ACHIEVING USABILITY OF ADAPTABLE SOFTWARE: THE AMF-BASED MODEL

Franck Tarpin-Bernard(1), Kinan Samaan(1), Bertrand David(2)
LIESP,
(1) INSA-Lyon, F-69621, France
Bat L. de Vinci - 21, avenue Jean Capelle, 69621 VILLEURBANNE Cedex, France
(2) Ecole Centrale de Lyon
36, avenue Guy de Collongue, 69134 ECULLY Cedex France

Abstract: This chapter proposes a novel model-based approach for adating interactive applications to different context while ensuring its usability. After a brief overview of the existing software architecture models for HCI and strategies for adaptation, we detail the different models we are proposing. This includes task, concept, platform and user models as well as an interaction model. All these models are linked via an underlying architecture: The AMF. It ensures the relationships between all the other models and encapsulates the key usability attributes. We will also shows how these models are embedded in a process and amethod for building adaptive software.

Key words: Model-based approach, adaptation, design and implementation method, AMF, Interaction model

1. INTRODUCTION

Maintaining adaptability between platforms while ensuring usability is one of the major challenges from both the HCI and Software Engineering perspectives. Designing and implementing interactive applications that are adaptable (manually) or adaptive (automatically) to the context of use requires to consider the characteristics of the user, the interactive platform as well as the constraints and capabilities of each environment. Several efforts have been made for tailoring an application to a specific context and
especially to the platform constraints. Examples include The Java Pluggable Look and Feel, Web Clipping, and Content Management Systems such as PHP Nuke, ZOPE, etc. However, ensuring the usability is still an open research question. This because the transformation techniques may take into account a specific usability attributes – Most of the time cross-platform consistency – rather considering the overall set of attributes we generally consider in usability measurement (Seffah et al., 2004; 2006).

A state of the art survey shows us that among the large majority of existing approaches for adaptation, the model-based approach seems to be the most powerful. Such approach uses high level and abstract representations that can be instantanciated latter on in the development lifecycle to meet specific usability requirements. However, these approaches need to combine apparently independent models such as concepts (e.g. UML), task (e.g. CTT), platform (e.g. CC/PP) or user profiles. The relationships between these models need to be defined at the design step and refined at run-time in order to be able to achieve the overall usability. Our belief is that, what we refer to as an interaction model is the right place to glue together all the other models and usability attributes. This model must support both design stage linking other models and run-time. In addition, because Software Engineering and HCI shown the importance of clearly separate functional core from presentation components, our interaction model is supported by well structured architecture.

Resulting from the fusion of well-known models either layer-based like Arch (UIMS, 1992) or multiagents like PAC (Coutaz, 90), our architectural model, called AMF, has been implemented in the format of an engine that at run time execute the interaction model which links the abstraction and presentation components. AMF combines best architectural practices, such patterns and specification notations, from the software engineering and HCI communities. As an example, we use UML models (from use case to class and sequence diagrams) and user-centered approaches based on task analysis.

2. STATE-OF-THE-ART

The architectural model is one of the key elements needed to achieve efficient and good software developments: methods - models - tools. Firstly, it organizes software structure to improve implementation, portability and maintenance. Secondly, it helps identify the functional components, which is essential during the analysis and design process. Its third role is to help further understanding of a complex system, not only for designers, but also
for end-users. For these reasons, the architecture model is the pivot of the lifecycle and we consider that a good model should fulfill four main goals:

- Support specification step (as a formalism);
- Be the skeleton of the implementation (as a framework);
- Insure consistency for executable applications;
- Serve as a representation for dynamic reconfigurations by the user.

Different approaches have been proposed to support various contexts of execution, including a wide range of devices and various user profiles. In order to support plasticity (Thevenin, 2001) - that is the ability to adapt itself to context without compromising usability - different approaches have been proposed.

