Available online at www.sciencedirect.com
ScienceDirect
Journal of Visual Languages and Computing 19 (2008) 695–720
Journal of Visual Languages & Computing
www.elsevier.com/locate/jvlc

An extension of UML for the modeling of WIMP user interfaces

Jesús M. Almendros-Jiménez*, Luis Iribarne

Information Systems Group, University of Almería, Spain

Received 16 December 2005; received in revised form 7 September 2007; accepted 13 December 2007

Abstract

The *Unified Modeling Language* (UML) [OMG, Unified Modeling Language Specification, Version 2.0, Technical Report, Object Management Group ⟨http://www.omg.org/technology/documents/formal/uml.htm⟩, 2005] provides system architects working on analysis and design (A&D) with one consistent language for *specifying*, *visualizing*, *constructing*, and *documenting* the artifacts of software systems, as well as for the business modeling. The *user interface* (UI), as a significant part of most applications, should be modeled using UML, and automatic CASE tools may help to generate UIs from UML designs. In this paper, we describe how to *use* and *specialize* UML diagrams in order to describe the UIs of a software system based on WIMP (*Windows*, *Icons*, *Menus and Pointers*). *Use case diagrams* are used for extracting the *main UIs*. Use cases are described by means of *user-interaction diagrams*, a special kind of *activity diagrams* in which states represent *data output actions* and transitions represent *data input events*. Input and output interactions in the *user-interaction diagrams* help the designer to extract the UI components used in each UI. We obtain a new and *specialized version of the use case diagram* for the UI modeling (called *UI diagram*) and a class diagram for UI components—called *UI-class diagram*. The *user-interaction*, *UI* and *UI-class diagrams*, can be seen as the *UML-based UI models* of the system. Finally, *UI prototypes* can be generated from *UI-class diagrams* with CASE tool support. As case study of our technique, we will describe an *Internet book shopping system*.
© 2008 Elsevier Ltd. All rights reserved.

Keywords: UML 2.0; Use cases; Model-driven development; User interface modeling; Human–computer interaction

1. Introduction

The *user interface* (*UI*) *design* has been explored in the communities of *Human–Computer Interaction* (*HCI*) and *Software Engineering* (*SE*): the software should carry out those tasks for which it is conceived and the UI should be friendly and usable enough for users [2–4]. UI design is a software development task which has been deeply studied from the early years of computer software development [5,6]. Due to the increasingly complexity applications, the UI design has become a more difficult task in which many artifacts and software components are required. Many proposals of *software architectures* (for instance, Seeheim model [7], MVC [8,9], PAC [10,11], ARCH [12] and PAC Amodeus [13] among others) propose guidelines about how the UI design and the implementation should be carried out. UI design aspects focus not only of the *behavioral* parts of the UI but also on the *layout* part [14–16]. In addition, UI design methods involve user participation during the information

*Corresponding author. Tel.: +34 950015687.
E-mail addresses: jalmen@ual.es (J.M. Almendros-Jiménez), liribarne@ual.es (L. Iribarne).

1045-926X/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2007.12.004

gathering phases, UI design and UI evaluation. All these aspects are called *user centered design methods* [17,18].

The handling of more complex applications has lead to the development of tools for UI design [5,6], in which the system designer works with *high-level and abstract models*, in particular visual modeling of UIs. The level of abstraction on UI design allows the UIs to be adapted and generated for several platforms. The design of UIs is now a development process in which a high-level specification and modeling of the UI play a crucial role.

The *model-driven development* (MDD) of UIs [19,20] consists of the specification of the UIs through declarative and visual models that describe multiple perspectives and artifacts involved in the development of the UI. The MDD is based on visual tools in which the system designer uses a graphical notation to depict multiple models from different perspectives. Code generators of such tools generate a code according to the specified model which allows the changes in the specification to be easily mapped into the code. The adoption of the MDD approach has the following advantages [20]:

- More abstract descriptions of UIs than traditional approaches.
- A systematic design and description of UIs:
  - UI design from different levels of abstraction.
  - Increasing refinement of models.
  - Reuse of specifications of user models.
- Automatizing of design and implementation of UIs.

However, there are some troubles to be solved:

- Models are sometimes hard to learn [3]: notation can be complex and the semantics may not be well-known. Visual design tools can help to overcome both problems.
- The integration of the functional requirements and the UI is still a task to be solved.
- There is no common agreement about what models should be used for modeling the UI.

Despite there is no common agreement about what models should be used in a MDD process for UIs, there are common models used in most approaches in the literature: *task*, *domain*, *user*, *dialogue* and *presentation models* (see Table 1).

Table 1
Models used in an MDD process for user interfaces

Model	Description
Task model	It specifies the tasks that the user will carry out on the user interface
Domain model	It describes the domain objects interacting in each task detected in the task model
User model	It describes the user requirements
Dialogue model	It describes the communication between the user and the user interface
Presentation model	It describes the user interface

Task models specify what tasks the user will carry out on the UI. Tasks are decomposed in subtasks in order to describe the steps to accomplish a given task. Task models very often include functional requirements and UI tasks. They also include non-functional requirements like time response. Some of the most well-known approaches for task models are textual methods like *GOMS* (*Goals*, *Operators*, *Methods*, *Selection rules*) [21,22] and formal methods like *ConcurTaskTrees* (CTT) [19,23].

The *domain model* describes the objects that interact in each task detected in the task model. Although some approaches use the *entity/relationship diagram* [24–27], most of them use *class diagrams* [28–37]. Objects can be classified [38] into *interaction objects* (i.e. UI components), and *domain/application objects* (i.e. non-UI objects) representing *databases* or *external devices*.

The *user model* describes user's requirements: preferences, information, devices owned, context, etc. Each user is identified with a *role*. The purpose of the user model is to provide an appropriate UI for the user requirements. The user model is used to personalize and/or adapt the interface for the user.

The *dialogue model* describes the communication between the user and the UI. Basically, it describes how the user introduces *input data*, how the user interacts with the UI, and how the UI shows *output data*. There are some approaches that use diagrams like *Petri nets* [32], *statecharts* [33,38,39] or *activity diagrams* [25–27,29,40]. In modern user interfaces like those based on *hypertext* with *dynamic content*, the dialogue model is replaced by the *navigational model* where the user can navigate through different web (*on-the-fly generated*) pages following *hyperlinks*. In most cases *statecharts* are used for modeling web navigation [41,42].

The *presentation model* represents the UI. It describes the UI components and its layout. There are some approaches [25–27,30,38,43] that offer both an *abstract* model and a *concrete* model. The abstract model describes abstract objects of interaction and the concrete model describes concrete objects for different platforms.

The *Unified Modeling Language* (*UML*) [1] provides system architects working on analysis and design (A&D) with one well-consistent language for *specifying*, *visualizing*, *constructing*, and *documenting* the artifacts of software systems, as well as for business modeling. The UI, as a significant part of most applications, should be modeled using UML [44], and automatic *CASE tools* may help to generate UIs from UML designs.

In this paper we will present how to *use* and *specialize* UML diagrams in order to describe the UI of a software system based on WIMP (Windows, Icons, Mouse, Pointers) interfaces. We will follow a particular MDD perspective. Such perspective has the following features:

(a) We use a UML *use case diagram* to describe the *main UIs*.
(b) We describe each use case by means of a special kind of *activity diagrams*, called *user-interaction diagrams*, in which states represent *data output actions* and transitions represent *data input events*. This perspective allows the designer to model the dialogue (i.e. input–output interaction) in each main UI. In addition, some states of the *user-interaction diagrams* correspond with small pieces of interaction described separately in other user interaction sub-diagrams. This provides a natural decomposition of the main UIs into *secondary UIs* for specific tasks.
(c) A new and *specialized version of the use case diagram* representing the UI model is obtained: the *UI diagram*. This *user-interface diagram* describes the main and secondary windows used for each defined task. These windows are obtained from the set of *user-interaction diagrams* built in the previous step. In other words, the description of use cases by means of *user-interaction diagrams* leads to a natural decomposition on pieces of user interaction. Now, the system designer can consider new windows for some pieces of interaction. These new windows are included in the *UI diagram* as use cases.
(d) Each input and output interaction of the *user-interaction diagrams* allows the system designer to extract the *UI components* (i.e. widgets) used in each UI. A *UI-class* diagram is obtained from the set of user-interaction diagrams containing the set of UI classes needed for the user interaction. The *user-interaction*, *UI* and *UI-class* diagrams can be considered as the *UML-based UI models* of the system.
(e) Finally, *UI prototypes* can be generated from the *UI-class diagram*. A CASE tool can help to determine the layout of each UI component in each window. There is a correspondence between the UI prototypes and the *UI diagram*. Each use case of the *UI diagram* defines a window in the UI.

In other words (see also Table 2):

- We consider the *use case diagram* as a *starting point* of the UI model. The use case diagram specifies the *main windows or tasks* that each user or group of users can carry out.
- We define a new kind of diagram, *UI diagram*: a specialized version of the use case diagram as a high-level description of the UI. This view represents the *task model*. The *UI diagram* describes the *main and secondary tasks* by means of use cases. In addition, each task/use case has its own presentation unit (i.e. window).
- In addition, we integrate this system view with a set of *specialized activity diagrams* called *user-interaction diagrams*, which define the *dialogue model*. Each use case of the *UI diagram* is described by means of a user-interaction diagram. In other words, each element of the task

Table 2
Correspondence between models and UML diagram views

Step	Model	View	UML diagram
#1	Dialogue	User-interaction diagrams	Specialized activity diagrams
#2	Task	User-interface diagrams	Specialized use case diagrams
#3	Presentation	User-interface class diagrams	Class diagrams
#4	Presentation	User-interface prototypes	Class diagrams

model is represented by means of an element of the dialogue model.

- In addition, a *UI-class diagram* is generated from the set of *user-interaction* and *UI diagrams*, representing the *presentation model*. This class diagram describes the *UI components* (i.e. windows and widgets) of the UI.
- Finally, *UI prototypes* can be generated from UI-class diagrams with a CASE tool support. This permits a preliminary and quick system user's view to be analyzed. This view also describes the *presentation model* of the modeled UI.

This perspective follows an MDD-based technique in which diagrams are integrated, providing multiple views of the developed system (i.e. the UI).

One of the main contributions of our work on UI modeling is the introduction of two concepts: *inclusion and generalization*. When a use case is described by means of a user-interaction diagram, some pieces of the interaction are separately described. This is the case of inclusion. Generalization is a more complex relation. When use cases are described by means of a *user-interaction diagrams*, there are some use cases that represent similar interactions. In other words, since the dialogue between the user and the UI follows the same logic, the UI components are similar (i.e. they follow the same *pattern*).

The UML use case diagram has two mechanisms: *inclusion* and *generalization/specialization* between use cases. In our *UI diagram*, we will use these kinds of mechanisms to relate use cases/tasks. A use case is included in another use case when the interaction is included. On the other hand, a use case generalizes another use case when most general one has *at least the same interactions/UI components* as the particular use case, and the particular one can *replace/rewrite* some of them by more particular ones. In other words, use cases are compared following criteria of *reuse* and *replacement*, representing *inheritance* between use cases.

Finally, our paper is also concerned with a *formal description of our model-driven development technique*. Such a formalization of our technique can be shown through two main purposes:

- A MDD technique that involves multiple perspectives can lead to *inconsistent models*. Models should be integrated and consistent. Our formal definition expresses when the UI and user interaction diagrams are consistent or well-formed. Such consistence proving can be viewed as a *model checking technique*.
- Our technique goes towards the *automatic generation of code* from UML models. Therefore, a formal translation of *user-interaction diagrams* and *UI diagrams* into UIs is defined (in particular, this translation can be used to generate UI-class diagrams).

The main limitations of the approach are two.

Firstly, we *restrict our technique to WIMP interfaces*. We believe that our technique could be extended to *Post-WIMP* interfaces based on *hypertext* with *dynamic content* and also extended to the handling of *adaptation* and *multi-modality*. Dialogue models for *Web* content have been studied in some previous works [41,42,45–47]. Our intention is to extend our work in the future by following the quoted proposals.

Secondly, our proposed technique is only a piece of a complete *method* (i.e. set of processes, notations for steps, etc.) for UI design. In particular, the *layout part* of the UIs is not covered by our technique and therefore the technique does not properly contribute to a complete *UI design method*. *Ergonomic criteria* [2,14–16,48,49] are still crucial for the *usability* of the UI. The main goal of the proposed technique is the dialogue and UI modeling in the framework of UML. In addition, the proposed technique is useful for rapid prototyping of UIs in modeling phases. However, such preliminary design should be just a prototype, and we recommend the system designer to improve and check the prototypes by using *user center design and evaluation methods* [17,18].

1.1. Organization of the paper

The rest of this paper is organized as follows. Section 2 overviews the related word. Section 3 describes our MDD for UI modeling. Section 4 shows an Internet book shopping (IBS) example that illustrates our technique. Section 5 formalizes the technique. Finally, Section 6 presents conclusions and discusses future work.

2. Related work

There are many works in the literature dealing with the problem of model-driven development of UIs. On the one hand, there are two main approaches: those *extending or partially using*

UML diagrams, and those *non-UML* proposals. On the other hand, most proposals model *WIMP* interfaces. However, there are some recent proposals [41,42,45] concerned with the design of modern UIs (Post-WIMP interfaces) based on *hypertext* with *dynamic content*: *client and server side applications*, and *on-the-fly page generation*. Finally, there is some recent interest [45,50–53] in *XML-based representation and manipulation* of *interaction data*, that is, the data that define and relate all the relevant elements of a UI.

