Enabling *Adaptivity* in User Interfaces ⋆

Javier Cámara[1], Carlos Canal[1], Javier Cubo[1], and
Juan Manuel Murillo[2]

[1] Department of Computer Science, University of Málaga
Campus de Teatinos, 29071. Málaga, Spain
{jcamara,canal,cubo}@lcc.uma.es
[2] Dept. of Computer Science, University of Extremadura, Spain.
Avda. de la Universidad s/n, 10071. Cáceres, Spain
juanmamu@unex.es

Abstract. The development of adaptive user interfaces has traditionally been restricted to research prototypes and few commercial products. Although there have been relevant achievements in the architectural support for self-adaptive context-aware systems [3, 19], the notion of context commonly supported is restricted and does not explicitly contemplate the facets of context related to user-application interaction. Furthermore, applications need to comply with the proposed architectures, making the incorporation of adaptivity more difficult (or not possible at all) in the case of already existing applications. This work addresses key issues for the incorporation of self-adaptive behaviour in GUI-Based applications, and proposes an aspect-based framework in order to overcome current limitations.

1 Introduction

As computing applications become more complex and sophisticated, users tend to spend more time and effort trying to instruct and configure them, having to explicitly state an ever-increasing amount of information in order to efficiently carry out the tasks they are demanded. This happens because computing applications are not context-aware entities, and must be supplied with additional (context related) information that we, as human beings, implicitly assume in different situations. Contextual information has a dynamic nature and changes as we interact with our environment. The notion of a changing context dependent on user-application interaction has been subject to research by the Computer Science Community by many years [18], producing a broad range of interactive systems with the same philosophy in common: that it can be worth learning something from the user and adapt to it in some non-trivial way. This kind of system has been labelled in different ways, ranging from *personalized systems* to *Intelligent User Interfaces* (IUIs). We will refer to them as *User Adaptive Systems*, or more specifically as *Adaptive User Interfaces* (AUIs) [11]. These contrast

⋆ This work has been partially supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and project P06-TIC-02250 funded by the Andalusian local Government.

with traditional user interfaces, rigid and passive, which allow only a small degree of user customization by setting preferences. Making interfaces flexible and adaptable to the user implies the extraction of a *user model* by retrieving information based on the interaction of the user with the interface. Then, based on predictions made using that information, the system modifies its behaviour and structure for a better interaction with the user.

Dey and Salber define context in [19] as: *Any information that can be used to characterize the situation of an entity. An entity is a person, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.* Specifically, in our proposal we deal with two types of context-related information:

- *Application context*: Information related to UI description, application domain, and task specifications.
- *User context*: Information related to user behaviour and preferences.

There is a third kind of Contextual information relevant to the application referred to as *environmental* or *sensed* [7]. This information is indirectly related to the behaviour of the user and includes location, time, etc. However, in this work we focus just on application and user context information, since the use of environmental context has been broadly studied in the field of context-aware applications [2]. In this work we advocate for the adoption of a new architectural approach to the development of adaptive interfaces based on *Aspect-Oriented Programming* (AOP) [17], which provides the foundations to enable adaptive behaviour in applications which have already been deployed. Hence, rather than dealing with user modeling or learning algorithms, this work is focused on how aspects can be used to apply such techniques on already existing systems, extending them with adaptive capabilities.

The rest of this paper is organised as follows: Section 2 points out the main issues for the incorporation of adaptive behaviour in UIs. Section 3 and 4 describe and detail the design and implementation of the framework. Section 5 describes some related work. Finally, section 6 discusses the benefits our proposal purports as well as some open issues.

2 Issues Reusing and Enabling Adaptivity

(i) Poor reusability. Building an AUI involves the inclusion of additional requirements to the system, such as user modeling techniques (which implies retrieving information derived from the interaction with the user) or dynamic reconfiguration of structure and behaviour. Developing these mechanisms is costly, and often requires the use of sophisticated techniques specifically tailored to each individual application. As a result, the application code which corresponds to the adaptive mechanisms is tightly coupled with the rest of the application, hampering reusability.

(ii) Lack of transparency. Even when the architectural support for the AUI provides a good modularization, AUIs are generally tailored for an specific ap-

plication [10, 20]. In such a way, the effective reuse of new behaviour is affected by its underlying representation.
(iii) User model fragmentation and redundancy. Different applications tend to have different user models. Moreover, each application retrieves and stores its own information locally, and with its own representation, impeding the reuse of this information by other applications. This results in a fragmented and heterogeneous user model which only reflects a partial view of the user's behaviour and preferences to different applications.
(iv) Lack of coordination. Conventional techniques do not consider the possibility of coordinating adaptation between different existing applications located within a shared context. In such a way, applications cannot communicate with each other in order to carry out tasks collaboratively.