This set of approaches suggests first revisiting some of the architecture models developed in the early 80. This tendency leads to huge improvements, like ARCH (UIMS, 1992) or multi-agents models like MVC (Krasner et Pope, 1988), PAC (Coutaz, 1990) and PAC Amodeus (Nigay et Coutaz, 1993). Some others researchers suggest also XML-based languages for specifying HCI and rendering engines as a mechanism of adaptation.

### 2.1 Interactive system architecture in HCI

Most of existing HCI architectural models distinguishes at least three main components:

- Presentation or views that manage the direct interaction with the user;
- Controllers, adaptors which are in charge of the communication between the the other components and/or with the users
- Abstraction or model which serves as an interface between the functional core and the two others elements

These various models have been presented differently in the literature and with different names. Here we use the taxonomy that classifies the architectures in three categories: layer-oriented like Seeheim (Pfaff, 1985) and Arch (UIMS, 1992), multi-agents like MVC (Krasner et Pope, 1988), 1984), PAC (Coutaz, 1987) and AMF (Ouadou, 1994), and hybrid like PAC-Amodeus (Nigay et Coutaz, 1993) or H4 (Depaulis et al., 1995). Layer-oriented models divide architecture in logical layers. Multiagents models exploit the layer-oriented model at and define each layer in the format of a set of agents. Hybrid architectures combine the advantages of the two previous approaches while combining a layered architecture where the dialogue components are structured using agents.
Although most of the existing integrated development environments (IDE) implement some of these architectural models, the Cartesian separation between abstraction and presentation still not fully achieved. Efforts are needed to support adaptation in particular hardware diversity. Some progress have been made towards this objective especially with the advent of platform independent scripting languages C# and the related standards for describing devices like CDC (SDNa), CLDC (SDNb) or CC/PP (CCPP).

2.2 Adaptation approaches

Adaptation techniques can be classified in four categories ranging from the most easiest to implement to the most powerful (Samaan, 2006):
- Translation techniques;
- Markup language-based approaches;
- Reverse and re-engineering techniques;
- Model-based approaches.

User interface translation is a technique widely used in the context web pages. There exist also tools for HTML <-> Java translation. These techniques generally provides insufficient provisions to support usability except in the context where the adaptation contexts are very similar, nearby the same. New approaches such as graceful degradation (Florins et al., 2004) use the specification of a user interface of the “best” platform – The one with
the highest screen resolution and the most powerful graphical toolkit. During the design, adaptation rules called degradation rules are used by developers to adapt the best interface to a specific platform. However, this approach is limited and to the translation from one specific language to another one available on similar platforms.

Markup languages-based approaches define platform independent descriptions with languages that can be easily reused for large variety of contexts. During the last five years, many UIDL (User Interface Description Language) have been introduced. They usually use XML and CSS scripting languages. Rendering engines are proposed to analyze the independent descriptions of UI. They produced a platform dependent description files using specific technologies (HTML, WAP, VoiceXML, etc.). Popular languages include XUL (Hyatt et al., 2001), XForms (Dubinko et al., 2000), AUIML (Azevedo et al., 2000), PlasticML (Rouillard, 2003), RIML (Koskimies et al., 2004). These languages can be classified in two different categories. The first one groups languages that define Abstract Interaction Objects (AIO) which are replaced by Concrete Interaction Objects (CIO). For instance, in UIML, a <part class="Button " id="MyButton"/> tag is replaced by a <Button name= " MyButton"/> tag in HTML and a JButton object in a Java Swing rendering. The other family provides a higher level of abstraction specifying user’s interactions like “select element” or “select command”. XForms and AUIML will use a choice tag that will be concretized by a set of radio button or a scrolling menu according to the device characteristics.

Reverse engineering and interface migration techniques analyze an existing UI description with the perspective to extract abstract representations (language-independent and device-independent). These representations are then instanciated for another platform. Firstly introduced for the migration of text applications to graphical ones (Chikofsky et Cross, 1990), these representations are now generally based on the markup languages presented above (e.g. VAQUITA (Bouillon et al., 2004)).