With respect to MDD-based techniques of UIs, there are many relevant proposals with CASE tool support (see [20] for a survey). Some of them are *non-UML* proposals and others are *hybrid* or *extensions* of UML. The most relevant and related with our approach are TRIDENT, TERESA, WISDOM, UMLi and IDEAS (see Table 3).

TRIDENT (*Tools foR an Interactive Development ENvironmenT*) [25–27] is a proposal for an environment of the development of highly interactive applications. UIs are generated in an automatic or quasi-automatic way. The task model of TRIDENT uses the *entity/relationship model* together with the so-called *activity chaining graphs*. The presentation model is based on *abstract* and *concrete* interaction objects, presentation units and logical windows. There is a CASE tool called SEGUIA (*System Export for Generating a User Interface Automatically*) [49] for generating the UI from TRIDENT models. This approach can be considered as a non-UML proposal. However, most of the ideas studied in TRIDENT were later included in UML proposals.

TERESA (*Transformation Environment for InteRactivE System RepresentAtions*) [53] is a tool designed for the generation of UIs for multiple platforms from task models designed with *CTT*. CTT [19,23] is a tool based on the formal language *LOTOS* for analysis and generation of UIs from task models. CTT is a formal and visual language for task specification in which *temporal* and *concurrency* relationships can be specified. There are four kinds of tasks in CTT: user tasks, application tasks, interaction tasks and abstract tasks (which are decomposed in concrete tasks). CTT permits the description of complex UIs in which multiple users and concurrent tasks can be carried out. Although CTT is a powerful tool, we have not adopted it in our framework CTT for task modeling. Our framework is focused on simpler interfaces based on WIMPs, and on an MDD technique based on UML diagrams. Our *UI diagram* has some similarities with the CTT tool. Through this specialization of the use case diagram, we might decompose tasks into subtasks which are also described by means of *user-interaction diagram*s to describe the dialogue model. We also make a clear distinction between user tasks (the use case and UI models), interaction tasks (the *user-interaction diagram*) and application tasks such as database handling in a different way from UI tasks.

Partially based on UML, WISDOM (*Whitewater Interactive System Development with Object Models*) [29,31,34,35] is a proposal for UI modeling. WISDOM uses UML but the authors have also adopted the CTT. The domain model captures the objects and the business model describes each user task and

Table 3
User interface model-driven development

Name	Type	Diagrams	Models	Technology	References
TRIDENT	Non-UML	Entity relationship diagrams and activity chaining graphs	Task and presentation models and UI prototypes	WIMP	[25–27,49]
TERESA	Non-UML	CTT	Task models and UI prototypes	WIMP/Web	[19,23,53]
WISDOM	Hybrid	Use case diagram, class diagram, activity diagrams and CTT	Business, domain, dialogue and presentation models and UI prototypes	WIMP/Web	[29,31,34,35]
UMLi	UML	Use case diagram, class diagram and activity diagrams	User, task and application models	WIMP	[30,38,54,55]
IDEAS	Hybrid	Sequence diagrams, class diagram and Statecharts	Task, presentation, domain and dialogue models and UI prototypes	WIMP	[33,56,57]
Our proposal	UML	Use case diagram, activity diagrams and class diagram	Task, dialogue and presentation models and UI prototypes	WIMP	

entities involved in such tasks. The business model in WISDOM is represented by the UML use case model and the domain model by a stereotyped class diagram. Use case model is completed by activity diagrams in order to describe the services that the system offers its users. The design model offers a gap between the application level and the UI. The dialogue model is specified by means of CTT. In WISDOM there are also interaction and presentation models. The presentation model describes the UI with stereotyped constructors to describe the structure and navigation through the objects. We agree with WISDOM in the use of the use case diagram as a starting point of the UI design. In addition, a class diagram is also used in our approach for describing objects interacting with the user in the UI. However, in our approach the task modeling is carried out with a specialized version of the use case diagram and completed by activity diagrams for dialogue modeling instead of CTT.

Fully based on UML, UMLi [30,38,54] proposes an extension of UML for UI modeling. UMLi aims to preserve the semantics of existing UML constructors since its notation is built by using UML extension mechanisms. Like our technique, the scope of UML is restricted to WIMP interfaces. It distinguishes between application and interaction objects, as we do. In fact, our modeling technique is focused on interaction objects. In our technique, application objects are supposed to be modeled by separate class diagrams in later steps. The user model is represented in UMLi by the use case model, the task model by activity diagrams, and application model by class diagrams. There is a distinction between concrete and abstract presentation models. The technique is concerned with UI specification rather than generation of the UI. UMLi distinguishes between *freecontainers* and *containers* as presentation abstract elements, *inputters* and *displayers* for input and output data, *editors* for input/output and *actioninvokers* representing event handlers. UMLi is supported by ARGOi [55]. This work is very similar to ours in the adoption of UML diagrams for UI modeling. However, the extensions are different. In our case, we focus more on the use case diagram and its specialization as task model, and we introduce and handle new concepts like inclusion and generalization between use cases.

Finally, IDEAS (Interface Development Environment within OASIS) [33,56] is also a UML-based technique for the specification of UIs in UML and the OASIS (Open and Active Specification of Information Systems) specification language [57]. The task model is based on sequence diagrams. The domain model is the class diagram. The dialogue model uses statecharts: transitions can model several windows (dialogue interaction diagrams) and internal transitions in each window. IDEAS can also specify an abstract UI with components. Therefore, this technique might be considered as hybrid, through there are some similarities with ours. The main similarity can be found in the dialogue model in which several windows can be described.

3. MDD for UI modeling

In this section we will summarize the kinds of models that we propose in our MDD-based technique. As we advanced in the introduction, our MDD-based technique is based on the use of specialized use case, activity and class diagrams.

Our MDD-based technique is inspired by a spiral methodology of at least three iterations. Each iteration involves the building of use case, *UI*, *user-interaction* and *UI-class diagrams*, except for the first iteration which only involves the description of the use case diagram and *user-interaction diagrams*. In the second iteration, the system designer reviews the original use case diagram to produce the *UI diagram*. This refinement is achieved by identifying the relationships between the use cases in the *user-interaction diagrams*. In the third iteration, the technique includes the generation of UI-class diagrams and UI prototypes as a new step. Each step of the methodology is supposed to be increasingly applied in such a way that models can be increasingly refined. Fig. 1 depicts the technique.

3.1. Use case diagram

Use case diagrams are used as a starting point for UI design. Use cases are a way of specifying required usages of a system. Typically, they are used to capture the requirements of a system, that is, what a system is supposed to do.

A use case diagram consists of a set of *actors* (*users* and *external systems*) and *use cases*. The relationships between actors and use cases are called *associations*. They represent the set of tasks that the user carries out in the system. Relationships between actors are *generalizations/specializations*.

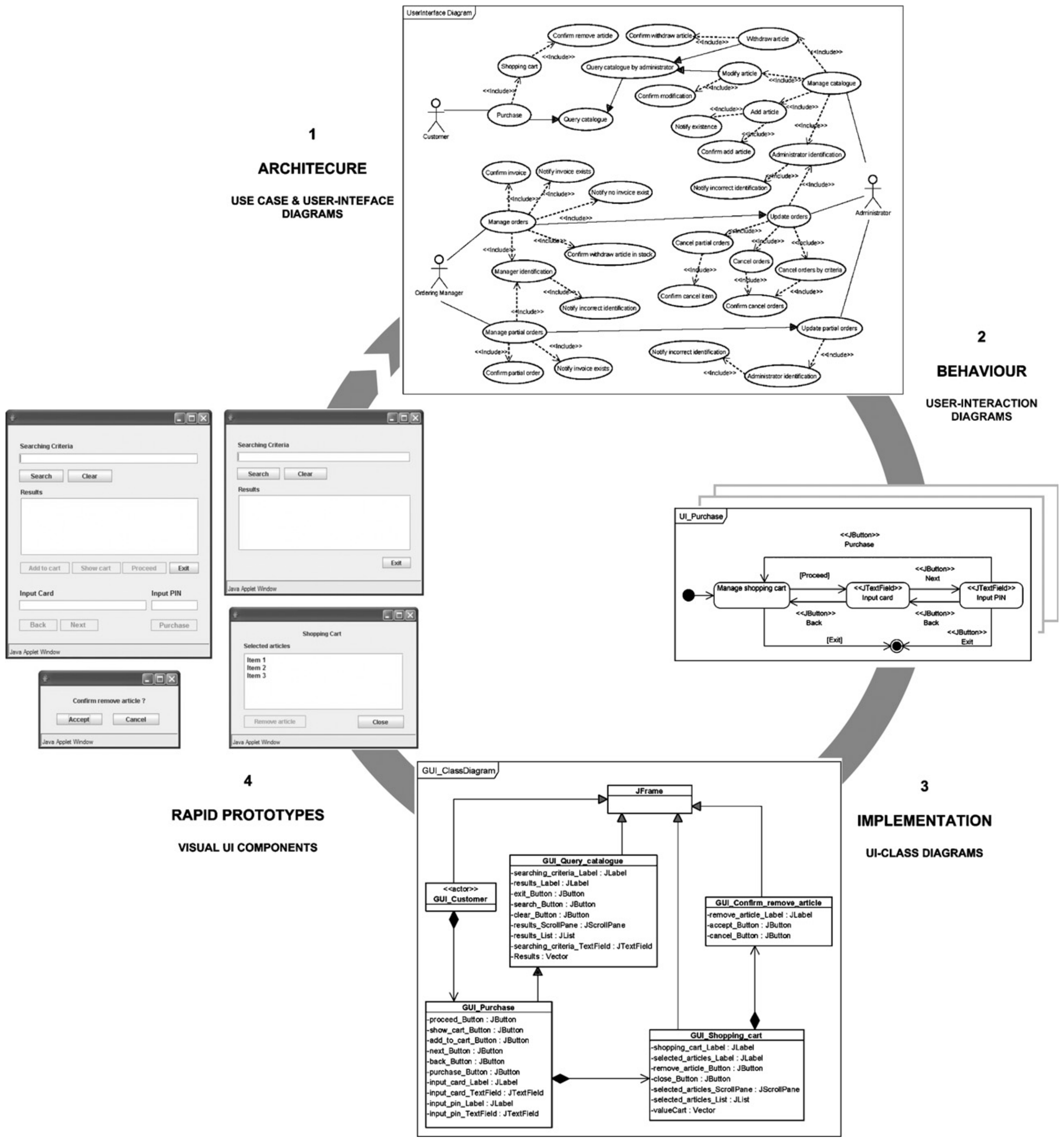

Fig. 1. Steps to apply for the proposed technique.

An actor *p* is more general than an actor *q* whenever *q* can interact with the system as *p* and, additionally, it can interact in more cases.

In order to design a prototype of the UI, the use case diagram should include the system actors, and for each actor it should include the set of (main) tasks in which the actor participates together with generalization/specialization relationships between actors. *Include* and *generalization/specialization* relationships between use case are omitted in the use case diagram since they will be described in the *UI diagram*.

From the point of view of UI modeling, the use case diagram can be considered as a *high-level* *description* of the system *main windows* of each user or group of users together with external systems providing services in each task.

3.2. User-interaction diagrams

The second modeling technique that we use in our framework is the *activity diagram*. In UML, use cases can be described through activity diagrams which specify the interactions between the actor and the system and the changes of state of the system and the communications with their environment. Activity diagram can also include possible variations of the basic behavior, including exceptional behavior and error handling. However, we need to *specialize the activity diagrams* for UI modeling in the following sense.

Our activity diagrams include *states* and *transitions*. The states represent *data output actions/data requesting*, that is, how the system responds to user interactions showing data (or requesting them). Transitions are used to specify when the user can introduce data or interact with the system, and the corresponding *event* is handled. Transitions can be *conditioned*, that is, the event handling is controlled by means of a *boolean condition*, which can be referred to *data/business logic* or a *previous user interaction*. In other words, it is possible more than one transition from a state, and which of them will run will depend on data/business logic or the previous user choices. We call *user-interaction diagrams* to this kind of activity diagrams used for dialogue modeling.

Therefore, *user-interaction diagrams* are graphs linking *states* by means of *transitions*, which are arrows connecting an *initial state* and a *final state*. Initial (resp. final) state is the starting (resp. end) point of the diagram. Each state represents data output (and input request) and transitions represent actor interactions (user's events). Transitions are labeled by means of *conditions/events*, representing the boolean conditions to be held and the events to be achieved for the state change. There can be *diamonds* between transitions that describe alternative paths, depending on a *boolean condition*.

Once the actors and the use cases associated with each actor are specified, the system designer should provide, in a second step, a set of *user-interaction diagrams* to describe each use case in the use case diagram.

However, from a practical point of view, it is convenient to use more than one *user-interaction diagram* for describing a use case. That is because the logic of a use case is usually too complex. A *user-interaction diagram* can be deployed in several *user-interaction diagrams*, in which a part of the main logic is separately described. Therefore, user interaction diagrams can include states which do not correspond to data output, rather than representing *sub-diagrams*. In this case, the states are called *non-terminal states*; otherwise, they are called *terminal states*.

Now, it is sometimes desirable to be able to combine the logic of the sub-diagrams and the main logic. For this reason, we will use in the main user-interaction diagram transitions with conditions which can be referred to the logic of the sub-diagrams.

3.3. UI diagrams

Once we have obtained the use case diagram together with a set of *user-interaction diagrams*, some of the *user-interaction diagrams* will correspond to use cases and others to states of use cases. Now, it could be useful to have a *new version of the use case diagram*, in which one could know what are the *main user-interaction diagrams*, that is, which are the *user-interaction diagrams* corresponding to use cases, and which are the *secondary user-interaction diagrams*, definitively, which are the states of use cases. For this reason, we will build a new version of the use case diagram, called *UI diagram* as follows.