3 Framework Architecture

This section describes our aspect-oriented framework, intended to enable adaptive capabilities on passive GUI-based applications. The architecture of this framework is structured using a modified version of the the *Adaptability Aspects* [4] architectural pattern in order to provide better maintainability and modularity. Adaptability Aspects are applied to a base application for the adaptation of its interface. As it can be observed in Figure 1, the framework incorporates the following functional elements:

- **Context Manager.** Identifies context changes and triggers adaptive actions implemented by the aspects. It constantly monitors user input through the `UserMonitor` aspect, and the UI through the `UIMonitor` aspect, identifying the structure, properties, and relations between its components.
- **Adaptation Data Provider**. Consists on a set of classes which manage the information related to the different models required for UI adaptation and its processing. Specifically, the **User Model** holds up information about user context, while **Task Model**, **Domain Model**, and **UI Model** comprise information about application context. These elements provide the input to the **Reasoner** module, where the specific adaptive logic is implemented (learning and inference). The reasoner produces an **Adaptation Model** as the result of the application of the adaptive mechanisms, which is used as a specification for the adaptation to be performed on the interface.

Adaptability Aspects adapt the interface using the adaptation model whenever they match user-generated or UI events. For that purpose, they use a set of **Auxiliary Classes** intended to improve reusability. These provide common mechanisms for the addition, modification, or removal of UI components.

Figure 2 depicts how the elements of the framework interact when a context change triggers an adaptation on the application behaviour:**(a)** The application starts execution.**(b)** The context manager begins to monitor the context continuously.**(c)** Whenever an event in the UI is detected, both adaptability aspects and adaptation data provider are notified.**(d)** The adaptation data provider

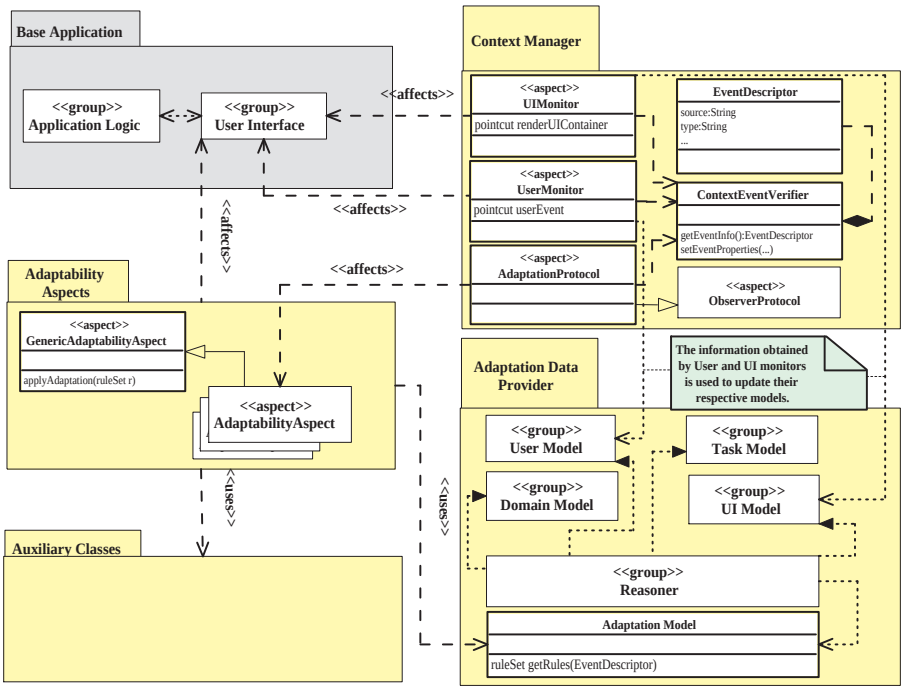

Fig. 1. Framework Architecture. Stereotypes `<<affects>>` and `<<uses>>` are used in some dependency relationships to represent classes whose behavior is monitored or changed by an aspect, or those used as auxiliary classes by an aspect, respectively.

updates the adaptation model.**(e)** Adaptability aspects access the adaptation model to verify if adaptation should be performed. **(f)**Aspects change the application behavior making use of the auxiliary classes. It is worth noticing that when adaptability aspects are notified about a context change, the adaptation to be performed is not carried out inmediately. On the contrary, adaptive behavior is introduced on the base application just after the adaptation model has been updated.

