Designing GUI components from UML Use Cases*

Jesús M. Almendros-Jiménez and Luis Iribarne
Dpto. de Lenguajes y Computación. Universidad de Almería, Spain.
{jalmen,liribarne}@ual.es

Abstract

In this paper we present how to develop graphical user interfaces from two UML models: use case and activity diagrams. Our method obtains from them a UML class diagram for representing GUI components, and it is suitable for generating code fragments which can be considered as GUI prototypes.

1. Introduction

Currently, the integrated development environments (IDE) represent a good practice of model-driven development (MDD) since they offer tools that raise the abstraction level for creating applications, such as language editors, form builders, and GUI controls. Modeling allows the developers to visualize source code in a graphical form: graphical abstractions such as flow charts to depict algorithmic control flows and structure charts or simple block diagrams with boxes representing functions and subprograms, and so on. MDD also involves creating models through a methodological process that begins with requirements and delves into high-level architectural design.

In the *Unified Modeling Language* (UML), one of the key tools for behaviour modeling is the *Use Case* model, originated from the *Object-Oriented Software Engineering* (OOSE). The key concepts associated with the use case model are *actors* and *use cases*. The users and any other systems that may interact with the system are represented as actors. The required behaviour of the system is specified by one or more uses cases, which are defined according to the needs of the actors. Each use case specifies some behaviour, possibly including variants, that the system can perform in collaboration with one or more actors.

Graphical User Interfaces (GUI) have become increasingly dominant, and the design of the "external" system has assumed increasing importance. The user interface, as a significant part of most applications, should also be modeled using UML. However, it is by no means always clear how to model user interfaces using UML, although there are recent approaches [5, 10, 9, 1, 2, 7] which have addressed this problem.

* This work has been partially supported by the Spanish project of the Ministry of Science and Technology "INDALOG" TIC2002-03968 under FEDER funds.

2. GUI model-driven development

In this paper, we focus on the design of *GUI with the UML Use case model*. The design of GUI is based on Use case model, and conversely, the design of uses cases is oriented to GUI design.

Use case model is intended to be used in early stages of the system analysis in order to specify the system functionality, as an external view of the system. However, use cases can be *specified* by means of activity diagrams, which provide a finer granularity and more rigorous semantics. Activity diagrams can be used for specifying user-system interaction, that is, states can represent outputs to the user which are labeled with UML *stereotypes* representing visual components for data output. In addition, transitions can represent user inputs which are labeled with UML stereotypes representing visual components for data input and choices.

Furthermore, the refinement of uses cases by means of activity diagrams achieves more precise specifications, enabling to detect $<<include>>$ and *generalization* relationships between use cases [12, 8]. These relations have an unstable semantics along the UML development, and have received several interpretations, reflecting a high degree of confusion among developers [11, 3]. Therefore, our proposal helps to clarify the cited relations. The GUI design reflects the relationships between use cases, by using the *applet* or *frame* inheritance as an implementation of use cases generalization, and the applet invocation and embedding as an implementation of the $<<include>>$ relation.

Our method handles use cases and activity diagrams and, following some rules of transformation, transforms both specifications into the user interface. The designer is the responsible for both specifications and the GUI are designed according to the specification. Use case diagrams can now be viewed as a high-level specification of each use case description, given by activity diagrams and, therefore, as a high-level specification of the *presentation logic* of the system.

In the literature there are some works which accomplish the design of GUI in UML. The closest to our approach are [10, 9, 7]. These proposals identify some aspects of GUI that cannot be modeled using UML notation, and a set of UML constructors that may be used to model GUI. However, a method for GUI design using the use case model is not completely addressed, and there also exists a lack of formal description of use cases

and GUI components.

Another similar work to our contribution is [1, 2] in which state machines and petri nets are used to specify GUI in UML. In the quoted approaches they specify user interaction but they also lack of use case relationships handling. Finally, in [6, 4], uses cases are mapped into a UML class diagram to represent the data logic, but not to design GUI.