The *UI diagram* contains the same actors and use cases of the use case diagram. In addition, it will add new use cases. The *new use cases* are obtained from the *non-terminal states* of the *user-interaction diagrams*. The system designer can decide *to include all the non-terminal states or not* depending on the number of windows that (s)he wishes.

In other words, the *UI diagram* is supposed to contain *the set of windows of the system*. Therefore, the system designer selects those that will have *its own window* from the set of non-terminal states.

Those non-terminal states not included in the *UI diagram* are supposed to be described separately because the logic was too complex to be described in a unique user-interaction diagram. However, they will share the same window as the main interaction.

In order to connect use cases, we will use the *include* or *generalization* use case relations. Inclusion and generalization use case relationships can be detected by comparing *user-interaction diagrams*.

3.4. Inclusion and generalization relationships

Firstly, the *user-interaction diagrams* can be compared by means of an *inclusion relationship*. This is the case when a user-interaction diagram contains as one of the states another user-interaction diagram. However, inclusion means that *the logic of the included state is not modified*, that is, neither states nor transitions of the main user-interaction diagram refers to the included one. This typically occurs when the main use case defines its own logic using the included use case as a piece of behavior, but *it does not "access" to the logic of the included use case*. The inclusion relationship can be considered as a *similarity relationship* between use cases.

Secondly, the *user-interaction diagrams* can be compared by means of *generalization/specialization relationships*. This is a more complex case that corresponds with the one in which a use case/user interaction *modifies the logic of another use case/user interaction* by adding new states or transitions, or *changes some states or transitions of another use case/user interaction*. Basically, both use cases are *similar*; they share some elements that could be *replaced/rewritten*; in other words, these use cases follow the same *pattern*, and one of them can be *reused* to describe/implement another.

Here, we have to assume that terminal states and transitions of *user-interaction diagrams* can be compared by means of a (partial and reflexive) *replacement relationship* $\sqsubseteq$. That is, (terminal) states s, s' can be compared with a relationship $s \sqsubseteq s'$, or two transitions λ, λ' can be compared with a relationship $\lambda \sqsubseteq \lambda'$ in such a way that s' can be replaced by s or λ' can be replaced by λ, respectively. The replacement relationship can express similar semantics (appearance or behavior). The replacement relationship is decided by the system designer. It typically refers to data output actions and input events which are similar from the appearance or behavior point of view. For instance, a user selection from a list of two columns can be replaced by means of a user selection from a list of three columns, or a click in a button called "accept" can be replaced by a click in a button called "confirm". Finally, conditions can be (for instance) replaced whether one of them is *more restrictive* than the other.

The replacement relationship induces a *replacement relationship on user-interaction diagrams*. For instance, two *user-interaction diagrams* that describe the dialogue of the user with UI components in which the user selects from the lists and accepts/confirms are similar and therefore each one can be replaced by each other. A user-interaction diagram a' can be replaced by a if the states and transitions of a' can be replaced by the states and transitions of a. Obviously, it also induces a *replacement relationship* between non-terminal states.

The replacement relationship defines a *similarity relationship* where some elements are replaced following the replacement relationship and new elements are added. This similarity relationship is represented by the *generalization/specialization relationship* between use cases.

The *generalization/specialization relationship* between *user-interaction diagrams* captures two cases:

- When the logic of an included use case is modified by adding states and transitions in the main user-interaction diagram referred to the included user-interaction diagram.
- When the user-interaction diagram of a non-terminal state of a user-interaction diagram can be replaced by means of another user-interaction diagram.

In practice, *both cases of generalization/specialization can be combined*, that is, a user-interaction diagram a generalizes a user-interaction diagram a' whenever there exists a state a'' of a' such that a generalizes (in some of the cases above) a''.

In other words, the generalization/specialization of use cases allows to build new use cases with a more complex logic containing the specialized use case, by adding transitions and states or modifying the existent ones. However, the inclusion allows to build new use cases with a more complex logic but without adding or modifying the states and transitions of the included use case.

Obviously, some *user-interaction diagrams* may not be compared through replacement and inclusion relations, that is, they do not describe "similar" activities. Nevertheless, some *user-interaction diagrams* can be compared through a combination of generalization/specialization and inclusion. In this case, *intermediate user-interaction diagrams* might be defined by decomposing the *user-interaction diagrams* in several steps, in such a way that the original *user-interaction diagrams* can be compared through *a chain of relationships*.

3.5. UI-class diagram and UI prototypes

The next step of our model-driven technique consists of the building of a class diagram for UI components, the so-called *UI-class diagram*. In addition, from the UI-class diagram, rapid *UI prototypes* can be built with the support of a CASE tool in order to define the preliminary layout of the UI prototypes.

The *UI diagrams* obtained in the previous state give us the main windows. Each use case connected to an actor can be converted into a window. Whenever an actor is connected to more than one use case, it can be considered as a window which contains (i.e opens) each window of each use case. A possible solution would be to consider a menu in the actor window from which each window for each task could be opened, or in more concise solutions, avoiding pop-up windows, a frame-based window would handle more than one window.

In addition, in the *user-interaction diagrams* obtained from use cases, we have also described input and output components for data output and request, and user events. They give us the UI components for each window.

Regarding the use case relationships in the *UI diagram*, inclusion between use cases has been considered as separate windows for specific tasks. Therefore, we can use the same previous solutions: a frame-based window system or a new window for each included use case. For generalization/specialization relationships, windows for use cases can be reused by using the inheritance relationship. In an object-oriented language, class inheritance can be used for inheritance of the layout and behavior of class windows. The replacement relationship could be solved by rewriting UI components in subclasses, and the dialogue model could be modified in subclasses by method rewriting.

3.6. Concrete modeling with the WIMP Java swing package

Now we would like to give precise correspondence between our models and the *Java swing package*. We have chosen this package due to the wide acceptance of this technology (*applet*, *frames*, and *event-handlers*). This correspondence can be viewed as a *concrete modeling* of our *abstract modeling technique* explained in previous sections. It provides a natural implementation of some of the concepts presented. We believe that we could adapt our technique to other UI technologies with little effort. The main difference *new UML stereotypes* for both input and output components, and making similar mappings between use cases and other kinds of user's windows.

Although we assume that the reader knows the basic behavior of Java swing classes, we would like to remark some concepts used in our mapping. Firstly, let us consider two kinds of window interfaces: *applet* and *frame*. A frame can include in the window area UI components such as buttons, labels, text areas and so on, and it can invoke other frames from it. An applet can only contain UI components in the window area. Therefore, frames will be used for the building of more complex UIs where several tasks can be done. Some of these UI components could be visible/enabled and disabled according to the executed tasks. We use inheritance between frames (and applets) assuming that it implies inheritance of behavior but not necessarily of appearance (inheritance requires recomposing the layout of UI components). Table 4 summarizes the rules for the Java swing package.

4. A UI modeling example

To illustrate our MDD-based technique, in this section we will explain a simple IBS model. Considering this scenario, next sections will describe those steps that should be followed to develop a UI project by using our particular view of use case diagrams, *UI diagrams*, *user-interaction diagrams* and *UI-class diagrams*.

4.1. Use case diagram

In the IBS example, there are three actors: the customer, the ordering manager, and the administrator. A customer directly makes the purchases by the Internet, querying certain issues of the product in a catalogue of books before making the purchase. The manager deals (total or partially) with customer's orders. And finally, the system's administrator can manage the catalogue of books by adding and eliminating books in the catalogue or modifying those already existing. The administrator can also update or cancel certain characteristics of an order or those orders fulfilling certain searching criteria.

As first step, we identify the main task to be carried out by each user that correspond with the

Table 4
Rules for a Java UI design

Rule	Section	Description
r1.	Actors	Each *actor* representing a user in the *user-interface diagram* is an *applet* or *frame*. When only one use case is connected to an actor, and the use case has no inclusions, the actor and the use case can be represented by means of the same applet. Otherwise, the actor is a frame
r2.	Use cases	Each *use case* of the *user-interface diagram* is a *frame*, except when is the only connected to an actor and has no inclusions; in the latter case, it is an applet. When it is a frame, it is invoked from the frame of the actor, or from the frame of the use case when it is included
r3.	Generalization and actors	The *generalization relationship* between two actors *p* and *q* (*p* generalizes *q*) corresponds with *inheritance* of the applet or frame represented by *q* from the applet or frame representing *p*
r4.	Generalization and use cases	The *generalization relationship* between two use cases *u* and *w* (*u* generalizes *w*) corresponds with *inheritance* of the applet or frame of *w* from the applet or frame of *u*
r5.	⟨⟨include⟩⟩	The *⟨⟨include⟩⟩ relationship* between two use cases *u* and *w* (*u* includes *w*) corresponds with the *invocation* from the frame of *u* of the frame of *w*
r6.	States	Each *state* of the *user-interaction diagram* necessarily falls into one of the following two categories: *terminal states* or *non-terminal states*. A terminal state is labeled with a *UML stereotype* representing a *Java output UI component* (stereotyped states). A non-terminal state is not labeled, and it is described by means of some *user-interaction diagram*
r7.	Transitions	Each *transition* in the *user-interaction diagrams* can be labeled by means of *conditions* or *UML stereotypes with conditions*. The UML stereotypes represent *Java input UI components*. The conditions represent *use choices or data/business logic*
r8.	Replacement relationship	The replacement relation is decided by the system designer. Basically, stereotyped states can be replaced if the *Java output UI component* can too. Similarly, stereotyped interactions can be replaced if the *Java input UI components* can too
r9.	States, non-terminal	The *user-interface diagram* can specify ⟨⟨*include*⟩⟩ or generalization relationships between the non-terminal state and the use case. We can follow these rules: (a) In the ⟨⟨*include*⟩⟩ relationship case, the non-terminal state is also a frame. It contains the Java UI components in the associated *user-interaction diagram*; (b) in the generalization relationship case, the non-terminal state is also an applet or frame containing the Java UI components in the associated *user-interaction diagram*, but the use case also contains these UI components
r10.	Conditions	The conditions of the transitions are not taken into account for the UI design

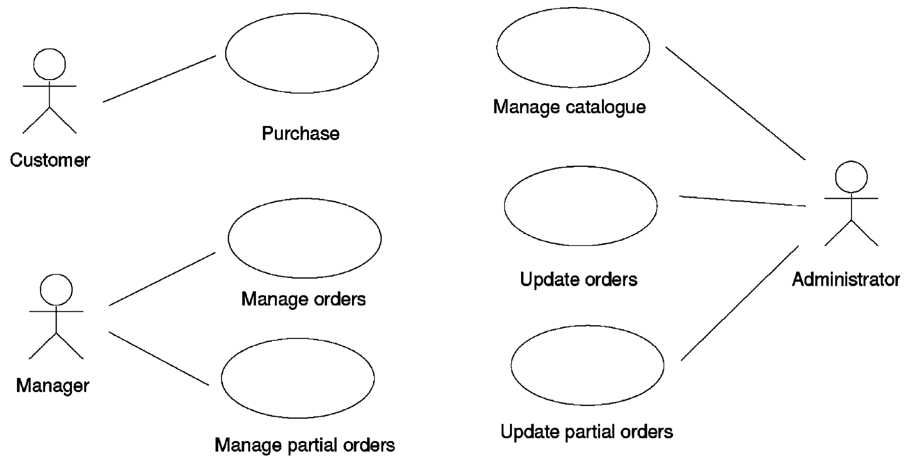

Fig. 2. Use case diagram of the IBS example.

main windows. This information is described with a use case diagram containing the identified actors of the system (see Fig. 2). In our case study, the actors are the `Customer`, the `Manager` and the `Administrator`, and the main tasks are `purchase`, `manage orders`, `manage partial orders`, `manage catalogue`, `update orders` and `update partial orders`.

4.2. *User-interaction diagrams*

Once the use case diagram has been described, the designer of the system specifies each use case by means of one or more *user-interaction diagrams*. As we have advanced before, each use case will correspond with a window. Therefore, *user-interaction diagrams* will describe the dialogue model of such window.

In our case study we have adopted a concrete modeling using the Java swing package following the guidelines defined in the previous section. In the case study, four Java UI components are used: `JTextField`, `JList`, `JLabel` and `JButton` for widgets and `frames` for windows.

UI components can be classified as input (a text area or a button) and output components (a label or list). Input/output components are associated with terminal states and transitions by using the appropriate stereotype. For instance, the stereotypes `JTextField`, `JList`, `JLabel` are associated with states and the stereotype `JButton` with transitions. Table 5 summarizes the features of the used UI components.

Fig. 3 shows the user-interaction diagram for the purchasing process together with a set of sub-diagrams. The main user-interaction diagram is `UI_Purchase`. The model shows how the customer begins the purchasing process by querying, adding or removing articles from the shopping cart. After a usual purchasing process, the shopping system requests the customer a card number and a PIN (i.e. card control code) to carry out the shipment.

The *user-interaction diagram* `UI_Purchase` is composed of three states. Two of them are terminal states, since they correspond to graphical elements. They are stereotyped (⟨⟨`JTextField`⟩⟩) and labeled by a text related to the graphical element. The other state has been described in a separate *user-interaction diagram*. In general, the name of a separate *user-interaction diagrams* is the same as the name of the state, in this case `Manage shopping cart` (i.e. `UI_ManageShoppingCart` in Fig. 3). In the `UI_ManageShoppingCart` *user-interaction diagram*, the `Query catalogue` and `Shopping cart` states are also described in separate *user-interaction diagrams*.