4 Detailing the Framework

Currently, there is a wide range of toolkits which can be used in order to build GUI-based computer applications, such as KDE/Qt, GNOME/GTK/GTK+, MFC, JFC, etc. All of them provide similar abstractions and basic mechanisms both for the programmer and the user. Although our approach is applicable to other toolkits and aspect languages, we use JFC (specifically Swing) and AspectJ[12] in the implementation of our prototype for their widespread use.

Within the **Context Manager**, the `UIMonitor` aspect obtains information about the structure, properties, and relations between UI components. In Swing

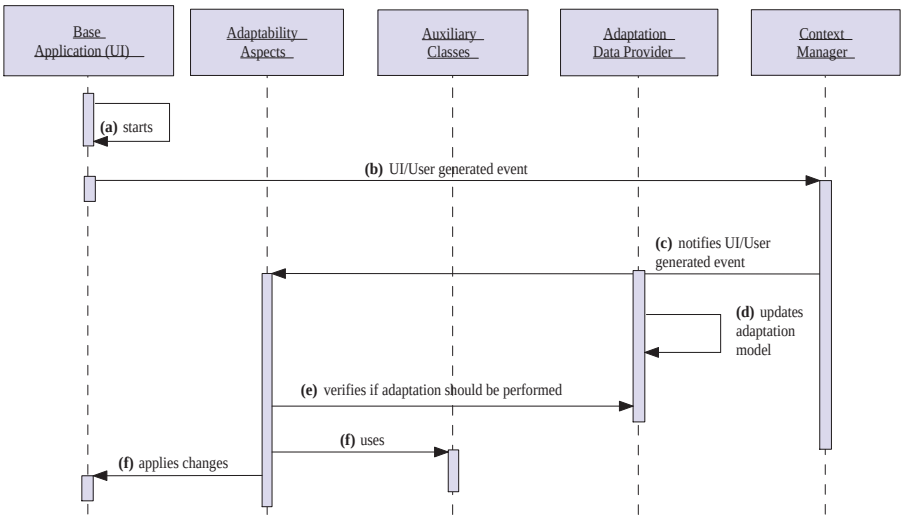

Fig. 2. Dynamics of the process followed for UI adaptation.

and other toolkits, components can either be regular, or have the capability to group other components together. Such components act as *containers* and include *panels*, *windows*, *frames*, and *dialogs*. The structure of a container (*e.g.,* a window), along with its nested components in a hierarchical structure is obtained defining a `renderUIContainer` pointcut to match the join points where a container is rendered. It can be observed how the `Container` object is exposed to the body of the advice applied to the join point. The inspection code in the advice builds recursively a structure with all the child components of the container, which is incorporated into the UI Model along with component properties.

```
1:  aspect UIMonitor {
2:  java.util.List myComponents=new java.util.ArrayList<Component>();
3:  pointcut renderUIContainer (Container c):
4:     call (* java.awt.Container.setVisible(*)) && target(c);
5:  void getComponentList(Container container,
5:                        java.util.List<Component> components){
6:     for (Component component : container.getComponents()){
7:        if (component instanceof Container)
8:           getComponentList((Container) component, components);
9:        components.add(component); }
10: } ...
11: after (Container c): renderUIContainer(c){
12:       if (!UIModel.getComponent(c.getName())){
13:          UIModel.add(getComponentList (c, myComponents));
14:          myComponents.clear(); }
15: }
16: }
```

The extraction of the properties of each of the components in the structure is realised through reflection. It is worth mentioning that textual information is of special relevance to the purpose of our framework. This kind of information is present in almost any UI component. Note that *Menu Items*, *Buttons*, *Labels*, *CheckBoxes*, etc. have well defined properties (*e.g.,* `Text`, `ToolTipText`, etc.) which provide semantic information about the role of the component in the application.

The implementation of the **User Monitor** as an aspect provides a way of retrieving user implicit information unobtrusively. While interacting with the UI, every time the user types a character or clicks on an object, an event occurs. To detect user action on the interface, an object must implement the `ActionListener` interface. The program must register this object as an action listener on the button (*i.e.,* the event source), using the `addActionListener` method. When the user clicks the button, it fires an action event. This results in the invocation of the action listener's `actionPerformed` method. The argument to the method is an `ActionEvent` object that gives information about the event and its source. The `userEvent` pointcut is defined in the `UserMonitor` aspect to match any invocations of an `actionPerformed` method within the scope of the application:

```
pointcut userEvent(EventObject e, EventListener l):
   execution (void *.*(*)) && args(e) && target(l);
```

Both `EventListener` and the `EventObject` are exposed to the body of the advice applied, which incorporates relevant information about the event (source object, time, action performed, etc.) to the user model. Hartman and Bass [9] provide a detailed discussion about this approach to capturing interaction between user and applications.

