming the previous models with an adaptation engine. For the adaptation, this engine considers two extra components: the platform model and a library of task and interaction patterns.

We can summarize the work of the adaptation engine in three points:

- It removes non-realizable tasks from the generic task model (e.g. removes a "print" task if the system does not detect a printer connected to the target). At this stage the engine also removes the elements of the abstract interface that are closely related to the removed tasks.
- According to the input devices of the target, the mechanism replaces each abstract task by a concrete task using a "task patterns" library (e.g. moving element with a pointing device).
- In parallel, the engine enriches the XML description of abstract interaction model by inserting the patterns that are associated to the task patterns using an "interaction patterns" library.

The third phase aims at generating a concrete interface where all the resources that will be used are selected but where the final parameters (layout, colour, volume…) are not set.

According to the characteristics of the devices (size, resolution, capacity…) and the user preferences, a second engine selects among potential resources for each element of the semi-concrete interface, the ones that are more appropriate to the circumstances of use. The dependencies that have been defined between the domain objects are considered so that the choices are coherent.

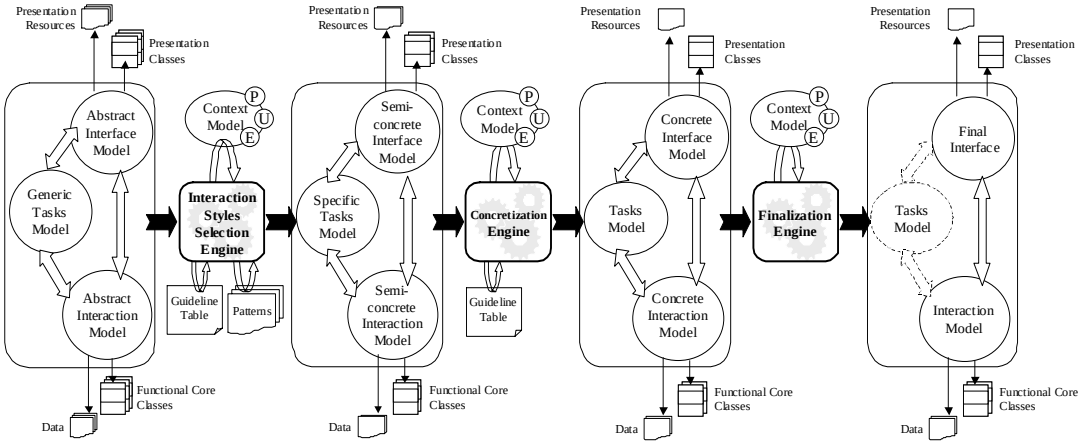

Figure 1. Our vision of the complete process of MUI generation

In the final adaptation, the application and especially its interface takes its final form. A third engine is building the final layout of the interface taking into account the presentation preferences described in the context model.

Let's focus now on the core of our approach, which is the AMF model.

BASIC DESCRIPTION OF AMF

A large number of architectures for Interactive Software have been described, e.g., MVC (Model-View-Controller) [4], PAC (Presentation-Abstraction-Controller) [3], ADC (Abstraction-Display-Controller) [6]. Most of these architectures are based on the traditional view of interactive software, namely the view that an interactive software system can be separated into the application and the user interface. The application part contains the functionality of the software and the user interface part contains the representation of this functionality proposed to the application user(s).

AMF is a multi-agents and multi-facets architecture model that specifies the software architecture of an interactive application. It enables the design of reusable elements. It can be extended and adapted to the need of specific applications. The AMF model can be seen as an upgrade of the PAC and MVC models. It combines the conceptual powerfulness of multi-agents architectures such as PAC while providing an operational implementation schema, which is a key factor of the success of MVC.

Fundamentally, AMF provides four key features:

- It generalizes the concept of facet, extending their number from 3 in both MVC and PAC to n, i.e. an open-ended set of useful facets (e.g. Cooperation in CSCW);
- It formalizes the control components;
- It fits well with task modelling approaches and design patterns;
- It defines an API and relies on a powerful runnable engine.

AMF provides a graphical formalism that represents the structure and specifies the temporal sequence of processes. Finally, a Java implementation of an AMF engine enables the execution of an AMF model coupled to applicative classes.

The class 'agent' is the basic component of AMF models. Each agent is made of facets and control administrators. It can imply other agents. Each class agent can generate several instances. Each facet incorporates logical communication ports and is associated to an applicative class where some functions, called «daemons», are mapped to the ports.