The rest of the paper is organized as follows. Section 3 describes the rules of a method that the designer should follow to build GUI components, using use cases, class and activity diagrams. Section 4 presents a GUI project example of an Internet book shopping that illustrates the use of the design rules. Then, Section 5 describes a formalism that helps us to validate the proposed method. Finally, Section 6 discusses some conclusions and future work.

3. A GUI modeling method (UML-GMM)

In our method a use case diagram consists of a set of actors (users and external systems) and use cases. Relationships between actors are generalizations, and relationships between uses cases are $<<$*include*$>>$ dependences, together with generalizations. In addition, relationships between actors and use cases are simple associations. Roles, multiplicity, directionality, and *extend* dependences will not be considered in our approach yet.

An activity diagram consists of a set of states, with two special cases: the initial and the final state. States can be linked with labeled transitions (arrows), and a transition can have several branches with a diamond representing the branching point.

3.1. Steps for applying the method

Now, we present the steps identifying a GUI project development:

(a) Firstly, an informal high level description of the system is carried out by means of a use case diagram. The use case diagram involves the actors and the main use cases.

(b) Secondly, for each use case its behaviour is described by means of one or more activity diagrams, fulfilling those restrictions of the methodology. The original use case diagram goes refining for obtaining a more formal diagram.

(c) Thirdly, the use cases and activity diagrams are translated into a class diagram.

(d) Finally, the class diagram obtained in the previous step produces code fragments which could be considered as GUI component prototypes.

Certainly, this development sequence is cyclic since the designer can refine high-level details of the use case diagram in the next phases.

3.2. Rules for a GUI design

In the description of our method, we have chosen Java as the programming language for GUI coding due Java *swing* for GUI, however our approach could be adapted for other kinds of user interface software. We assume the reader knows the basic behaviour of Java swing classes, however we would like to remark some concepts used in our method. Firstly, we consider two kind of window interfaces: *applet* and *frame*. A frame can include in the window area GUI components such as buttons, labels, text areas, and so on, and can have embedded (or invoke) one or more applets or frames. An applet can only contain in the window area GUI components. Therefore frames will be used for the building of complex user interfaces where several tasks can be done (some of these GUI components could be disabled according with the executed task). We use inheritance for frames and applets assuming that it means inheritance of behaviour but not necessarily of appearance.

We can summarize the rules of design as follows:

r1. Each *actor* representing a user in the use case diagram is an *applet* or *frame*. Actors representing external systems are not considered for visual component design. The choice of applet or frame depends on whether the actor can perform one or more tasks, that is, depends on the number of associations with use cases.

r2. 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.

r3. Each *use case* in the use case diagram is an *applet or frame*. It is embedded into the frame of the associated actors.

r4. 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. The $<<$*include*$>>$ *relationship* between two use cases u and w (u includes w) corresponds with the *invocation or embedding* from the applet or frame of u of the (sub)applet or frame of w.

r6. Each *state* of the activity diagram necessarily falls in one of the two following categories: *terminal states* or *non-terminal states*. A terminal state is labeled with a *UML stereotype* representing an *output GUI component* (stereotyped states). A non-terminal state is not labeled, and it is described by means of any activity diagram. The non-terminal states can be "use cases" of the diagram or not.

r7. Each *transition* in the activity diagrams can be labeled by means of *conditions* or *UML stereotypes with conditions*. The UML stereotypes represent *input GUI components*. This kind of transitions is also called *stereotyped transitions*. The conditions represent *use choices or business logic*.

r8. In the case of the non-terminal states, the use case diagram can specify $<<$*include*$>>$ or generalization relationships between the non-terminal state and the use case, and we follow these rules: (a)

terminal state is also an applet or frame. It contains the GUI components in the associated activity diagram; (b) In the generalization relationship case, the non-terminal state is also an applet or frame containing the GUI components in the associated activity diagram, but the use case also contains these GUI components.

r9. A non-terminal state, which does not appear in the use case diagram, is neither an applet nor a frame, and the GUI components in the associated activity diagram are GUI components of the applet or frame of the use case.

r10. The conditions of the transitions are not taken into account for the GUI design.