All these approaches are useful and efficient if and only the initial context used for designing the application and the real contexts have some similarities. In all the other situations, it becomes very hard to warranty the usability of the application. The situation is more drastic in the context of interactive applications that are not form-based like the web and wap applications usually modeled.

An answer to these limitations consists of using a set of models (tasks, concepts, presentation, dialogue, platform, etc.) to describe the application at a high level. For example, UIML (Abrams et al., 1999), UsiXML (Limboung et al., 2004), ArtStudio (Thevenin, 2001), TERESA (Mori et al., 2004), dygimes (Luyten, 2004), Comets (Calvary et al., 2005) use different models. The abstract Interface, the most abstract model, is transformed step
by step into a concrete platform-dependant Interface according to the information stored in the set of models.

Fundamentally speaking, as these approaches considered a very large set of parameters (in the various models), the resulting Interfaces are presumably of better quality with a higher usability. Furthermore, they support a certain level of adaptation when the context of use is evolving (change in the user, the interaction platform, the environment or the activity). However, these techniques are harder to implement mainly because of the difficulty to relate all the models together.

We proposed here a model-based approach and an underlying the software architecture model as a way to achieve both usability and to maintain the same level of adaptability. In the next section, we are presenting the AMF model, its extensions for adaptation in the context of a model-based approach.

3. AMF AND ITS RELATIONSHIPS WITH OTHER MODELS

3.1 AMF Fundamentals

AMF extends multi-agent models like PAC while generalizing the concept of facet while embedding a set of coherent behaviors and functionalities. AMF proposes also a graphical formalism mainly dedicated to the representation of control using standardized UI elements. The number of new facets can be unlimitedly defined (such as Help, Error, Distribution, Rights, etc.) in addition to the classical PAC’s facets (Presentation and Abstraction). Similar to Object-Oriented approaches, AMF eases top-down analysis through an iterative decomposition of the application into facets at different levels.

The agent is the basic element for structuring an AMF-based application. Each agent can contain other agents and several instances of a same agent class can co-exist. Thus, an application is composed of a hierarchy of agents, which root is the main application manager agent.

Each agent is composed of several types of facets that group a set of services. Each service can be reached through a communication port. Three kinds of communication ports exist: input (I), output (O) and I>>O. Input ports represent services offered by the facet. On the opposite, an output port represents a required service. An I>>O port first serves as an input port and then as an output port.
To model the control facet, AMF defines special elements called **Administrators**. These administrators play three major roles:

- The interconnection between the ports,
- The execution of a treatment on the data exchanged by the ports,
- The handling of activation rules that determine the listened sources and with an eventual listen order (e.g. first, then second, then …) and the targets to notify.

Depending on their types, the administrators can have several source ports and/or several target ports. AMF formalism contains several types of administrators; we describe here the three most used ones (Figure 2). The basic administrator is used to build a unidirectional link without any special treatment. The **Return administrator** is like the Simple administrator but it carries back the result returned by the activation of the targets ports. The result can have several forms depending on the number of targets connected to the administrator (Single result or Array). The **Filter administrator** allows handling the activation of the target ports of a collection of Agents and to select the most appropriate ones to activate. This allows using multiple instances of an Agent, especially useful when a collection of items (each one represented by an Agent) is dynamically managed (e.g. Appointments of a Schedule) or whereas only some of targets must be activated depending on activation conditions (e.g. the window having the focus).

<table>
<thead>
<tr>
<th>Simple</th>
<th>Returning value</th>
<th>Filtering Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Figure 2: The main AMF administrators

The **control propagation** is done accordingly to the AMF tree and can be done only between ports of facets owned by the same agent, or of ports of facets of sub-agents. In this last case, the propagation is done by the administrator of the same agent, the one containing the facet of the source port. The set of administrators with their links that owns an agent constitutes its Control Facet as defined in PAC.