AMF proposes a unified formalism to model control components because such formalisms are rare and usually difficult to use in real contexts (see Petri nets for instance). Yet, these components are the major pieces of architectural models and it is of great importance to provide an efficient modelling tool. The control component of each agent is its main part because it manages all the communications between the facets of the agent and other agents. AMF defines 2 kinds of elements:

- At the Facet level, communication ports present the services that are offered by the facet and the ones that are needed (respectively input and output ports).
- At the Agent level, control administrators are connecting communication ports. These administrators can easily be standardized (OR, AND, Sequence, etc.) and extended to handle complex controls such as multi-user synchronization or interaction tracking.

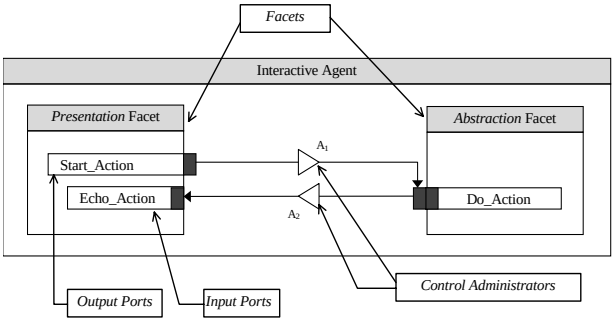

Figure 2. Basic elements of AMF architecture

We briefly introduce 2 special features of the control administrators:

- After being activated, a target port is always returning a message to the source port. This "acknowledgement" message is generally ignored but it can be used to return data to the source port. When it is the case, the control administrator is represented with a black triangle (see figure 3a).
- The possible existence of multiple instances of a unique class drove us to provide a default mechanism that broadcasts messages from a control administrator to the target ports of all the instances of an agent. To be able to activate a specific instance of an agent, we add an optional parameter to the activate function in order to explicitly define a target agent. The identity of the agent is usually known only during runtime. So we do not need a new type of administrator but only a new activation technique. Yet, for a better understanding of the visual model, our advice is to add a little vertical bar at the end of the administrator to explain that a filtering is done on the target agents (see figure 3b).

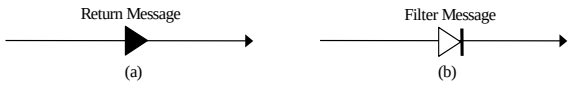

Figure 3. Return and Filter features of control administrators.

Finally, a Java implementation of an AMF engine enables the execution of an AMF model coupled to applicative classes.