Adaptability Aspects are notified by the context manager whenever a specific event occurs. This is achieved extending an implementation of the *Observer* pattern described in [8]. Specifically, whenever an event is matched by monitor aspects, the `EventDescriptor` in the `ContextEventVerifier` class is updated by the aspect. Then, the following `AdaptationProtocol` updates all the observing aspects, applying adaptation if the adaptation model determines that the produced event requires adaptation.

```
1:  public aspect AdaptationProtocol extends ObserverProtocol{
2:     declare parents: ContextEventVerifier implements Subject;
3:     declare parents: AdaptivityAspect implements Observer;
4:     protected pointcut subjectChange(Subject s):
5:     call (call ContextEventVerifier.setEventInfo(..)) && target(s);
6:  protected void updateObserver(Subject s, Observer o) {
7:     ContextEventVerifier cev = (ContextEventVerifier) s;
8:    ((AdaptivityAspect)o).applyAdaptation(
8:           am.getRules(cev.getEventDescriptor());
9:   }
10: }
```

The **Adaptation Data Provider** produces appropriate correspondences between user action and UI adaptation. This module enables the system to appropriately process the information which has been previously acquired. In order to represent UI components, the framework uses UI-specific ontologies, and domain-specific ontologies for representing the application's domain. Semantic information present in components is used to establish relations between UI and domain-specific ontologies. For the task model, we integrate a task ontology based on ConcurTaskTrees [16], which is a hierarchical notation which allows the specification of (sub)tasks or nodes that need to be performed to successfully complete a task.

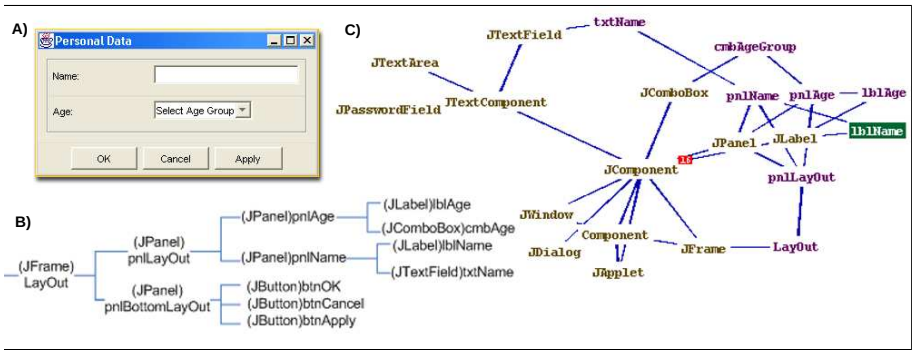

Fig. 3. A) Sample application dialog.**B)** Component hierarchy.**C)** Lattice representation of the UI Model ontology extended with the component hierarchy from the dialog.

The generation of an Adaptation Model involves significant decision-making capability. Hence, the Reasoner must support the writing of rules specifying the decisions that need to be made. Although these decisions may be sometimes relatively simple, truly adaptive behaviour implies that most of the time adaptation rules are likely to change over time. For this reason the prototype uses a rules engine, where a set of rules can be repeatedly applied to the collection of facts available in the different model ontologies. Rules that apply are executed, modifying the Adaptation Model accordingly. Specifically, the framework uses Jess [5], which is very convenient since it allows direct creation and manipulation of Java objects.

5 Related Work

Supporting GUI adaptation based on AOP is not a new idea. Sendín et al. take in [21] a non-intrusive approach to the problem of GUI plasticity, but depending exclusively on environmental context information. Hence, the UI is adapted depending on device characteristics, location, etc., but it is not responsive to user interaction.

Proposals such as the Context-Broker Architecture (CoBrA) or the Context Toolkit [3, 19] relieve developers from building specific adaptive mechanisms, letting them focus on adaptive behaviour. However, these proposals do not explicitly deal with the particularities of user interfaces, and lack the transparency of the AOP approach since applications have to comply with the architecture's specifications.

To our knowledge, there is no proposal available focused on enabling adaptivity in already existing applications based on user-computer interaction context. Furthermore, ours is a non intrusive approach that enables the use of an additive plug-in structure to reuse general patterns of adaptive behaviour.