States of *user-interaction diagrams* can be stereotyped or not. Stereotyped states represent terminal states, which can be labeled by ⟨⟨`JTextField`⟩⟩, ⟨⟨`JList`⟩⟩ and ⟨⟨`JLabel`⟩⟩ stereotypes.

Transitions can be labeled by means of *stereotypes*, *conditions* or both. For instance, a button is connected with a transition by using a ⟨⟨`JButton`⟩⟩ stereotype, and the name of the label is the name of the button. For example, a `Show cart` transition stereotyped as ⟨⟨`JButton`⟩⟩ will correspond with a button component called “Show cart”.

Conditions can represent *user choices* or *business/data logic*. The first one is a condition of the user's interaction with a UI component (related to button or list states), and the second one is a data/business logic condition.

For example, in our case study the selections from a list are modeled by conditions: in the `UI_QueryCatalogue` *user-interaction diagram* the list `Results` is modeled by a ⟨⟨`JList`⟩⟩ state and a `[Selected article]` condition detects when an element of the list is selected. Fig. 3 shows some transitions (p.e., `[Close]`, `[Exit]` or `[Proceed]`) that correspond with conditions of the *user choice* type. The `[Exit]` output transition of the `Manage shopping cart` state means that the user has pressed a button called `Exit`, which has been defined in the separate `UI_ManageShoppingCart` *user-interaction diagram*. Nevertheless, the `[cart not empty]` condition is a *data/business logic* condition.

The usual way of “condition/event” transition can connect (non-) terminal states to (non-) terminal states. A condition/event transition between states means which condition should be present to trigger the event. In our case study, an event can only be a button. For instance, to remove an article from the shopping cart, it must previously be selected from the cart list (i.e. `UI_ShoppingCart` in Fig. 3). In addition, boolean conditions can be used in transitions from non-terminal states to other states. This particular kind of boolean conditions as used to control the logic of the secondary user-interaction diagram into the main user-interaction diagram. There are two cases. The first case when the

Table 5
Graphical components' features

Java classes	Stereotypes	Sections	Input/output
JButton	⟨⟨`JButton`⟩⟩	Transitions	Input
JLabel	⟨⟨`JLabel`⟩⟩	States	Output
JTextField	⟨⟨`JTextField`⟩⟩	States/transitions	In/out
JList	⟨⟨`JList`⟩⟩	States	Output
JFrame	None	Use cases	In/out

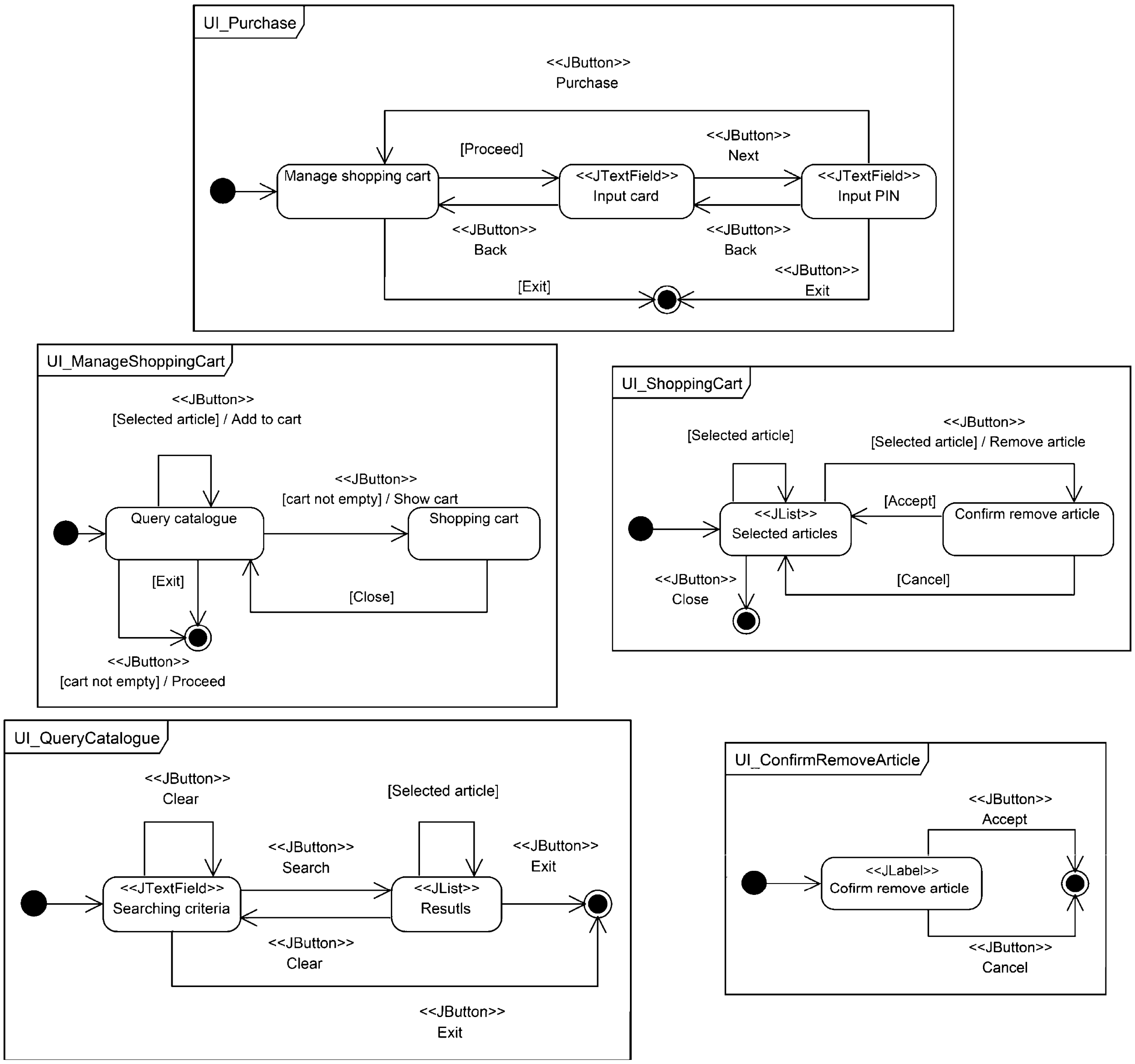

Fig. 3. The whole user-interaction diagram of the `Purchase` use case.

boolean condition refers to a "*exit condition*" of the sub-diagram. For instance, the `[Exit]` transition from `Query catalogue` refers to the exit button clicked when the user closes the `Query catalogue` window. However, the `[Selected article]` condition in the transition triggered from `Query catalogue` state is not an exit condition but an *internal transition* on `UI_QueryCatalogue` diagram which is checked from outside. It makes `UI_ManageShoppingCart` to modify the logic of `UI_QueryCatalogue`. In other words, the logic of `UI_ManageShoppingCart` "interrupts" the logic of `UI_QueryCatalogue`.

4.3. UI diagram

Once *user-interaction diagrams* have been described, the system designer proceed to build the *UI diagram*. This kind of diagram contains new use cases which are some non-terminal states of the user interaction diagrams. In addition, the system designer has to identify use case relationships in the new *UI diagram* as follows.

4.3.1. Include relationships

Let us consider the purchasing process described in the previous *user-interaction diagrams*.

Purchase use case is a window that includes a non-terminal state Manage shopping cart. This state is described by means of a separate user-interaction diagram. In addition, the logic of this state is not modified in the UI_Purchase user interaction diagram. It integrates the logic UI_ManageShoppingCart diagram by checking which buttons (i.e. Exit and Proceed) the user pressed when (s)he exited from Manage shopping cart state. This kind of checking on "*exit conditions*" does not modify the logic of the included use cases. Therefore, that use case could be related by ⟨⟨include⟩⟩ relationship to the main use case. However, the system designer has decided to consider only a window for the tasks of Purchase and Manage shopping cart and an additional window for Shopping chart task occurring in the user interaction diagram UI_ManageShoppingCart. For this reason, there are only one inclusion relationship from the Purchase use case to Shopping cart use case (see Fig. 4).

The system designer has also identified an inclusion relationship between Manage catalogue and Withdraw article, Modify article and Add article use cases (Fig. 5). Here, four windows can be optionally opened (depending on a menu) from the Manage catalogue window. In addition, the Administrator identification window is mandatorily opened from the Manage catalogue window in order to carry out the system's administrator tasks.

4.3.2. Generalization relationships

In order to illustrate the generalization/specialization relationship we will focus our attention to

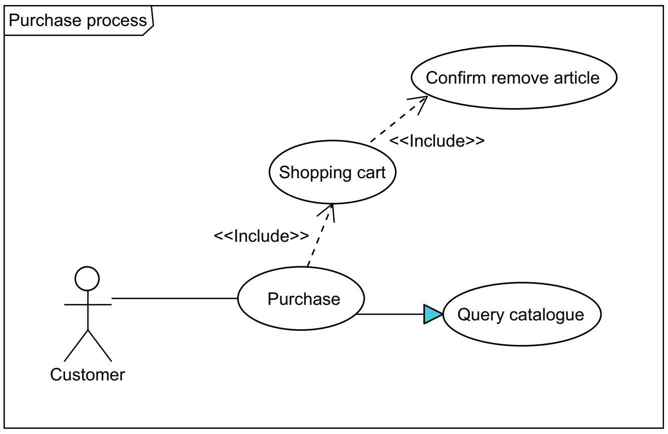

Fig. 4. A user-interface diagram for the purchasing process.

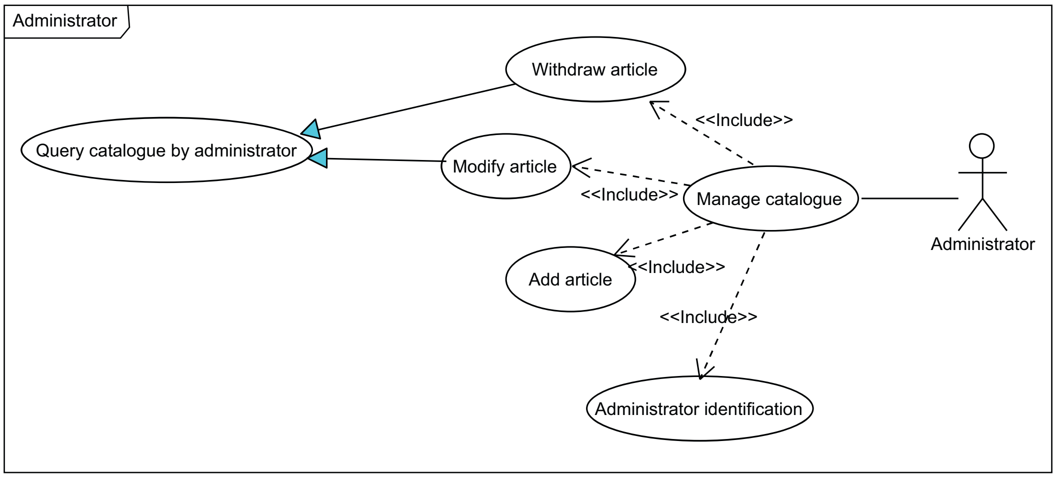

Fig. 5. A piece of the user interface of the administrator side.

three use cases: Purchase, Query catalogue and Query catalogue by administrator. In previous sections we have identified two cases of generalization/specialization.

The first case can be identified by means of the Query catalogue and Manage shopping cart states. Here the UI_Purchase user-interaction diagram contains a state (use case) Manage shopping cart which specializes Query Catalogue in the following sense. The UI_QueryCatalogue user-interaction diagram describes how to query the catalogue of the IBS by introducing the searching criteria and showing the results. However, the UI_ManageShoppingCart user interaction diagram can interrupt the querying process by adding the searched items to the shopping cart. It is specified by adding the button Add to cart as transition from (and to) Query catalogue with the condition [Selected Article]. Hence the logic of UI_QueryCatalogue is modified in UI_ManageShoppingCart diagram, and we can identify a specialization relationship between Manage shopping cart and Query Catalogue use cases. However, the system designer has decided to consider a unique window for Purchase and Manage shopping cart. Therefore, the generalization relationship is defined between Purchase and Query catalogue. Furthermore, it is supposed that there will be a window for Query catalogue from which Purchase inherits.

The second case is the relation between Query catalogue and Query catalogue by administrator use cases. Here, the administrator is supposed to have higher privileges for querying the catalogue and, therefore, the user-interaction diagram of the Query catalogue by administrator (see Fig. 7) specializes the UI_QueryCatalogue user-interaction diagram in the following sense.

The states of the UI_QueryCataloguebyAdministrator diagram corresponding with the searching criteria and results are modified with respect to the UI_QueryCatalogue diagram. The fields of searching and results are supposed to be different but the logic is similar. In other words, the Query catalogue use case can be replaced by Query catalogue by administrator use case since the states can be replaced. In this case we can identify a generalization relationship between them. Analogously, Withdraw article and Modify article use cases combine both kinds of specialization, once they have specialized the Query catalogue by administrator use case in the same way as Purchase specializes Query catalogue use case, and indirectly specializes Query catalogue use case (see Figs. 6 and 7).

The complete *UI diagram* of our case study can be seen in Fig. 8.

4.4. UI-class diagrams and UI prototypes

Once the *UI diagram* has been built and a set of *user-interaction diagrams* has been obtained, now we can generate a *UI-class diagram*.

In our case study, the *UI-class diagram* is built from Java *swing* classes. Use cases are translated into classes with the same name as these use cases. The translated classes specialize a Java *Frame* class.