The AMF Model can be published using an XML notation. Here is the Document Type Definition we use:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Agent (Agent*, Administrator+, Facet+)>
<!ATTLIST Agent
    Name CDATA #REQUIRED
    Sub-agent CDATA #REQUIRED
    Type CDATA #REQUIRED
>
<!ELEMENT Administrator (Sources+, Targets+)>
<!ATTLIST Administrator
    Name CDATA #REQUIRED
    Type (Simple | Return | Filter | ReturnFilter | Sequence)
#REQUIRED
    TypeAC (Abstract | Concrete) #REQUIRED
>
<!ELEMENT Targets EMPTY>
<!ATTLIST Targets
    Name CDATA #REQUIRED
>
<!ELEMENT Sources EMPTY>
<!ATTLIST Sources
    Name CDATA #REQUIRED
>
<!ELEMENT Facet (Port+)>
<!ATTLIST Facet
    Name CDATA #REQUIRED
    Type CDATA #REQUIRED
>
<!ELEMENT Port EMPTY>
<!ATTLIST Port
    Name CDATA #IMPLIED
    Type CDATA #REQUIRED
    TypeIO (2 | i | o) #REQUIRED
    TypeAC (Abstract | Concrete) #REQUIRED
    DaemonName CDATA #REQUIRED
    FacetName CDATA #REQUIRED
>
```

The AMF Engine

The goals of the AMF are to help design, implementation, use and maintenance. Our approach consists in combining both multiagent view (like PAC) and layered view (like Arch). The multiagent view is used during the design and a layered technology is used for implementation.

Actually, agents are dual entities: one part located into the AMF engine manages the control of the interactions while another one, on the application side, manages both widgets for interactivity and real domain-dependent abstractions.

The 5 levels of Arch model are present:

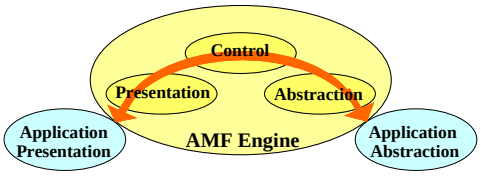

Figure 4. The Layers of the AMF implementation

To implement AMF architecture, we built an engine that manages all the AMF objects (agents, facets, ports and administrators) and their communications. The external elements, which are both objects that define the functional kernel of the application and objects that use a graphical toolkit, are linked to the AMF objects. For instance, each communication port is associated to a function called daemon in the Application side. This daemon is automatically triggered when the port is activated.

At runtime, for each user's action (button pressed, menu selection…), the corresponding event received by an application object (i.e. the one that manages the window) activates an output port of the associated AMF agent in the engine (↘ symbol in the graphical models). At the end of the control processing, input ports are activated and their daemons are run.

AMF concepts can be compared to ones of Java Beans. Indeed, facets are components (Beans) that are able to present themselves (with their ports) and that communicate by sending and receiving messages. Ports and administrators are very similar to listeners and adapters (in fact, the Java implementation of AMF uses them). However, AMF relies on a sophisticated engine so that programmers can use predefined components, such as standard administrators, which are real objects and not only java interfaces.

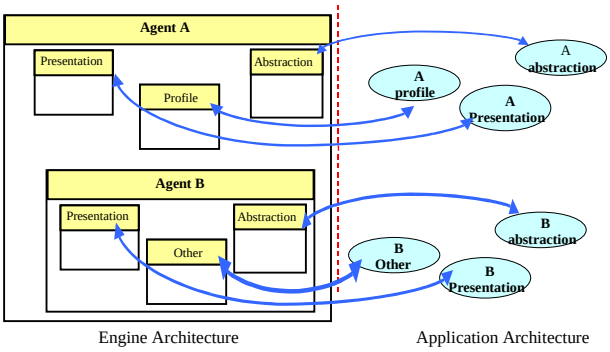

Figure 5. Links between AMF objects inside the engine and application classes outside.

THE AMF DESCRIPTION OF AN ABSTRACT INTERACTION MODEL