Figure 3 presents a simple interactive agent using the AMF formalism. This agent aims to provide feedback to the user when s/he uses a particular functionality of the application. This user event is represented by the bolt entering the facet and it leads to the activation of the ‘Start_Action’ *output port* of the *facet* ‘Presentation’. The **Simple administrator** ‘A1’ which has this port as a source dispatches the event to its unique target port ‘Do_Action’. As soon as the service has been executed, the ‘Do_Action’ *I>>O port* sends an event as a result of its activation. Being connected to
‘A2’ *Simple administrator*, the message it sent is thus dispatched to activate the ‘Echo_Action’ *input port* that will provide the feedback to the user.

![Diagram](image)

Figure 3: A simple interaction described with AMF

### 3.2 AMF Implementation

#### 3.2.1 Hybrid architecture

AMF is supported by several tools including an editor (or more exactly a graphical development environment) and an execution engine. To implement the AMF architecture, the AMF facets play the role of adapters (in the meaning of Arch, Figure 4; Samaan and Tarpin-Bernard, 2004a). Thus these classes are concrete facets whereas AMF facets are abstract.

![Diagram](image)

Figure 4: The AMF and the ARCH models links

The application control and the adapter facets are finally described within the AMF formalism. The concrete facets like the concrete presentation and the functional core are developed in applicative code. The link between AMF facets and the applicative classes is materialized by the communication ports. Indeed, the ports are associated to functions of the applicative classes. We call these functions daemons. The activation of a port triggers its daemon. Figure 5 synthesizes the meta-model of AMF in hybrid architecture.
3.2.2 The AMF engine

By AMF engine, we designate a software process that builds the application and ensures its correct behaviour. It links the following elements:

- The application functional core objects, whose role consists in handling the data and performing operations (applicative domain objects);
- The application concrete presentation objects (applicative presentation objects);
- The “AMF objects” themselves, used to manage the communications between the applicative objects (administrator, facets, agents, etc.).

Building the application consists in the setup of the agent hierarchy. The instantiation is done in the following order from the application (top-level) agent: sub-agents, facets, daemons, ports and then administrators, in a recursive way.

The execution of an application built with AMF requires to instantiate and to initialize not only the applicative objects but also the AMF architecture objects (Agents, Facets, Administrators and Ports). This is one of the roles of the engine to perform this loading operation. It instantiates and loads all objects required to the application execution. To determine which objects are required, and then to link them, the engine starts by parsing the control facets description files, that are written in XML. The applicative objects are indicated in the XML control files as URLs. Of course, the engine can also build objects on demand during run-time.

Then, when the user interacts with the application (button activation, menu selection…), the presentation concrete facet (e.g. the JFrame class in Java) activates an output port of its abstract AMF facet. This activation launches the event dispatch process, which normally ends by the activation
of an input port of another AMF facet, which triggers the daemon implemented in an applicative object (the concrete facet).

![AMF Engine Diagram](image)

**Figure 6**: Links between AMF and the applicative classes (concrete facets)

### 3.2.3 The editor

The editor is the AMF instrumentation entry point. Indeed, AMF having a graphical formalism, the first step is to elaborate the architecture model. This editor (Figure 7) allows building graphically the AMF-based applications, while making possible the description and integration of AMF templates, components and configurations.

In addition to the production of an XML description of the architecture model, the editor fully generates the control of the application (the control facets for a hierarchy of agents), i.e. the links existing between the ports of the facets. This functionality is tremendously interesting, as, even if possible, a handmade production of this description is tedious and requires memorizing the description elements names to able to link them.
Besides, the editor generates the skeleton of the Java source files of the concrete facets. This process is comparable to a UML MDA process, which creates the structure of the user objects. In our editor, the source files structure generation consists in creating Java packages and classes with public methods that will be the daemons of input and input/output ports. These methods are currently built with an empty body that remain to be filled by a programmer to achieving the application creation. Currently, the editor allows inputting the application source code but in a rudimentary way and we advices the programmer to use a Java IDE like Eclipse for that. When the code is written and compiled, it is possible to run the application from the editor invocating the AMF engine with the application XML control file as an URL (this file contains the root agent control description which links other control description files depending on the application). In doing so, the developer can check the application behaviour directly from the editor. As the control part is separated from the sources files, the developer can then modify the control (changing administrators and modifying ports interconnection paths), until the application as the expected behaviour.