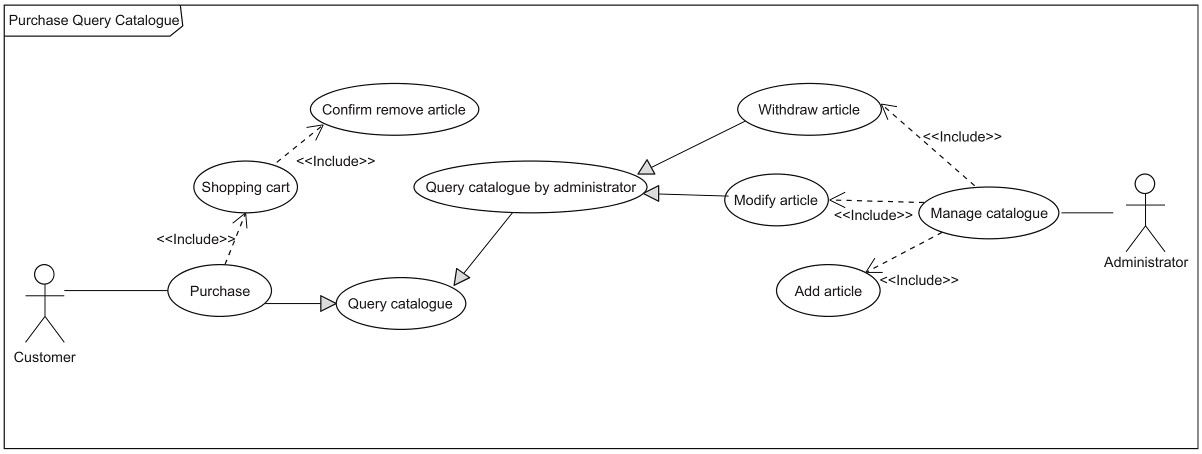

Fig. 6. A piece of the user-interface diagram.

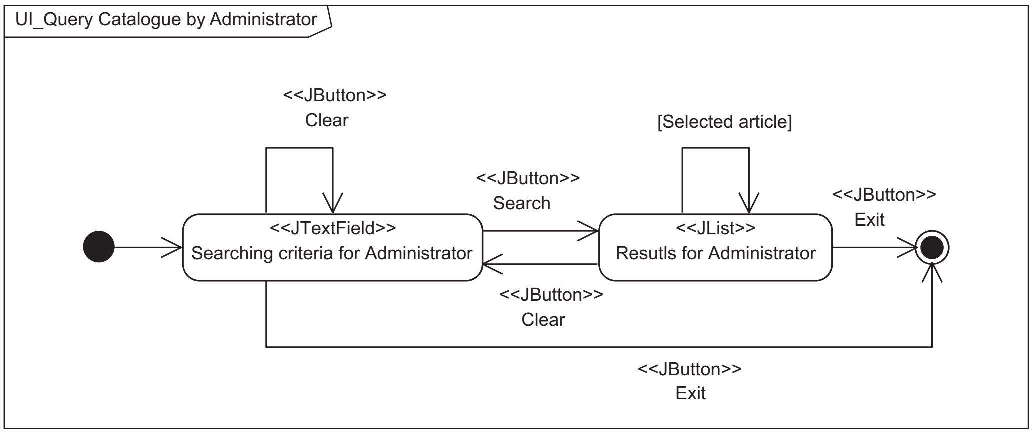
UI_Query Catalogue by Administrator
<<JButton>> Clear
[Selected article]
<<JButton>> Search
<<JButton>> Exit
<<JTextField>> Searching criteria for Administrator
<<JList>> Resutls for Administrator
<<JButton>> Clear
<<JButton>> Exit

Fig. 7. The user-interaction diagram for the query catalogue by administrator.

The components of the applet or frame (use case) are described in the *user-interaction diagrams*. A terminal state is translated in the class (frame) as a Java *swing* attribute represented by the stereotype of the state. For example, those terminal states stereotyped as ⟨⟨JTextField⟩⟩ are translated into a *JTextField* attribute in the *UI-class diagram*. Fig. 9 shows the *UI-class diagram* of the customer's side.

The *UI-class diagram* contains five windows, four of which directly specialize the Java *JFrame* class: the GUI_Customer class, the GUI_Query_catalogue class, the GUI_Confirm_remove_article class and the GUI_Shopping_cart class. The other class (i.e. GUI_Purchase class) inherits from the Java *JFrame* class through the GUI_Query_catalogue class. These five classes correspond to those four use cases at the customer's side in the *UI diagram* together with the customer actor. Furthermore, note how the stereotyped states and transitions in the *user-interaction diagrams* of the purchase process are translated into Java *swing* attributes in the *UI-class diagram*. The stereotyped name of a transition or state is translated into the appropriate Java swing class attribute. For example, the ⟨⟨JButton⟩⟩ stereotype of the Proceed transition that occurs in the UI_ManageShoppingCart *user-interaction diagram* (see Fig. 3) is translated into a JButton attribute.

Finally, rapid UI prototypes can be obtained from the *UI-class diagram*. Such prototypes are *draft* versions which should be improved and checked by using user centered design and user evaluation methods.

Fig. 10 shows a visual result of the Purchase window. Note how the Purchase window is very similar to the Query Catalogue window, except that the second one includes three additional buttons. This similarity between windows was revealed in the *UI diagram* as a generalization relationship between use cases: between the Query catalogue and Purchase use cases. In the IBS final prototype, the customer will always work on a Purchase window opened from the Customer window, never on a Query Catalogue window, though the former inherits the behavior of the latter (i.e. by the relation of generalization). Let us remark that the Shopping cart window will be invoked from the Purchase window, and Purchase inherits from Query catalogue window.

The Shopping Cart window (Fig. 10) will be opened when the Show Cart button is pressed on the purchase process (Fig. 10). In Fig. 8 note how in the *UI diagram* both windows are associated by means of an inclusion relation between use cases. On the other hand, the two warning windows (Fig. 10) are also associated: the Remove article window and the Shopping Cart window, and the Proceed window and the Purchase window.

Let us remark that user-interaction diagrams reveal how UI components are enabled and disabled depending on the user interaction. For instance, given a state, only those buttons representing transitions from the state will be enabled.

For space reasons, we have included here a part of the case study. A complete version of the project is available at http://indalog.ual.es/mdd/usecases.html.

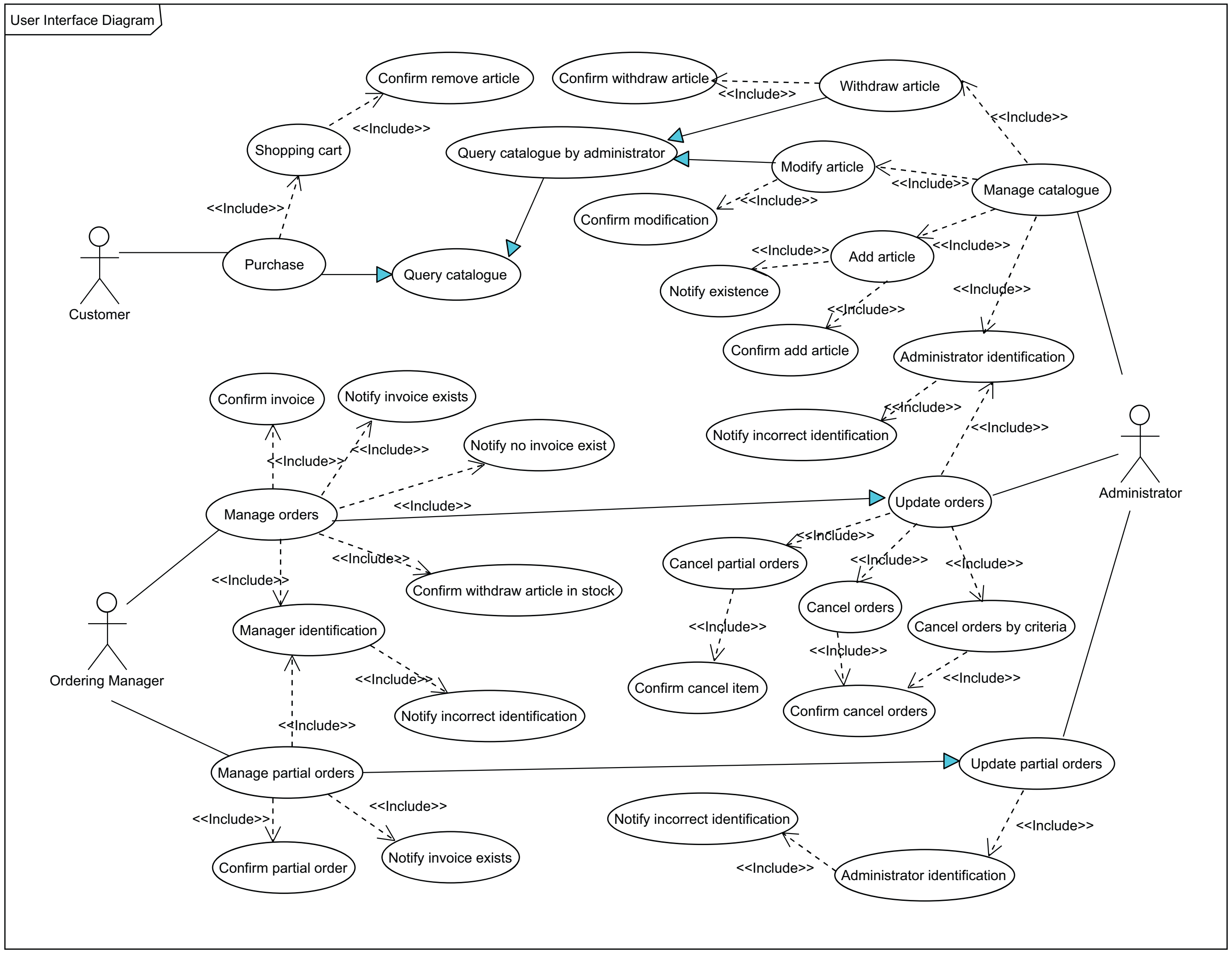

Fig. 8. The Internet shopping user-interface diagram.

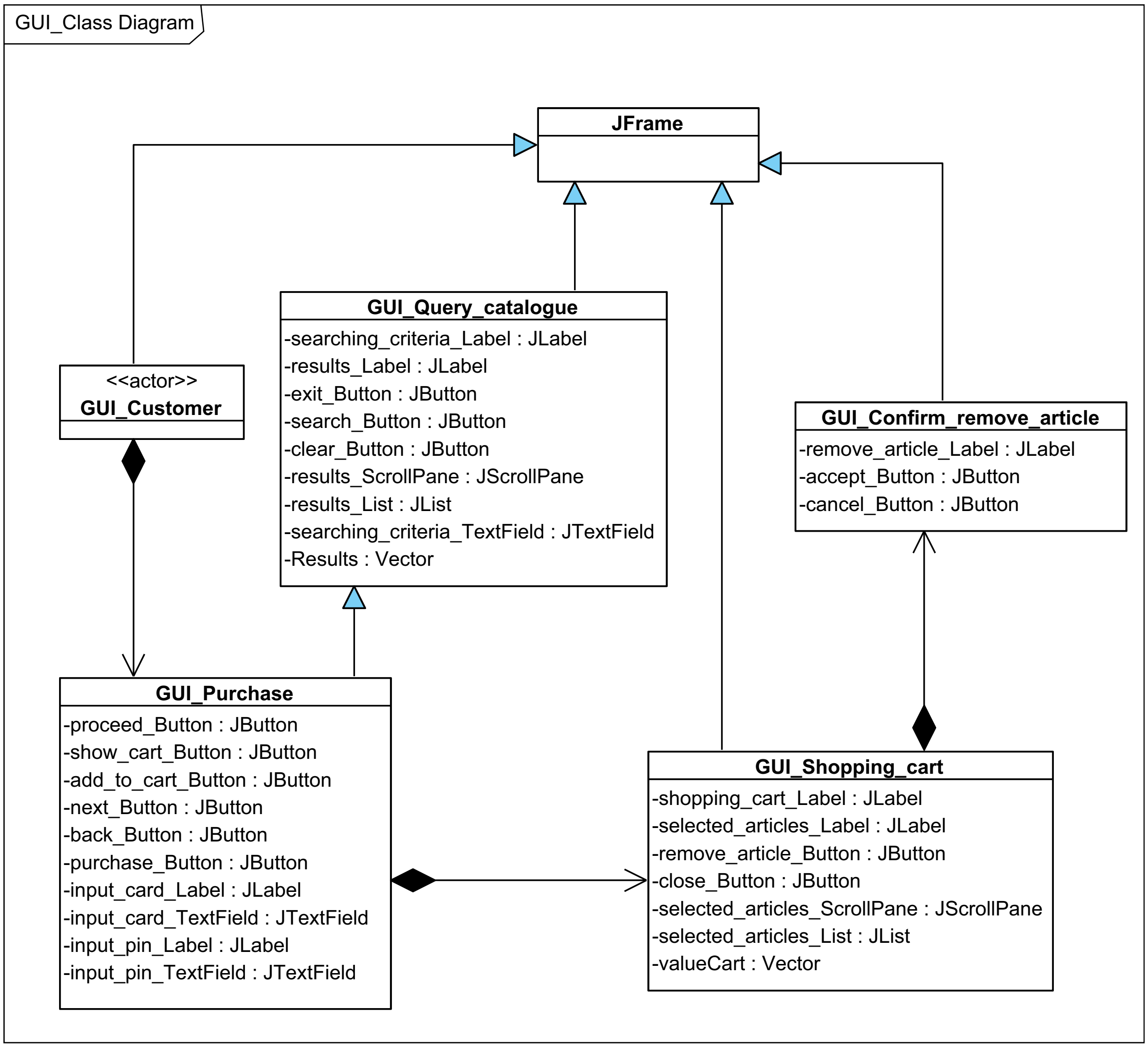
GUI_Class Diagram
JFrame
GUI_Query_catalogue
-searching_criteria_Label : JLabel
-results_Label : JLabel
-exit_Button : JButton
-search_Button : JButton
-clear_Button : JButton
-results_ScrollPane : JScrollPane
-results_List : JList
-searching_criteria_TextField : JTextField
-Results : Vector
<<actor>>
GUI_Customer
GUI_Confirm_remove_article
-remove_article_Label : JLabel
-accept_Button : JButton
-cancel_Button : JButton
GUI_Purchase
-proceed_Button : JButton
-show_cart_Button : JButton
-add_to_cart_Button : JButton
-next_Button : JButton
-back_Button : JButton
-purchase_Button : JButton
-input_card_Label : JLabel
-input_card_TextField : JTextField
-input_pin_Label : JLabel
-input_pin_TextField : JTextField
GUI_Shopping_cart
-shopping_cart_Label : JLabel
-selected_articles_Label : JLabel
-remove_article_Button : JButton
-close_Button : JButton
-selected_articles_ScrollPane : JScrollPane
-selected_articles_List : JList
-valueCart : Vector

Fig. 9. A *UI-class diagram* obtained from *user-interface diagrams* and *user-interaction diagrams*.