To illustrate our approach, we are considering a classical game called «The Towers of Hanoi » (figure 6), which consists in moving rings of different sizes to reach a goal. The rings are stacked up on three stems; they have an initial position and should be moved to reach a target-

position. The shifting must respect the following rules: only one ring can be moved at a time and a ring with a given size cannot be placed upon a ring of a smaller size.

There are three types of object in this application: the game which contains the rule and the other objects, the stem (with three instances) upon which the rings are slipped and the ring (with 3 to 5 instances according to the complexity of the game). The interaction consists in a succession of operations: the selection of the ring (on the top of a stem) followed by the shifting, then the validation of the move (respect rule) and finally, the detection of the end of the game.

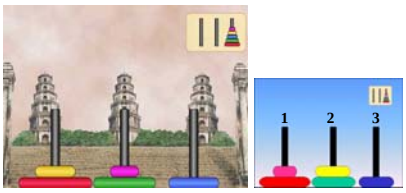

Figure 6. "The Towers of Hanoi" application

The first step for the designer consists in defining a generic task model. The Task Model is a tasks tree that is hierarchically organized. Various formalisms have been proposed to model the task model. We use the CTT notation and CTTE editor [9] for its description and modeling. In our approach, the Generic task model contains regular nodes corresponding to common tasks and abstract nodes that will be "specialized" later in relation with the context of use (e.g. a "Selection&Move" node that will be specialized by a "Drag & Drop" sub-tree). Figure 7 presents the Generic task model of "The Towers of Hanoi" using a CTT notation.

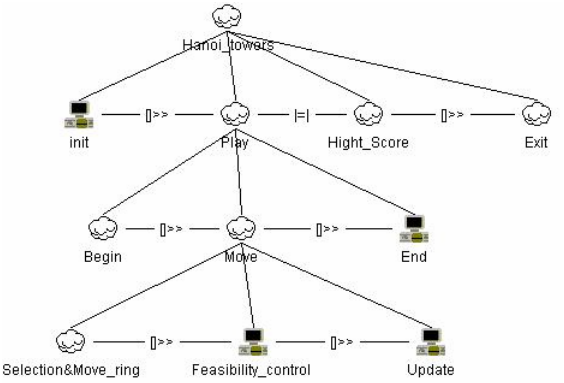

Figure 7. Generic Task Model of "The Towers of Hanoi"

From the generic task model we can establish the abstract interaction model of the application. This model is an AMF description that contains abstract ports. These ports represent functionalities that can be executed differently according to the specifications of the target.

Figure 8 represents the abstract interaction model of "The Towers of Hanoi" game. After a move, if it is a valid one (*Validate_move ports*), the scene must be re-painted (*Refresh ports*). The task of selecting and moving a ring in this model is abstract (elements are represented with dotted lines). Indeed, this action can be carried out differently according to the means of interaction that are available on the given target: a mouse, then the user may drag & drop, or a keyboard, then he/she will type the number of the source stem and after the target stem's one. To skip from an abstract interaction model to a concrete model, we need to replace abstract ports using the patterns.

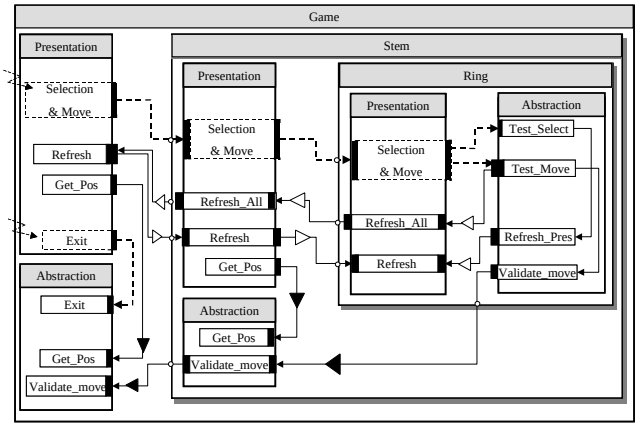

Figure 8. Abstract Interaction Model of "The Towers of Hanoi"

Here is an extract of the XML description of the AMF abstract model for the "Towers of Hanoi" application. We only detail the Ring agent. Note that the "selection_move" port is an abstract port. In addition, the names of the elements are rich ("#" symbols are used by the engine) so that we can use dynamic links.