3.3 Links with other models

In this section, we are describing the relationships between AMF model and respectively domain and task models.

3.3.1 Interaction model and domain model

The domain model describes all the concepts of the functional core of the application and their relationships. Nowadays, its modeling generally relies on the elaboration of UML models and especially class diagram, which describe the static structure of the application in terms of classes and
relations (association, specialization, aggregation, composition and dependence).

We said that in AMF hybrid architecture, each facet is associated to a concrete facet called applicative class. This class contains all the method that are associated to the ports of the abstract facet and additional elements (data and internal functions). As a consequence, the domain model represented by the UML class diagram has usually the same structure than the AMF model; however it is not always the case. To help the designer, we have identified 3 situations:

- Agents are autonomous and each applicative class is associated to a AMF facet. In this common case, there is a bijective relationship and a full symmetry between both models.
- An agent can use specific services realized by a set of applicative classes that are not supposed to be visible and accessible (application of the Facade pattern of Gamma et al., 1995). In this case, these extra classes are not represented in the AMF model. The considered agent is the only element that has a AMF representation and the symmetry is partial.
- Several agents use a complex set of external applicative classes (e.g.: database access, calculation library, etc.). In this case, the external applicative classes are grouped in a package and a dedicated agent is added to the AMF model to ensure the links between the agents and the external services.

In the first case, the UML relations between classes (association, aggregation, etc.) can not be maintained. Indeed, the flexibility and the powerness of AMF require not having direct coupling of applicative classes so that every communications use the AMF engine. Thus, we have proposed rules of translation of the relations between UML classes to AMF model:

- The aggregation of 2 classes is a composition of AMF agents;
- The association between 2 classes leads to the definition of control administrators and communication ports associated to the services that are supposed to be invoked through the association.
- The specialization relations between classes are maintained.

### 3.3.2 Task model and Interaction model

Interaction tasks are naturally associated to the communication ports of Presentation facets. Similarly, computer tasks can be clearly associated to abstraction facet. Normally, these tasks are also directly linked to the domain model.

In our work, we are using CTT formalism (Paternò, 1999) to model tasks. Figure 8 represents the relationships between tasks (interaction and computer oriented) and the ports of Presentation and Abstraction facets in a sample
application. This application is a music player that provides classic features like Select_Title, Play_Title, Stop_Title…

![Diagram](image.png)

**Figure 8**: Example of relationships between tasks and ports in the interaction model

In model based approaches, task model is often the starting model for the design of an interactive application. This model is very flexible in terms of level of details of the modeling. Indeed the specification can be very high (almost functional), e.g. change the volume, or precise, e.g. key ‘up’ / key ‘down’. For this reason, we are considering two versions of the model: the abstract and the concrete models.

The abstract task model represents a generic view of the application, which is independent from the context of use and more specifically to the interaction platform. If we go back to our sample, the modelled tasks are abstract tasks because they do not define the elementary interactions. From this model, it is possible to build and abstract AMF interaction model using what we call abstract ports. These **abstract ports** describe services that must be provided without specifying either how they will be concretely implemented or their relations with other elements. On AMF graphical formalism, abstract ports are represented with a dash border.

In the concrete task model, each leaf of the task tree represents a concrete interaction with objects of the user interface and is generally detailed enough for identifying physical interaction (mouse click, key pressed, etc.). This model is context-dependent. Passing from abstract to concrete task tree supposes making some interaction choices and replacing abstract task by sub-trees of concrete tasks.

In the music player sample, if we consider an interaction platform that only provides a keyboard as interaction mean (it is a very simplified context
but sufficient for the illustration), the abstract task *Select_Title* could be replaced (Figure 9) by the following sub-tree:

- *Up/down*: to move into the list of titles;
- *Validate_Title*: to validate the choice.