5. Formalization of the technique

In this section we will formalize the described technique. The interest of this formalization is double:

- A MDD technique that involves multiple perspectives can lead to *inconsistent models*. Model should be integrated and consistent. Our formal definition expresses when the UI and user interaction diagrams are consistent or well-formed. Such consistence proving can be considered as a *model checking technique*.
- Our technique goes towards the *automatic generation of code* from UML models. Therefore, a formal translation of *user-interaction diagrams* and *UI diagrams* into UIs is defined (in particular, this translation can be used to generate UI-class diagrams).

Firstly, we will provide a formal definition of *well-formed UI diagrams*. In particular, we will define the *include* and *generalization* use case relationships. We will also define a *well-formed UI diagram* which follows some restrictions. In addition, we will provide an *abstract definition of UIs*, and we will define two *abstract relationships between UI*: *inclusion and generalization*. This will allow us to define a generic transformation technique for *UI diagrams* into a set of abstract UIs.

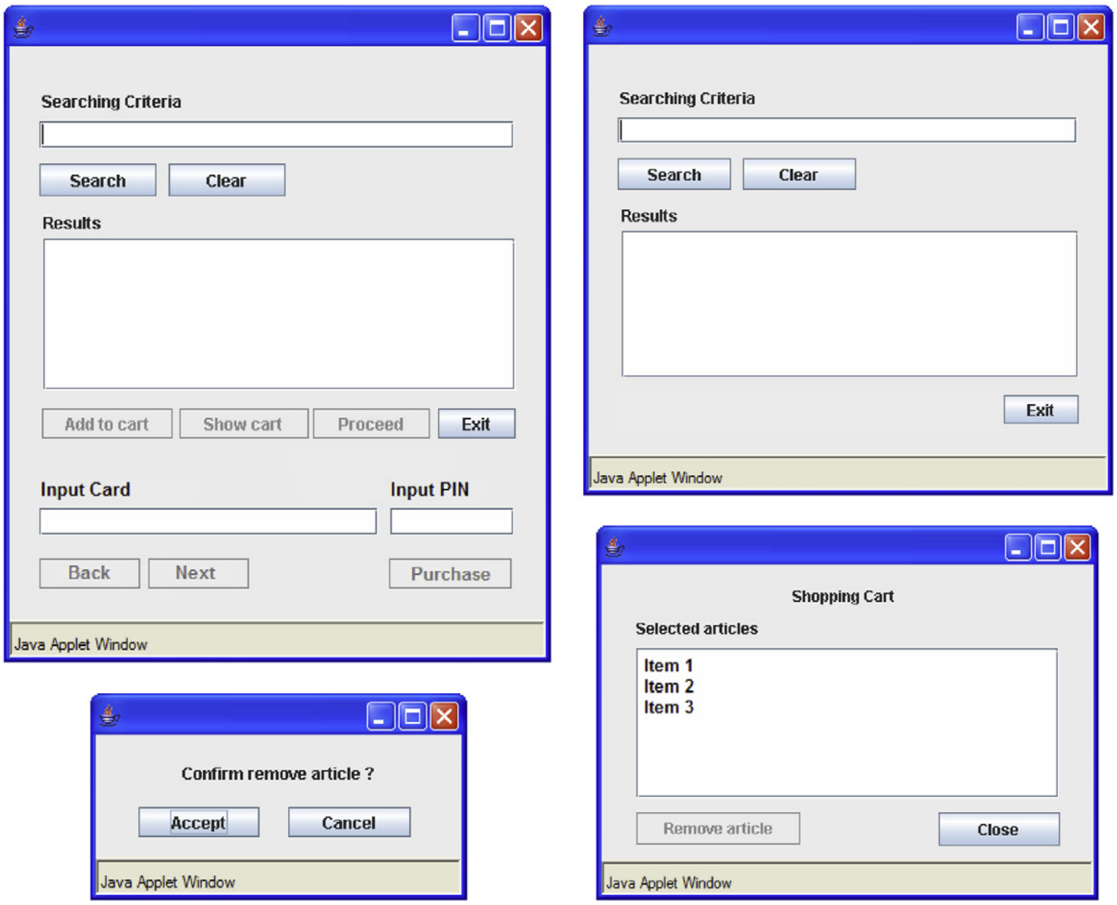

Fig. 10. The applet windows of the customer's side.

Now, let us define a *UI diagram* as follows:

Definition 5.1 (*UI diagram*). A *UI diagram*

$UID = (n, ACT, UC, -\!\!\triangleright, —, \overset{<<i>>}{\dashrightarrow})$

consists of: (i) a diagram name n; (ii) a finite set ACT of actor's names $p, q, r, \ldots$; (iii) a finite set UC of use cases $u, v, w, \ldots$; (iv) and three relations $-\!\!\triangleright$, $—$ and $\overset{<<i>>}{\dashrightarrow}$, where (1) $-\!\!\triangleright \subseteq (ACT \times ACT) \cup (UC \times UC)$; (2) $— \subseteq ACT \times UC$; (3) and $\overset{<<i>>}{\dashrightarrow} \subseteq UC \times UC$. As usual, we write $p -\!\!\triangleright q$, rather than $(p, q) \in -\!\!\triangleright$, and analogously for $—$ and $\overset{<<i>>}{\dashrightarrow}$.

Now, we formally define a use case as follows:

Definition 5.2 (*Use case*). A use case

$u = (n, S, SI, IS, OS, COND, \rightarrow)$

consists of:

(1) a use case name n;
(2) a finite set S of states which consists of:
   (a) a finite set UC of use cases;
   (b) a finite set SS of stereotyped states of the form (sn, p) where sn is a state name and $p \in OS$;
   (c) three special states SP, the initial, final and branching states;
(3) a finite set SI of stereotyped interactions of the form $[C]/(in, i)$ where $C \in COND$, in is an interaction name, and $i \in IS$. The condition $[C]$ is optional;
(4) a finite set IS of input stereotypes $i, j, \ldots$;
(5) a finite set OS of output stereotypes $p, q, \ldots$;
(6) a finite set $COND$ of boolean conditions $C, D \ldots$;
(7) a transition relation $\rightarrow \subseteq S \times_{\lambda} (SI \cup COND) \times S$. As usual, we write $A \overset{\lambda}{\rightarrow} B$ rather than $(A, \lambda, B) \in \rightarrow$, where λ can be $[C]$ or $[C]/(in, i)$.

In a *user-interaction diagram* we have two kinds of states: *stereotyped* and *non-stereotyped* states. The *non-terminal states* correspond with use cases.

Transitions are labeled with (*conditioned*) *stereotyped interactions* and *conditions*.

Notation: We denote by *name*(u) the name of a use case u. Also we denote the set of use cases—resp. transitions—of a use case u as *usecases*(u)—resp. *transitions*(u). $SI(u)$—resp. $SS(u)$—denotes the set of stereotyped interactions (in, i) in u—resp. stereotyped states (sn, p) in u. Finally, we denote by *exit*(u) the *exit conditions* of a use case u, defined as the set of interaction names *in* on transitions $s \overset{[C]/(in,i)}{\rightarrow} s'$ together boolean conditions C of transitions $\overset{[C]}{\rightarrow}$ which go to the final state in the use case u. Finally, we assume a (*partial and reflexive*) replacement relation $\sqsubseteq$ defined for stereotyped states, stereotyped interactions and conditions.

The replacement relation $\sqsubseteq$ can be extended to use cases, as follows:

Definition 5.3 (*Replacement of use cases*). Given use cases u, v: $v \sqsubseteq u$ whenever $(s, \lambda, t) \in \rightarrow$ belongs to *transitions*(u) iff there exists $(s', \lambda', t') \in \rightarrow$ of *transitions*(v), such that $s' \sqsubseteq s$, $t' \sqsubseteq t$ and $\lambda' \sqsubseteq \lambda$.

Now, we can define the inclusion and generalization relationship between use cases as follows:

Definition 5.4 (*Use case inclusion*). Given two use cases u, v we say that u includes v if $v \in$ *usecases*(u).

Definition 5.5 (*Use case generalization*). Given two use cases u, v we state that u generalizes v, if there exists $w \in$ *usecases*(v) such that $w \sqsubseteq u$.

Therefore, the states and transitions of a more general use case can be replaced in a particular use case by more particular stereotyped states and transitions. In addition, the more particular use case can add new ones. Next, we will define the well-formed use cases.

Definition 5.6 (*Well-formed use cases*). A use case u is well-formed when:

(1) for all $s \overset{\lambda}{\rightarrow} t \in$ *transitions*(u) then λ has the form $[C]/(in, i) \in SI$ iff
  (a) $s = (sn, p)$, $p \in OS$, or
  (b) $s \in$ *usecases*(u) and s generalizes u;
(2) for all $v \in$ *usecases*(u) then there exists $v \overset{\lambda}{\rightarrow} t \in$ *transitions*(u) such that λ has the form $[C]$ or $[C]/(in, i)$ for every $C \in$ *exit*(v).

Well-formed use cases are those fulfilling the following conditions:

(1.a) an output component triggers an input interaction;
(1.b) input interactions are added in a more general use case in order to obtain more particular ones; and finally,
(2) the exit conditions of a non-terminal state must be included in the main use case.

Now, a *well-formed* UI diagram includes well-formed use cases, and the *include* and *generalization* relationships between use cases in the *well-formed UI diagram* correspond with the analogous relations defined for use cases.

Definition 5.7 (*Well-formed use interface diagram*). A well-formed *UI diagram* $UID = (n, ACT, UC, \rightarrow\!\triangleright, —, \overset{<<i>>}{\dashrightarrow})$ satisfies that:

(1) every $u \in UC$ is well-formed;
(2) for all $u, u' \in UC$, $u' \rightarrow\!\triangleright u$ if u generalizes u';
(3) and for all $u, u' \in UC$, $u \overset{<<i>>}{\dashrightarrow} u'$ if u includes u'.

Let us remark that the inclusion and generalization relationships between use cases in well-formed *UI diagrams* are a subset of the same relationships between use cases.

Now, we can formally define our transformation technique which provides each *UI diagram* with a set of UI components.

With this aim, we will provide an abstract definition of UI and UI components. A UI has a name and contains a set of UI components which in particular can be UIs. UI components are stereotyped interactions and states which represent the input and output UI components.

Definition 5.8 (*User interface*). A UI $U = (n, W, I, O)$ consists of:

(1) a UI name n;
(2) a finite set W of UIs;
(3) a finite set I of stereotyped interactions (in, i);
(4) and a finite set O of stereotyped states (sn, p).

UIs can be compared by means of generalization and inclusion relationships. The first one corresponds with an inheritance relationship, and the second one with a containment relationship.

Definition 5.9 (*UI generalization*). Given two UIs: (n, W, I, O), (n', W', I', O'), we say that (n, W, I, O) generalizes (n', W', I', O') when:

(a) for all $U \in W$, there exists $U' \in W'$ such that U generalizes U';
(b) for all $i \in I$, there exists $i' \in I'$ such that $i' \sqsubseteq i$;
(c) and for all $o \in O$, there exists $o' \in O'$ such that $o' \sqsubseteq o$.

Definition 5.10 (*UI inclusion*). Given two UIs: U and U', we say that $U = (n, W, I, O)$ includes U' if $U' \in W$.

Finally, we define the transformation of a *UI diagram* into a set of UIs.

Definition 5.11 (*UI associated to a UI diagram*). Given a well-formed *UI diagram* $UID = (n, ACT, UC, -\!\!\rhd, —, \overset{<<i>>}{\dashrightarrow})$, we define the UI associated to UID, denoted by $UI(UID)$, as the set $\{UI(p) | p \in ACT \textit{ is a user}\}$, where

$UI(p) = (p, W, I, O)$

where

$$\begin{cases} W = \{UI(u) | p—u\} \cup \{UI(u) | p'—u, p -\!\!\rhd^{*} p'\}, \\ I = \emptyset, \\ O = \emptyset \end{cases}$$

and

$UI(u) = (name(u), W, I, O)$

where

$$\begin{cases} W = \{UI(v) | u \overset{<<i>>}{\dashrightarrow} v\}, \\ I = I(u), \text{where} \\ \quad I(u) = \bigcup_{v \,\in\, usecases(u) \text{ and not } u \overset{<<i>>}{\dashrightarrow} v} I(v), \\ O = S(u), \text{where} \\ \quad S(u) = \bigcup_{v \,\in\, usecases(u) \text{ and not } u \overset{<<i>>}{\dashrightarrow} v} S(v), \end{cases}$$

where $-\!\!\rhd^{*}$ denotes the transitive closure of the relation $-\!\!\rhd$.