```
<?xml version="1.0" encoding="UTF_8"?>
<!DOCTYPE Agent SYSTEM "amf.dtd">
<Agent Name="GAME" Sub-agent="1" Type="game">
    <Agent Name="STEM" Sub-agent="1" Type="stem">
        <Agent Name="RING" Sub-agent="0" Type="ring">
            …
            <Administrator
            Name="Test_Select#RING#STEM#GAME"
            Type="Return" TypeAC="Abstract">
                <Sources
        Name="Selection_Move#PRESENT#RING#STEM#GAME"/>
                <Targets
            Name="Test_Select#ABSTR#RING#STEM#GAME"/>
            </Administrator>
            <Facet Name="ABSTR#RING#STEM#GAME"
            Type="abstr#ring#stem#game">
                <Port
            Name="REFRESH#ABSTR#RING#STEM#GAME"
        Type="refresh#abstr#ring#stem#game" TypeIO="2"
            TypeAC="Concrete"
            FacetName="ABSTR#RING#STEM#GAME"
            DaemonName="refresh"/>
                <Port
            Name="TEST_MOVE#ABSTR#RING#STEM#GAME"
            Type="test_move#abstr#ring#stem#game" TypeIO="2"
```

```
            TypeAC="Concrete"
            FacetName="ABSTR#RING#STEM#GAME"
            DaemonName="test_move"/>
                …
                <Port
            Name="Refresh_all#PRESENT#RING#STEM#GAME"
            Type="refresh_all#present#ring#stem#game" TypeIO="2"
            TypeAC="Concrete"
            FacetName="PRESENT#RING#STEM#GAME"
            DaemonName="refresh_all"/>
                <Port
        Name="Selection_Move#PRESENT#RING#STEM#GAME"
            Type="selection_move#present#ring#stem#game"
            TypeIO="i" TypeAC="Abstract"
            FacetName="PRESENT#RING#STEM#GAME"
            DaemonName="null"/>
            </Facet>
        </Agent>
        …
</Agent>
```

THE INTERACTION PATTERNS

The AMF model is a part of the «design patterns» approach because some combinations of agents – facets – ports constitute potential patterns that can be isolated and described.

Thus, we have defined several patterns related to interaction means (mouse, keyboard…) that are used to interact with the application in different contexts. For sure, other patterns may be defined.

As an example we present hereafter an interaction pattern used for a task of selection and removal of an element among a set of elements located into a container (Fig 9). This pattern is applied if the interaction is done with a mouse.

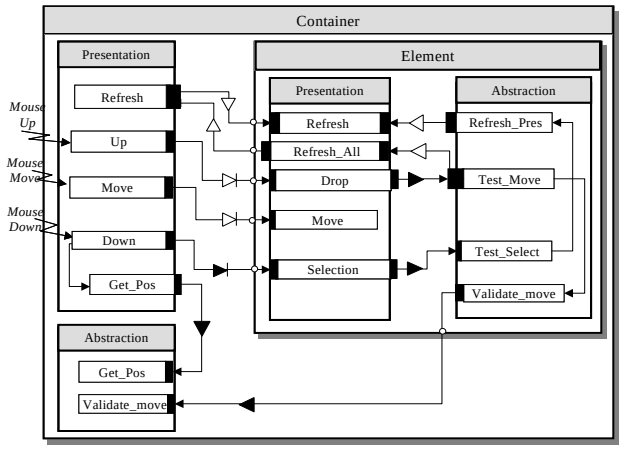

Figure 9. The graphical "Select and Move" Pattern for a mouse.

For this pattern we have the following XML description:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Agent SYSTEM "amf.dtd">
<Agent Name="CONTAINER" Sub-agent="1" Type="container">
    <Agent Name="ELEMENT" Sub-agent="0" Type="element">
        <Administrator Name="Refresh#ELEMENT#CONTAINER"
        Type="Simple" TypeAC="Concrete">
            <Sources
        Name="REFRESH#ABSTR#ELEMENT#CONTAINER"/>
            <Targets
        Name="REFRESH#PRESENT#ELEMENT#CONTAINER"/>
        </Administrator>
        …
        <Facet Name="PRESENT#ELEMENT#CONTAINER"
        Type="present#element#container">
        …
            <Port
            Name="MOVE#PRESENT#ELEMENT#CONTAINER"
            Type="move#present#element#container" TypeIO="i"
            TypeAC="Concrete"
            FacetName="PRESENT#ELEMENT#CONTAINER"
            DaemonName="Move"/>
            <Port
        Name="SELECTION#PRESENT#ELEMENT#CONTAINER"
            Type="selection#present#element#container" TypeIO="2"
            TypeAC="Concrete"
            FacetName="PRESENT#ELEMENT#CONTAINER"
            DaemonName="Selection"/>
        </Facet>
    </Agent>
    <Administrator Name="Refresh#CONTAINER" Type="Simple"
    TypeAC="Concrete">
        …
    <Administrator Name="Move#CONTAINER" Type="Filter"
    TypeAC="Concrete">
        <Sources Name="MOVE#PRESENT#CONTAINER"/>
        <Targets
        Name="MOVE#PRESENT#ELEMENT#CONTAINER"/>
    </Administrator>
    <Administrator Name="Drop#CONTAINER" Type="Filter"
    TypeAC="Concrete">
        <Sources Name="UP#PRESENT#CONTAINER"/>
        <Targets
        Name="DROP#PRESENT#ELEMENT#CONTAINER"/>
    </Administrator>
    <Administrator Name="Selection#CONTAINER"
    Type="ReturnFilter" TypeAC="Concrete">
        <Sources Name="DOWN#PRESENT#CONTAINER"/>
        <Targets
    Name="SELECTION#PRESENT#ELEMENT#CONTAINER"/>
    </Administrator>
    …
    <Facet Name="ABSTR#CONTAINER" Type="abstr#container">
        <Port Name="GET_POS#ABSTR#CONTAINER"
        Type="get_pos#abstr#container" TypeIO="i"
        TypeAC="Concrete"  FacetName="ABSTR#CONTAINER"
        DaemonName="Get_pos"/>
        <Port Name="VALIDATE#ABSTR#CONTAINER"
        Type="validate#abstr#container" TypeIO="2"
        TypeAC="Concrete"  FacetName="ABSTR#CONTAINER"
        DaemonName="Validate"/>
    </Facet>
    …
</Agent>
```

INTERACTION MODELS ADAPTATION

The adaptation engine replaces the abstract tasks with a concrete task and the interaction pattern that is related to the task is inserted into the abstract interaction model of the application. This replacement is done according to the characteristics of the target. Hence, concrete ports and concrete administrators will replace the abstract ports that are inside the abstract interaction model.