![Diagram of Select_Title sub-tree](image)

Figure 9: Example of concretization of an abstract task and the associated abstract port

This concretization operation is a repetitive task that can be assisted by a design tool. Indeed, many tasks are repeated similarly in a design. The next section introduces our vision for the use of task and interaction patterns in the design and implementation process.

### 3.3.3 Task and interaction patterns

The AMF interaction model has been designed to support the patterns-oriented design approach (Gamma *et al.*, 1995). Indeed, fragments of models (agents, facets, ports and administrators) are, by definition, potential patterns. They define a validated solution to a well-defined problem. We call these patterns **interaction patterns**. Thus, several patterns have already been defined (Tarpin-Bernard *et al.*, 1998) but most of them still need to be identified.

In the specific area of interaction adaptation, we have defined several patterns for designing the same generic task in various contexts (Samaan and Tarpin-Bernard, 2004b; Samaan, 2006). For instance, we have model the very common task of moving an element inside a container (e.g. an icon in an area, an item in a list, etc.). The design principle is classical: 1) the container object received the notification of a request of selection of an element, 2) the element is identified and the validity of the potential move is checked, 3) the destination is defined, 4) the move is validated, 5) the display is refreshed.

Figure 10 shows a pattern that realizes this abstract task and two concrete variants corresponding to two different contexts of use. This interaction pattern has a *Container* agent that contains component agents (multiple
instanciations) which can be moved \((Element)\). The abstract port \((Select&Move)\) of the \textit{Presentation} facet of the composed agent receives the user action events and transmits them to the component agents. The abstract ports and administrators are replaced by concrete elements in the concrete version of the pattern corresponding to different contexts of use (here specific interaction devices).

Figure 10: A pattern for the abstract task « Select and Move an element »
4. A METHOD FOR DESIGNING ADAPTABLE APPLICATIONS

4.1 Process

Like most of model-based approaches, our method consists in splitting the design and implementation process in several steps (four in our case).

In the first step, abstract task model and domain model are elaborated. The second step defines the abstract interaction model that represents the general structure using AMF. The third step is a concretization step which leads to a concrete AMF model according to the context of use. Finally, the application is instanciated and executed. In the following section, we are going to describe the main techniques we are applying in these steps.

Step 1: Elaboration of task model and domain model

This step is the most fundamental one. Using techniques generally based on use cases identification, it is possible to build UML models (class diagrams, sequence diagrams, activity diagrams) and task diagrams. S/he will start by building UML or task diagrams. Because many books are dedicated to this first part of the process, we will not detail it. However, we underline that at this stage the task model is abstract which supposes to have generic tasks.

At the end of this step the designer has an abstract task model and a domain model. Some extra models can also be defined at this stage (user model, environment model, etc.). In collaborative situations, it is usual to define at this stage a model for specifying roles and rights.

Figure 11 shows a part of the models resulting from this first design step applied to the music player. The abstract task model is on the left side whereas the UML class diagram of the functional core is on the right side. The models are not yet interconnected.

This approach is flexible enough to respect designer’s habits and culture (software engineering vs. HCI ergonomics). In our own works, we use to start by the task model.
Figure 11: Task and domain models after the first design step (music player sample)

Step 2: Abstract interaction design

This step consists in linking previous models while building the abstract AMF interaction model.

The temporal logic of the task model usually organizes interactions in a main modal flow defining interaction environment, which are navigation blocks. Inside each block, the interaction is modal or not (no forced sequence between the tasks).

In order to identify these environments, we currently use the task grouping feature of CTTE. Thus, considering a specific context and thanks to heuristics, the designer can obtain a set of PTS – Presentation Tasks Set – which are the tasks that should be accessible simultaneously. These interaction environments are represented in the interaction model by AMF agents that are not naturally identified in the domain model.