Finally, we can prove the following result from our transformation technique:

Theorem 5.1. *The UI associated to a well-formed UI diagram satisfies the following conditions*:

(a) *for all p, p' users in ACT, $p' -\!\!\rhd p$ then $UI(p)$ generalizes $UI(p')$;*
(b) *for all $u, u' \in UC$, $u' -\!\!\rhd u$ then $UI(u)$ generalizes $UI(u')$;*
(c) *for all $u, u' \in UC$, $u \overset{<<i>>}{\dashrightarrow} u'$ then $UI(u)$ includes $UI(u')$;*
(d) *for all p user in ACT and $u \in UC$, $p—u$ then $UI(p)$ includes $UI(u)$*

Proof. (a) By definition $UI(p') = (p', W', I', O')$ where $W' = \{UI(u') | p'—u'\} \cup \{UI(u') | p''—u', p' -\!\!\rhd^{*} p''\}$, $I' = \emptyset$ and $O' = \emptyset$ and $UI(p) = (p, W, I, O)$ where $W = \{UI(u) | p—u\} \cup \{UI(u) | p'''—u, p -\!\!\rhd^{*} p'''\}$, $I = \emptyset$ and $O = \emptyset$. Now, if $p' -\!\!\rhd^{*} p$ then $p' -\!\!\rhd^{*} p'''$ and therefore $W \subseteq W'$, $I = I' = \emptyset$ and $O' = O = \emptyset$,concluding that $UI(p) = (p, W, I, O)$ generalizes $UI(p') = (p', W', I', O')$.

(b) By definition $UI(u) = (name(u), W, I, O)$ where

$$W = \{UI(v) | u \overset{<<i>>}{\dashrightarrow} v\}, \quad I = I(u),$$

$$I(u) = \bigcup_{v \,\in\, usecases(u) \text{ and not } u \overset{<<i>>}{\dashrightarrow} v} I(v), \quad O = S(u),$$

$$S(u) = \bigcup_{v \,\in\, usecases(u) \text{ and not } u \overset{<<i>>}{\dashrightarrow} v} S(v).$$

and $UI(u') = (name(u'), W', I', O')$ where

$$W' = \{UI(v') | u' \overset{<<i>>}{\dashrightarrow} v'\}, \quad I' = I(u'),$$

$$I(u') = \bigcup_{v' \,\in\, usecases(u') \text{ and not } u' \overset{<<i>>}{\dashrightarrow} v'} I(v'), \quad O' = S(u'),$$

$$S(u') = \bigcup_{v' \,\in\, usecases(u') \text{ and not } u' \overset{<<i>>}{\dashrightarrow} v'} S(v').$$

In addition, if $u' -\!\!\rhd u$ then there exists $w \in usecases(u')$ such that $w \sqsubseteq u$. Now, we can proceed by induction in the number of *included use cases of* u, that is, in the number of elements of the set $iuc(u)$ defined as $iuc(u) = usecases(u) \bigcup_{v \in usecases(u)} iuc(v)$ in order to prove that $UI(u)$ generalizes $UI(w)$.

Base case: $iuc(u) = \emptyset$. That is, the number of included use cases of u is zero. Then u does not include any v; and therefore $W = \emptyset$, $I = SI(u)$ and $O = SS(u)$.

Now, given that $w \sqsubseteq u$ then: $(s, \lambda, t) \in \rightarrow$ belongs to $transitions(u)$ iff there exists $(s', \lambda', t') \in \rightarrow$ of $transitions(w)$, such that $s' \sqsubseteq s$, $t' \sqsubseteq t$ and $\lambda' \sqsubseteq \lambda$.

Therefore for every $i \in SI(u)$ there exists $i' \in SI(w)$ such that $i' \sqsubseteq i$, and analogously for $SS(u)$; therefore $UI(u)$ generalizes $UI(w)$.

Inductive case: $iuc(u) \neq \emptyset$. That is, the number of included use cases of u is greater than zero. We can reason that if $w \sqsubseteq u$ then, by definition, for every use case v of u there exists a use case v' of w such that $v' \sqsubseteq v$.

Now, the number of included use cases of v is strictly smaller than the number of use cases of $u : iuc(v) < iuc(u)$.

Now, we can reason by induction hypothesis that $UI(v)$ generalizes $UI(v')$. Reasoning analogously to the base case for every $i \in SI(u)$ there exists $i' \in SI(w)$ such that $i' \sqsubseteq i$, and the same for $SS(u)$.

Therefore we can conclude that $UI(u)$ generalizes $UI(w)$.

Finally, we can reason that $UI(u)$ generalizes $UI(w)$, and $UI(u')$ includes $UI(w)$, trivially by definition, and therefore, by definition, $UI(u)$ generalizes $UI(u')$. The cases (c) and (d) are trivially true by definition. $\square$

6. Conclusions and future work

In this paper, we have studied a MDD technique for modeling WIMP-based UIs. We have defined a new kind of UML diagram (i.e. the *UI diagram*) which specializes the use case diagram for UI modeling. In addition, we have shown how to describe use cases by means of specialized activity diagrams (*user-interaction diagrams*), in order to specify the dialogue model of each UI. Finally, we have shown how to generate class diagrams for UI, and how to build *draft UI prototypes* for the UI.

6.1. Advantages of the proposed technique

Our technique for UI modeling is based on UML, that is, *it adopts UML diagrams for describing UIs*. Some UML diagrams are adapted to UI but it is not a real extension. Firstly, the proposed technique specializes the use case diagram for UI representation, maintaining the *intended semantics of use cases but interpreted in the context of UIs*. Secondly, it introduces *new stereotypes* for the class and activity diagrams preserving the same structure and semantics. It allows the UML system designers to adopt our technique with a short training, and a UML CASE tool to be used for our technique.

The most relevant extension of UML in our proposal is the definition of a new diagram called *UI diagram*, built from the UML use case diagram. The adoption of the UML use case model for UI modeling has already been discussed in other works [31,58–61]. Use cases, as the "starting point" of the UI modeling, are useful for a "high-level" description of the UI. However, it is well-known that use case diagrams may include some use cases referred to parts of the system not related to UIs, such as classes, human tasks, components of other systems interacting with ours, and so on; even on decomposing use cases through the include and extend relationships, one could specify particular parts of the system which are not related with the UI: data logic can be specified in the use case diagram. For this reason, a specialized version of the use case model has been adopted.

The *UI diagram* introduces some use case diagram mechanisms (*inclusion* and *generalization* between use cases) in order to describe the structure of the UI: main and secondary tasks/windows, and *similarity* between tasks. Similarity analysis between tasks is something new with respect to other proposals based on UML [29–31,34,35,38,54]. Similarity analysis is one of the most relevant contributions of this paper. *Inclusion* expresses similarity in terms of "pieces" of user interaction with the UI which are shared by some tasks. Generalization is a more complex kind of similarity related to two concepts: *reuse+replacement* of user interactions and UI components.

One of the advantages of our proposed technique is that it offers a separation between *application level* and *UI*. Each task specified in the *user-interface diagram* is supposed to be carried out by a UI: the user introduces input data and the system replies with output data. However, some system tasks can be carried out at the application level, for instance: the handling of databases, system requests to external devices and so on, and such tasks are specified out of the *UI diagram*. Our task model (i.e. the *UI diagram*) only describes the tasks in which the user participates. This task model is completed by means of activity diagrams that describe the user dialogue and the application level tasks. Application level tasks can be described by means of natural language in boolean conditions of transitions in the *user-interaction diagrams*. In addition, the *UI diagram* also represents actors (i.e., system users) and use cases connected to actor represent tasks to be carried out by each user.

Our technique describes the dialogue model by means of a specialized version of the activity diagram. There are many works [32,36,39–42] that suggest the adaptation of UML diagrams and the activity and state diagrams for user dialogue description. Our proposal works with WIMP UIs. Therefore, the user dialogue consists of tasks in which the user selects from a list, clicks in a button, accepts/cancels, and so on. However, in the dialogue model the UI can still represent *the adaptation to the user interaction* in the following sense: from a given state, it is possible to find visible/enabled or disabled buttons, depending on the operation. For instance, in the presented Internet shopping system case study, "purchase" button is disabled whenever the shopping cart is empty.

An interesting contribution to our paper is the *formalization* of the proposed MDD-based modeling technique. Multiple perspectives of the same system can be inconsistent, and a CASE tool should detect failures on design due to inconsistencies. For this reason a formal definition of the required *consistence properties* is needed. Proving such properties for a given modeling can be considered as a *model checking technique*. In addition, our work goes towards the definition of an *automatic generation of code from UML designs*. Automatizing is only possible from a well-defined process. Our technique has been formally specified in order to define a *model transformation* of UML models into a UI. In particular this transformation can be used *for generating the UI-class diagram*. We have presented the definition of *well-formed UI and user-interaction diagrams* which are consistent with each other. We have defined an *abstract definition of UIs* and two relationships on them: *inclusion and generalization*. Inclusion between two UIs means that one of the UIs is the container and the other is the contained. Generalization is equivalent to *inheritance* between window classes. Like conclusion of our formalization, we have proved that inclusion between use cases in the *UI diagram* corresponds with inclusion between UIs, and generalization between use cases corresponds to inheritance.

6.2. Shortcomings of the proposed technique

The main limitations of the approach are two.

Firstly, we *restrict our technique to WIMP interfaces*. We believe that our technique could be extended to *Post-WIMP* interfaces based on *hypertext* with *dynamic content* or the handling of *adaptation* and *multi-modality*. Dialogue models for *Web* content have been studied in some previous works [41,42,45–47]. Our intention is to extend our work in the future by following the quoted proposals.

Secondly, our proposed technique is only a piece of a complete *method* (i.e. set of processes, notations for steps, etc.) for UI design. In particular, the *layout part* of the UIs is not covered by our technique and therefore the technique does not properly contribute to a complete *UI design method*. *Ergonomic criteria* [2,14–16,48,49] are still crucial for the *usability* of the UI. The main goal of the proposed technique is the dialogue and UI modeling in the framework of UML. In addition, the proposed technique is useful for rapid prototyping of UIs in modeling phases. However, such preliminary design should be just a prototype, and we recommend the system designer to improve and check the prototypes by using *user center design and evaluation methods* [17,18].

6.3. Extensions of the proposed technique

We plan to study how to *integrate the UI view with the data and business logic* by using other types of diagrams (sequence diagrams and class diagram for application objects). Application tasks are specified in the proposed technique by means of boolean conditions but usually they will require database interactions. Some kind of UML diagram could be used for describing how the UI interacts with database components.

On the other hand, the *automatic verification of UI specifications* is also an interesting extension of our work. Some properties [62] can be checked from high-level specifications of UIs. Model checking techniques can be used for that. There are many interesting works in the field (see [62–67] for some examples). One of the contributions of our paper is the formalization of our technique, whose aim is in particular to prove properties of consistence of the developed models. We believe that some other properties could be proved in our models by using the proposed formalization.

Another extension of our work is the study of *XML-based representation of our modeling technique*. There are some interesting proposals like XIML [50], UIML [51], UsiXML [52], Teresa + XML [53], WebML [45] together with the UI work by the W3 consortium [37] and Mozilla.[1] The aim of such proposals is to provide XML dialects

[1]http://www.mozilla.org/projects/xul/

for UI specification. On one hand, XML-based descriptions of UI layout and behavior allow the system designers to improve their work: model repositories, abstract UIs, patterns, and so on. On the other hand, interoperability between UI tools is mandatory for complex software architectures. We believe that an XML dialect could be defined from our models as future work.

Finally, we would like also to *incorporate our technique in a CASE tool in order to automatize the MDD technique*. Our technique has been extensively used in student projects in the classroom. However, the main tasks are done by hand for students due to the lack of a CASE tool supporting our technique. The tasks to be automatized are for instance the generation of UI-class diagrams from user interaction diagrams. This is a model transformation in which each UI component is translated into an icon class of the UI-class diagram and the corresponding associations between system windows and widgets could be obtained. Secondly, the generation of the UI diagram could be supported by the CASE tool with developer participation, that is, the CASE tool should ask the developer which states of the user interaction diagrams will be system windows and which of them not. Finally, the CASE tool could help to generate UI prototypes.

Acknowledgments

The authors would like to thank the anonymous referees for their insightful comments and suggestions that greatly helped them improve the contents and readability of the paper. This work has been partially supported by the EU (FEDER) and the Spanish MEC under Grants TIN2005-09207-C03-02 and TIN2007-61497.

References

[1] OMG, Unified Modeling Language Specification, Version 2.0, Technical Report, Object Management Group ⟨http://www.omg.org/technology/documents/formal/uml.htm⟩, 2005.

[2] J. Nielsen, Usability Engineering, Academic Press, New York, 1993.

[3] T.R.G. Green, Cognitive dimensions of notations, in: Proceedings of the Fifth Conference of the British Computer Society, Human–Computer Interaction Specialist Group on People and Computers V, Cambridge University Press, New York, NY, USA, 1989, pp. 443–460.

[4] E. Soloway, S. Iyengar, Empirical Studies of Programmers, Human/Computer Interaction Series, vol. 1, Ablex Publishing Corporation, Norwood, NJ, 1986.