A name-based approach is used to replace the generic names (CONTAINER & ELEMENT) by the concrete ones (STEM & RING).

If we consider the Towers of Hanoi example running on a platform with a mouse, the pattern presented in the figure 9 will be instanced. A name-based rule enables to maintain the link between the ports and the interface element that receives the action. Then, the pattern replaces the abstract ports in the interaction model of the application. This process produces a final interaction model of the application (figure 10).

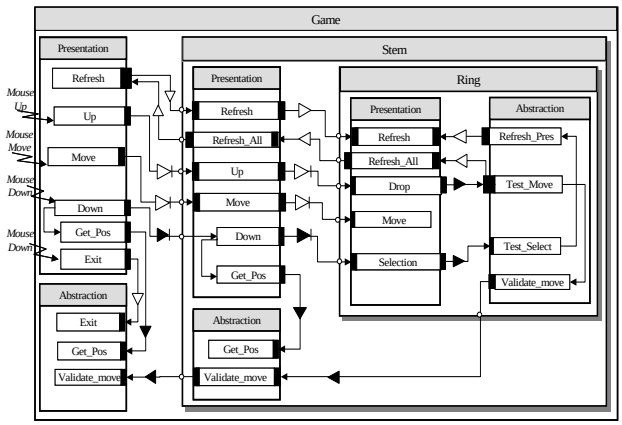

Figure 10. Concrete interaction model of the Towers of Hanoi with a mouse

Here is an extract of the XML description of the AMF concrete model for the application. The DaemonName fields of the concrete ports are method names of the Java classes defined by the developer.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Agent SYSTEM "amf.dtd">
<Agent Name="GAME" Sub-agent="1" Type="game">
    <Agent Name="STEM" Sub-agent="1" Type="stem">
        <Agent Name="RING" Sub-agent="0" Type="ring">
            <Administrator Name="Refresh#RING#STEM#GAME"
            Type="Simple" TypeAC="Concrete">
                <Sources
            Name="Refresh#ABSTR#RING#STEM#GAME"/>
                <Targets
            Name="Refresh#PRESENT#RING#STEM#GAME"/>
            </Administrator>
            <Administrator
            Name="Refresh_all#RING#STEM#GAME"
            Type="Simple" TypeAC="Concrete">
                <Sources
            Name="Test_Move#ABSTR#RING#STEM#GAME"/>
                <Targets
            Name="Refresh_all#PRESENT#RING#STEM#GAME"/>
            </Administrator>
            …

                <Port
            Name="DROP#PRESENT#RING#STEM#GAME"
            Type="drop#present#ring#stem#game" TypeIO="2"
            TypeAC="Concrete"
            FacetName="PRESENT#RING#STEM#GAME"
            DaemonName="Drop"/>
                <Port
            Name="MOVE#PRESENT#RING#STEM#GAME"
            Type="move#present#ring#stem#game" TypeIO="i"
            TypeAC="Concrete"
            FacetName="PRESENT#RING#STEM#GAME"
            DaemonName="Move"/>
                <Port
            Name="SELECTION#PRESENT#RING#STEM#GAME"
            Type="selection#present#ring#stem#game" TypeIO="2"
            TypeAC="Concrete"
            FacetName="PRESENT#RING#STEM#GAME"
            DaemonName="Selection"/>
            </Facet>
        </Agent>
…
    </Agent>
…
</Agent>
```

The DaemonName fields of the concrete ports are method names of the Java classes defined by the developer. Here is the interface of the RingPres class.