6 Conclusions

This work advocates for an aspect-based approach to enable adaptive behaviour on already existing, GUI-based applications. We have presented a framework which allows to overcome the different problems described in section 2: **(i)Poor reusability and (ii) Lack of transparency.** The framework's architecture enables reusability and transparency, providing an explicit and non-invasive way of altering and extending the UI. The use of the presented framework permits to apply general adaptation patterns to different facets of regular applications in a transparent way (*i.e.,*the application does not need to be specifically prepared for adaptation and will still benefit from adaptive behaviour not specifically designed for it).**(iii)User model fragmentation and redundancy.** The framework is able to collect user information unobtrusively and in a centralized manner. The use of a global user model accessible to all applications through a *generic user modeling server* [13] enables adaptive applications to cooperatively retrieve and use both implicit and explicit (*i.e.,* preferences) information from the user. This ensures data consistency, and a fast growth of the amount of information obtained from users. As a result, learning and inference based on that information is more accurate and efficient, since applications have better user models available in shorter periods of time. User information is precious to AUIs, to the point that learning algorithms are specifically tailored to work with very restricted sets of information [14]. **(iv)Lack of coordination.** The ontology model supported by the Adaptation Data Provider enables coordinated adaptation since cross-inference and learning can be performed on different application UIs. Domain and Task Models can be developed to comprise several applications, bridging tasks across different UIs.

Currently, our framework prototype is being extended in order to apply it to real-world examples of adaptive systems. We intend to validate our proposal implementing systems which have already been described in the literature [6, 10] with our approach to test its applicability. In this sense, we have a special interest in applying it to *end-user/Programming by Demonstration* (PBD) development systems [15, 1], a field in which our approach can realise its full potential.

References

1. M. M. Burnett, C. R. Cook, and G. Rothermel. End-user software engineering. *Commun. ACM*, 47(9), 2004.
2. G. Chen and D. Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Dartmouth College, 2000.
3. H. Chen, T. Finin, and A. Joshi. An intelligent broker for context-aware systems.
4. A. Dantas and P. Borba. Adaptability aspects: An architectural pattern for structuring adaptive applications with aspects. In *Proc. of SugarLoafPLoP'2003*.
5. E. Friedman-Hill. *Jess in Action*. Manning Publications, 2003.
6. K. Z. Gajos, M. Czerwinski, D. S. Tan, and D. S. Weld. Exploring the design space for adaptive graphical user interfaces. In *Proc. of AVI'06*.
7. P. Gray and D. Salber. Modelling and using sensed context information in the design of interactive applications. *LNCS*, 2254, 2001.
8. J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. of OOPSLA'02*.
9. G. S. Hartman and L. Bass. Logging events crossing architectural boundaries. In *Proc. of INTERACT'05*, volume 3585 of *LNCS*, 2005.
10. L. A. Hermens and J. C. Schlimmer. A machine-learning apprentice for the completion of repetitive forms. *IEEE Expert*, 9(1), 1994.
11. A. Jameson. *The Human-Computer Interaction Handbook*, chapter Adaptive Interfaces and Agents. Lawrence Erlbaum Associates, 2003.
12. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. of ECOOP'01*, volume 2072 of *LNCS*, 2001.
13. A. Kobsa. *The Adaptive Web: Methods and Strategies of Web Personalization*, chapter Generic User Modeling Systems. Springer, 2007.
14. P. Langley. Machine learning for adaptive user interfaces. *LNCS*, 1303, 1997.
15. A. I. Mørch, G. Stevens, M. Won, M. Klann, Y. Dittrich, and V. Wulf. Component-based technologies for end-user development. *Commun. ACM*, 47(9), 2004.
16. F. Paterno, C. Mancini, and S. Meniconi. Concurtasktrees: a diagrammatic notation for specifying task models. In *Proc. of INTERACT'97*.
17. F. R.E. and D. Friedman. *Aspect-Oriented Software Development*, chapter Aspect-Oriented Programming is Quantification and Obliviousness. Addison-Wesley, 2004.
18. E. Ross. Intelligent user interfaces: Survey and research directions. Technical Report CSTR-00-004, University of Bristol, 2000.
19. D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proc. of CHI'99*, 1999.
20. R. Segal and J. O. Kephart. Mailcat: An intelligent assistant for organizing E-mail. In *Proc. of Agents'99*, 1999.
21. M. Sendín, J. Lorés, F. Montero, and V. López-Jaquero. Towards a framework to develop plastic user interfaces. In *Proc. of Mobile HCI'03*, volume 2795 of *LNCS*, 2003.