However, the domain model is very useful to help organizing the agents. The rules presented in section 3.3.1 lead to structure the hierarchy of agents and define the content of abstraction facets. Then, based on the rules presented in section 3.3.2, the designer can identify abstract ports for each presentation facets postponing any decision about concrete interactions.

As a consequence, the interaction model refines elements from the task and the domain models. An explicit relationship can be maintained, which will be very useful later if we want to adapt dynamically the application.

The next step consists in instantiating the models to a specific context.

Step 3: Models Concretization

The purpose of this step is to replace abstract interactions by concrete interactions. As seen in section 3.3.3, the designer should be helped in this task by a library of interaction patterns related to contexts of use. Once a pattern is chosen, the abstract ports are replaced by concrete ports and
administrators. Currently, we have not yet defined heuristics or selection rules but this will be very helpful for the designer.

Then, these concrete AMF facets are associated to applicative classes (ports are linked to daemons) that have been implemented from the code generated by the editor or which respects some programmation rules.

In section 4.2, we will show that several design options can be taken by the designer to handle the adaptation issue that will impact either implementation or just the interaction model.

Once the concretization step is fully over, a concrete interaction model description (XML file) can be produced by the editor.

**Step 4: Application execution**

This run-time step, sometimes called finalization step, is responsible for instanciating application elements (AMF agents and instances of applicative classes). In our Java implementation, this happens inside a specific component called **AMF engine**. This software component is the heart of a AMF application and ensures 2 main functions:

- Loading the concrete interaction model description files defined in the previous step and instanciating the referenced objects (AMF objects and applicative classes).

- Controling the interaction behaviour of the application processing and routing messages between methods of applicative classes thru AMF objects (ports, facets, agents and administrators).

The «concrete» facets that are the applicative classes (presentation, abstraction and others) are associated to the AMF engine to provide finalized behaviours. This assembling task is monitored by the AMF engine according to the interaction description files.

Here, we do not directly study the question of the layout of the presentation as this part of the problem is mainly managed by the applicative classes. However, in the next section, we will give some key answers.

### 4.2 Adaptation options

After having defined a design process and some building rules, we focus our attention on the adaptation options available for the designer. These options depend mainly on the adaptation goals and on the difference between the target contexts. These contexts are defined by the user, the interaction platform, the environment and the activity. We call distance between two contexts of use the set of differences between both contexts. Even if no
metrics are available, this notion is useful to understand when choosing one option or another.

Today, we have identified four strategies for supporting the adaptation. All of them can be combined in a real application according to the required level of adaptation.

First of all, when the distance between two contexts is small, that is when the adaptation does not require modifying the structuration of the AMF model; it is common to support it inside the presentation applicative classes. For instance when two interaction platforms own the same input capabilities but have small display differences (resolution, colors…) the adaptation of the size or the layout will not impact the AMF model. Actually, it supposes that the applicative class manages all the possible layouts (which is very similar to the Comet approach) or is enable to be parametrized by external description files (see UIDL approaches). In these situations, it is not necessary to modify any connection in the AMF model.

If the target interaction platforms have similar output devices and various input devices but which support similar interaction techniques, the main infrastructure can be maintained but dedicated presentation facets will be inserted. These Device facets interact with the real devices. This is particularly useful with non standard devices such as data glove, eye trackers or RFID readers (Masserey et al., 2005).

These two first strategies are applied in the last steps of the design process as no significative modifications in the AMF models are required.

When the adaptation implies bigger changes such as another interaction style (e.g. drag-drop with a mouse vs. keyboard interaction) it is necessary to use different communication ports and sometimes new control administrators. As seen earlier, interaction patterns can really ease the designer’s task. These patterns can involve one single agent or a small hierarchy.

In the hardest situation, when the adaptation distance is so high that a deep change need to be done, a total restructuration of the AMF model can occur. Of course, the designer does not need to start from scratch as he can reuse large part of the interaction model. This kind of adaptation requires making some changes in the second step of our design process using two kinds of actions:

- Filtering tasks that are not achievable in the context (removing abstract tasks in the tree),
- Restructuring interaction environments (grouping differently the remaining tasks).