```
// File : iRingPres.java
public interface iRingPres
{
        void    Refresh();
        void    Refresh_All();
        private Down(MouseEvent arg);
        private Move(MouseEvent arg);
        private Drop(MouseEvent arg);
}
// iRingPres
```

CONCLUSION

In this paper we have shortly presented an architecture-based approach for the generation of Multiple User Interfaces. It incorporates the use of task patterns and interaction patterns. To describe the interaction model and the interaction patterns we used the AMF architecture which is composed of an XML description of an AMF model and a run-time engine.

We use AMF to define the interaction model, which enables us to obtain an abstract description of the interaction model of the application. Processing (filtering and enriching) this description with the XML parsing mechanisms helps us to concretize the abstract model in a progressive way. At the end of the process, we obtain a concrete description of the AMF interaction model. A Java implementation of an AMF engine enables the execution of an AMF model coupled to applicative classes. We can now imagine building other AMF players (non-java) that will allow the application to the AMF-XML files.

This approach is original in the sense that it tries to unify the task model, the interaction model and the resources of the application, i.e. the functional resources (Java classes of the application domain) and interaction resources (images, menus…).

The designer has to specify the task model, the interaction model, the java classes (which ensure the various interaction styles) and the presentation resources. The system analyses these elements and, using interaction guidelines and patterns, it maps filtered elements on the resources.

We are aware of the difficulties and limits in considering the definition of a process that is wholly automatic. The complexity of the problem requires simplifications that inevitably lead to stereotyped and non-adapted interfaces to the specificity of materials. The introduction of the adapted task patterns and the interaction ones may decrease the complexity of the issue. However, it is obvious that the contribution of the designer should take place in this type of process. In this context, we will consider the introduction of a constraint definition file. The designer defines this file, which is used to restrict the modifications upon some elements during the process of the dynamic generation of the concrete interface.

REFERENCES

1. Abrams M., Phanouriou C., Baeongbacal A. L., Williams S. M., Shuster J.E., "UIML: An Appliance-Independent XML User Interface Language," In Computer Networks, Vol. 31, 1999, pp. 1695-1708.
2. Calvary G., Coutaz J., Thevenin D., Limbourg Q., Bouillon L., Vanderdonckt J. A unifying reference framework for multi-target user interfaces, Journal of Interacting With Computer, Elsevier Science B.V, June, 2003, Vol 15/3, pp 289-308.
3. Coutaz J.: PAC, an Object Oriented Model for Dialog Design, in Proceedings Interact'87, North Holland, 1987, pp.431-436.
4. Krasner G.E., Pope S.T. A Cookbook For Using the Model-View-Controller User Interface Paradigm in The Smalltalk-80 System. Journal of Object Oriented Programming, 1988, 1, 3, pp. 26-49.
5. Luyten K., Van Laerhoven T., Coninx K., Van Reeth F., «Runtime transformations for modal independent user interface migration». Interacting with Computers. Vol. 15, No. 3, June 2003. pp. 329–347.
6. Markopoulos P, Johnson P. Rowson J. Formal architectural abstractions for interactive software. International Journal of Human Computer Studies, Academic Press, (1998), 49, pp. 679-715.
7. Mori G., Paternò F., Santoro C. « Tool Support for Designing Nomadic Applications» Proceedings of IUI 2003, Miami, Florida, January 12-15, 2003.
8. Paternò F., Model-based Design and Evaluation of Interactive Applications. Springer-Verlag, November 1999.
9. Seffah, A., Radhakrishan T., Canals G. Workshop on Multiples User Interfaces over the Internet: Engineering and Applications Trends. IHM-HCI: French/British Conference on Human Computer Interaction, September 10-14, 2001, Lille, France.
10. Samaan K., Tarpin-Bernard F. « L'Utilisation de Patterns d'Interaction pour l'Adaptation d'IHM Multicibles ». IHM'03, CAEN-FRANCE, novembre 2003.
11. Souchon N., Limbourg Q., Vanderdonckt J. Task Modeling in Multiple Contexts of Use. In Proceedings of DSVIS'2002 Workshop. 2002.
12. Tarpin-Bernard F., David B.T., *AMF : un modèle d'architecture multi-agents multi-facettes.* Techniques et Sciences Informatiques. Hermès. Paris. Vol. 18. No. 5. p. 555-586. Mai 1999.
13. Thevenin, D., Coutaz, J. Plasticity of User Interfaces: Framework and Research Agenda. In Proceedings of INTERACT'99, 1999, pp. 110-117.
14. Vanderdonckt, J., Bodart, F., 1993. Encapsulating knowledge for intelligent automatic interaction objects selection. In: Ashlund, S., Mullet, K., Henderson, A., Hollnagel, E., White, T. (Eds.), Proceedings of the ACM Conference on Human Factors in Computing Systems InterCHI'93 Amsterdam, 24–29 April 1993), ACM Press, New York, pp. 424–429.
15. XIML Forum Site Web. http://www.ximl.org.