[5] B.A. Myers, User interface software tools, ACM Transactions on Computer Human Interaction 2 (1) (1995) 64–103.

[6] B. Myers, S.E. Hudson, R. Pausch, Past, present, and future of user interface software tools, ACM Transactions on Computer Human Interaction 7 (1) (2000) 3–28.

[7] G.R. Pfaff, User Interface Management Systems, Springer, Berlin, Germany, 1985.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture, A System of Patterns, vol. 1, Wiley, New York, 1996.

[9] M. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, MA, 1983.

[10] J. Coutaz, PAC: an implementation model for dialogue design, in: Proceedings of the second IFIP International Conference on Human–Computer Interaction, INTERACT'87, North-Holland, Amsterdam, 1987, pp. 431–437.

[11] J. Coutaz, Architecture models for interactive software, in: Proceedings of the 1989 European Conference on Object-Oriented Programming, ECOOP'89, 1989, pp. 383–399.

[12] L. Bass, R. Little, R. Pellegrino, S. Reed, R. Seacord, S. Sheppard, The arch model: Seeheim revisited (version 1.0), The UIMS Tool Developers Workshop, April, ACM SIGCHI Bulletin 24 (1) (1991).

[13] J. Coutaz, Formal Methods in Human–Computer Interaction, Software Architecture Modelling: Bridging Two Worlds Using Ergonimics and Software Properties, Springer, Berlin, Germany, 1998, pp. 49–73.

[14] M. McCurdy, C. Connors, G. Pyrzak, B. Kanefsky, A. Vera, Breaking the fidelity barrier: an examination of our current characterization of prototypes and an example of a mixed-fidelity success, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '06, ACM Press, New York, 2006, pp. 1233–1242.

[15] J.A. Landay, SILK: sketching interfaces like krazy, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'96, ACM Press, NY, USA, 1996, pp. 398–399.

[16] J. Lin, M. Thomsen, J.A. Landay, A visual language for sketching large and complex interactive designs, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'02, ACM Press, NY, USA, 2002, pp. 307–314.

[17] K. Vredenburg, J.-Y. Mao, P.W. Smith, T. Carey, A survey of user-centered design practice, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'02, ACM Press, NY, USA, 2002, pp. 471–478.

[18] J.-Y. Mao, K. Vredenburg, P.W. Smith, T. Carey, User-centered design methods in practice: a survey of the state of the art, in: Proceedings of the 2001 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '01, IBM Press, 2001, p. 12.

[19] F. Paternò, Model-Based Design and Evaluation of Interactive Applications, Springer, Berlin, Germany, 1999.

[20] P. Pinheiro da Silva, User interface declarative models and development environments: a survey, in: Proceedings of the International Workshop on Interactive Systems: Design, Specification, and Verification, DSV-IS 2000, Lecture Notes in Computer Science, vol. 1946, Springer, Berlin, Germany, 2000, pp. 207–226.

[21] S. Card, T. Moran, A. Newell, The Psycology of Human Computer Interaction, Lawrence Erlbaum, Hillsdale, 1983.

[22] A. Newell, S. Card, The prospects for psycological science in human–computer interaction, Human–Computer Interaction 1 (1985) 209–242.

[23] G. Mori, F. Paternò, C. Santoro, CTTE: support for developing and analyzing task models for interactive system design, IEEE Transactions on Software Engineering (2002) 797–813.

[24] C. Janssen, A. Weisbecker, J. Ziegler, Generating user interfaces from data models and dialogue net specifications, in: Proceedings of the ACM Conference on Human Factors in Computing Systems, INTERCHI'93, ACM Press, NY, USA, 1993, pp. 418–423.

[25] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Sacré, J. Vanderdonckt, Architecture elements for highly-interactive business-oriented applications, in: Selected Papers from the Third International Conference on Human–Computer Interaction, EWHCI '93, Springer, Berlin, Germany, 1993, pp. 83–104.

[26] F. Bodart, A. Hennebert, J. Leheureux, J. Vanderdonckt, Towards a dynamic strategy for computer-aided visual placement, in: Proceedings of the Second ACM Workshop on Advanced Visual Interfaces, AVI'94, ACM Press, NY, USA, 1994, pp. 78–87.

[27] F. Bodart, J. Vanderdonckt, On the problem of selecting interaction objects, in: Proceedings of the BCS Conference HCI'94, Cambridge University Press, Glasgow, 1994, pp. 163–178.

[28] A. Puerta, The MECANO project: comprehensive and integrated support for model-based interface development, in: Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces, CADUI'96, Presses Universitaires de Namur, Namur, Belgium, 1996, pp. 19–25.

[29] N.J. Nunes, Object modeling for user-centered development and user interface design: the WISDOM approach, Ph.D. Thesis, University of Madeira, April 2001.

[30] P. Pinheiro da Silva, Object modelling of interactive systems: the UMLi approach, Ph.D. Thesis, University of Manchester, 2002.

[31] N.J. Nunes, J. Falcao e Cunha, WISDOM—a UML based architecture for interactive systems, in: Proceedings of the Seventh International Workshop on Design, Specification and Verification of Interactive Systems, DSV-IS 2000, Lecture Notes in Computer Science, vol. 1946, Springer, Berlin, Germany, 2001, pp. 191–205.

[32] M. Elkoutbi, R.K. Keller, User interface prototyping based on UML scenarios and high-level Petri nets, in: Proceedings of 21st International Conference on Application and Theory of Petri Nets, ICATPN 2000, Lecture Notes in Computer Science, vol. 1825, Springer, Berlin, Germany, 2000, pp. 166–186.

[33] M. Lozano, I. Ramos, P. González, User interface specification and development, in: Proceedings of the IEEE 34th International Conference on Technology of Object-Oriented Languages and Systems, IEEE Computer Society Press, Washington, DC, USA, 2000, pp. 373–381.

[34] P.F. Campos, N.J. Nunes, A UML-based tool for designing user interfaces, in: UML 2004—The Unified Modelling Language, Modelling Languages and Applications Seventh International Conference Satellite Activities, Lecture Notes in Computer Science, vol. 3297, Springer, Berlin, Germany, 2005, pp. 273–276.

[35] N.J. Nunes, Representing user-interface patterns in UML, in: Proceedings of the International Conference on Object Oriented Information Systems, OOIS' 2003, Lecture Notes in Computer Science, vol. 2817, Springer, Berlin, Germany, 2003, pp. 142–151.

[36] D.J. Anderson, Extending UML for UI, Technical Report, Proceedings of the Towards a UML Profile for Interactive Systems Development, TUPIS'00, 2000. URL: ⟨http://www.uidesign.net/2000/papers/TUPISproposal.html⟩.

[37] J. Conallen, Modeling Web application architectures with UML, Communications of the ACM 42 (10) (1999) 63–70.

[38] P. Pinheiro da Silva, N.W. Paton, User interface modeling in UMLi, IEEE Software 20 (4) (2003) 62–69.

[39] I. Horrocks, Constructing the User Interface with Statecharts, Addison-Wesley, Reading, MA, 1990.

[40] B. Lieberman, UML activity diagrams: detailing user interface navigation, Technical Report ⟨http://www.ibm.com/developerworks/rational/library/content/RationalEdge/oct01/UMLActivityDiagramsOct01.pdf⟩, 2001.

[41] K. Leung, L.C.K. Hui, S. Yiu, R. Tang, Modeling Web navigation by statechart, in: Proceedings of the 24th Annual International Computer Software and Applications Conference, COMPSAC'2000, IEEE Computer Society Press, Washington, DC, USA, 2000, pp. 41–47.

[42] M. Winckler, P. Palanque, StateWebCharts: a formal description technique dedicated to navigation modelling of Web applications, in: Proceedings of the 10th International Workshop on Interactive Systems. Design, Specification and Verification, DSV-IS 2003, Lecture Notes in Computer Science, vol. 2844, Springer, Berlin, Germany, 2003, pp. 61–76.

[43] J. Vanderdonckt, F. Bodart, Encapsulating knowledge for intelligent automatic interaction objects selection, in: Proceedings of the Conference on Human Factors in Computing Systems, INTERCHI'93, ACM Press, NY, USA, 1993, pp. 424–429.

[44] S. Kovacevic, UML and user interface modeling, in: Proceedings of the Unified Modeling Language Conference, UML'98, Lecture Notes in Computer Science, vol. 1618, Springer, Berlin, Germany, 1998, pp. 253–266.

[45] S. Ceri, P. Fraternali, A. Bongio, Web Modeling Language (WebML): a modeling language for designing Web sites, in: Proceedings of the Ninth International World Wide Web Conference on Computer Networks, North-Holland, Amsterdam, The Netherlands, 2000, pp. 137–157.

[46] N. Koch, Software engineering for adaptive hypermedia systems: reference model, modelling techniques and development process, Ph.D. Thesis, Ludwig-Maximilians Universität Munchen, 2001.

[47] A. Puerta, Supporting user-centered design of adaptive user interfaces via interface models, in: Proceedings of the First Annual Workshop On Real-Time Intelligent User Interfaces for Decision Support and Information Visualization, San Francisco, 1998, p.11.

[48] M. Rettig, Prototyping for tiny fingers, Communications of the ACM 37 (4) (1994) 21–27.

[49] SEGUIA, System Expert Generating User Interfaces Automatically ⟨http://www.isys.ucl.ac.be/bchi/research/seguia.htm⟩.

[50] A. Puerta, J. Eisenstein, XIML: a common representation for interaction data, in: Proceedings of the Seventh

International Conference on Intelligent User Interfaces, IUI '02, ACM Press, NY, USA, 2002, pp. 214–215.

[51] M. Abrams, C. Phanouriou, A.L. Batongbacal, S.M. Williams, J.E. Shuster, UIML: an appliance-independent XML user interface language, Computer Networks 31 (11–16) (1999) 1695–1708.

[52] A. Stanciulescu, Q. Limbourg, J. Vanderdonckt, B. Michotte, F. Montero, A transformational approach for multimodal Web user interfaces based on UsiXML, in: Proceedings of the Seventh International Conference on Multimodal Interfaces, ICMI '05, ACM Press, NY, USA, 2005, pp. 259–266.

[53] S. Berti, F. Correani, G. Mori, F. Paternó, C. Santoro, TERESA: a transformation-based environment for designing and developing multi-device interfaces, in: Proceedings of ACM CHI 2004 Conference on Human Factors in Computing Systems, vol. II, ACM Press, NY, USA, 2004, pp. 793–794.

[54] P. Pinheiro da Silva, N.W. Paton, User interface modelling with UML, in: Information Modelling and Knowledge Bases XII, IOS Press, 2000, pp. 203–217.

[55] N.W. Paton, P. Pinheiro da Silva, ARGOi, an object-oriented design tool based on UML, Technical Report ⟨http://trust.utep.edu/umli/software.html⟩, 2007.

[56] J. Molina, P. González, M. Lozano, Developing 3D UIs using the IDEAS tool: a case study, Human–Computer Interaction. Theory and Practice, Lawrence Erlbaum Associates, 2003.

[57] O. Pastor, F. Hayes, S. Bear, OASIS: an object-oriented specification language, in: Proceedings of Advanced Information Systems Engineering, CAiSE'92, Lecture Notes in Computer Science, vol. 593, Springer, Berlin, Germany, 1992, pp. 348–363.

[58] L.L. Constantine, L.A.D. Lockwood, Structure and style in use cases for user interface design, in: M. van Harmelen (Ed.), Object Modelling and User Interface Design, Addison-Wesley, 2001. Reading, MA.

[59] UIDesign, UIRupture, Technical Report ⟨http://www.uidesign.net/2000/opinion/UIRupture.html⟩, 2000.

[60] J. Heumann, User experience storyboards: building better UIs with RUP, UML and use cases, Technical Report, ⟨http://www.ibm.com/developerworks/rational/library/content/RationalEdge/nov03/f_usability_jh.pdf⟩, 2003.

[61] F. Paternò, Towards a UML for interactive systems, in: Proceedings of the Eighth IFIP Working Conference on Engineering for Human–Computer Interaction, EHCI 2001, Lecture Notes in Computer Science, vol. 2254, Springer, Berlin, Germany, 2001, pp. 7–18.

[62] J.C. Campos, M.D. Harrison, Formally verifying interactive systems: a review, in: Proceedings of the International Workshop on Design, Specification and Verification of Interactive Systems, DSV-IS'97, Springer, Berlin, Germany, 1997, pp. 109–124.

[63] G. Doherty, J.C. Campos, M. Harrison, Representational reasoning and verifications, in: Proceedings of the Formal Aspects of the Human Computer Interaction, BCS-FACS Workshop, SHU Press, 1998, pp. 193–212.

[64] J. Fekete, M. Richard, P. Dragicevic, Specifications and verifications of interactor: a tour of esterel, in: Proceedings of the Formal Aspects of the Human Computer Interaction, BCS-FACS Workshop, SHU Press, 1998, pp. 103–119.

[65] P.A. Palanque, R. Bastide, in: Proceedings of the International Workshop on Design Specification and Verification of Interactive Systems, DSV-IS'95, Springer, Berlin, Germany, 1995.

[66] P.A. Palanque, R. Bastide, V. Sengès, Validating interactive system design through the verification of formal task and system models, in: Proceedings of the IFIP Working Conference on Engineering for Human–Computer Interaction, EHCI'95, Chapman & Hall, London, 1995, pp. 189–212.

[67] F. Paternò, M. Mezzanotte, Formal analysis of user and system interactions in the CERD case study, in: Proceedings of the IFIP Working Conference on Engineering for Human–Computer Interaction, EHCI'95, Chapman & Hall, London, 1995, pp. 213–226.