For this last action, the designer can use tasks grouping heuristics proposed by CTTE (Mori et al., 2002). This way, he can obtain adequate groups of tasks which increase the usability of the resulting application.
Indeed, a PC with a large screen is able to display in a single interaction workspace a larger number of tasks than a mobile phone with a small display. Thus, the designer can apply series of grouping steps to reach a satisfying structuration according to the context characteristics, including non platform properties such as user’s preferences and abilities.

In the future, we are going to study other techniques to restructure the interaction workspaces. Particularly, we would like to consider the intermediate tasks (virtual/real nodes in the tasks tree).

5. CONCLUSION AND FUTURE DEVELOPMENTS

In this paper, we presented a model based approach organized around AMF, a multiagent model, to achieve adaptability of interactive software as one of multiple aspects of usability. Mainly, we have elaborated an environment composed of a model builder and a runtime engine. These technical tools are associated to a design and implementation process. This helps integrating into a AMF architecture the various models that need to be considered to provide a good adaptation to the context of use (platform, application and user preferences). Several techniques, mainly based on patterns, can assist the designer in choosing the “best” interaction techniques (in a usability perspective) and easily implement them.

During the description of AMF model we did not really develop the interest of its multifacet characteristic. As mentioned in several other chapters in this book, usability is not only interaction oriented, but can be concerned by no visible software objects. With AMF, we can consider presentation, control and abstraction parts, not only at a very large grain, but also at a thinner grain, which allows doing a richer modeling. Indeed, we can collect interesting and reusable behaviors, which can constitute new facets and be reused elsewhere. This way, we allow designers expressing what they consider as new important aspects of the interactive application by creating new facets, which would be reused later by themselves or other designers. The goal of these new facets is also to draw attention during the development process and mainly during usability studies, i.e. answer time for SQL commands which must respect delay constraints to ensure usability of the application. These new facets can be of different natures, i.e. presentation, to create several, alternative presentations for different devices, or different users (mainly in collaborative applications), user’s profile, to collect information on user’s behavior, presentation preferences, main interactions undertaken and their contexts, in order to be able to determine en appropriate interaction presentation for the user; undo mechanism, in order to indicate the way to stop and undo executed
commands, and others, which are explained by creation of new reusable facets. Patterns can be created to express the relation between a new facet and its activation coming from control expressed by appropriate administrators in relation with their ports, can also be created and used in other application designs.

From an architectural point of view, these facets can receive during the adaptation process a particular attention, in order to determine appropriate organizational answer, i.e. specific location of this facet in client – server architecture, location determined by usability requirements as acceptable answer delay. To obtain it, it is possible to locate the facet at the same place as caller and to connect it directly (by procedure call), or to locate it elsewhere and use distant procedure call, soliciting middleware for the call coming from distant facet (see AMF-C; Tarpin-Bernard et al., 1998). This deployment aspect is studied at implementation stage and still open-ended for use (execution) phase adaptations.

Connection between user’s actions (interactions) and this new facet and/or between behavior (functional core) facet and this one is expressed graphically at control level with appropriate administrator(s). This visual programming of control can be manipulated by the designer during elaboration process phase as well by the user (in a simplified view to define) during the use (execution) phase. These functionalities are not yet fully operational and constitute interesting future developments.

Finally, we are currently working on the integration of AMF builder into Eclipse environment so that we can enforce the relationships between AMF model and other views, mainly UML and the visualization of the interface.

6. REFERENCES


Interfaces with XML: Advances on User Interface Description Languages, K. Luyten, M. Abrams, J. Vanderdonckt and Q. Limbourg (eds), Gallipoli, Italy, 2004, pp. 71-78.


CCPP - W3C Composite Capabilities/Preference Profile. Online: http://www.w3.org/Mobile/CCPP/