With regard to the use case relationships, they are interpreted as follows:

r11. The $<<include>>$ relationship between a use case u and a use case w (u includes w) means that *one* of the non-terminal states of the activity diagram of u is w.

r12. The generalization relationship between a use case u and a use case w (u generalizes w) means that the activity diagram representing w contains the states and transitions of the activity diagram of u, but some states or transitions s of u can be *replaced* in w by states (resp. transitions) s' following a *replacement relationship* $s' \sqsubseteq s$. In addition, w can add new states and transitions starting from (and reaching) the particular case of the state u.

In practice, this replacement relation should be decided by the designer. Basically, stereotyped states can be replaced if the *output GUI component* can be replaced. For instance, a list with two columns can be replaced by a list with three columns without lost of functionality. The same happens with stereotyped interactions which can be replaced if the *input GUI components* can be replaced. For instance, a selection of any of the cited list. Finally, conditions can be, for instance, replaced if one of them is *more restrictive* than the other.

Let remark that inclusion is a particular case of generalization, that is, if u includes v then v generalizes u. However, we handle the generalization by considering two applets, one for each use case u and v, but u does not invoke v, rather than u includes the behaviour of v.

4. A GUI modeling example

To illustrate the functionality of the UML-based GUI modeling method, we will explain a simple Internet book shopping (IBS) model. In the IBS there basically appear three actors: the customer, the ordering manager, and the administrator. A customer directly carries out the purchases by the Internet, querying certain issues of the product in a catalogue of books before carrying out the purchase. The manager deals with (total or partially) customer's orders. And finally, the system's administrator can manage the catalogue of books adding and eliminating books in the catalogue or modifying those already existing. Considering this scenario, in the next tinued to develop a GUI project using uses cases.

4.1. Step 1: Describing use cases

Initially, the use case diagram contains the identified actors of the system. In our case study, the actors are the `Customer`, the `Manager` and the `Administrator`. The non-formal definition of the system will be refined, causing more precise use cases diagram(s). Figure 1 shows the complete presentation logic definition for the Internet Shopping system.

`Manage catalogue` is an applet that directly depends on four use cases, connected to them by means of an $<<include>>$ relation. The include relationships between the use cases `Withdraw article`, `Modify article` and `Add article` were modeled by the system's designer as relations of optionality (the branches of the use case `Manage Catalogue`'s behaviour go to these states in the activity diagram). The use case `Administrator identification` was considered by the system's designer as a relation of mandatory (this state is always reached in the activity diagram) of the use case `Manage Catalogue`. The use case `Manage catalogue` is composed of the use cases `Withdraw article`, `Modify article` and `Add article` (i.e., applets or frames). Both the manager and the administrator should be identified themselves before carrying out any kind of operation restricted to his/her environment of work.

The relation of *generalization* is intended as an inheritance of behaviour and, therefore, of GUI components. For example, the `Query catalogue` use case has been established as a generalization of the `Purchase` use case. That means that the purchasing applet also allows a query operation on the catalogue. In fact, the applet of purchasing inherits from query catalogue. Note how the use case `Query catalogue by administrator` also inherits from query catalogue, and it generalizes the use cases `Withdraw article`, and `Modify article`.

The distinction between *include* and generalization relationships is established by the system's designer into the activity diagrams of those include-connected use cases. In next sections we will only focus on the `Purchase` use case to explain our method.

4.2. Step 2: Describing activity diagrams

Each use case will correspond with a Java applet (or frame) component in the method. Activity diagrams describe certain graphical and behavioural details about the graphical components of an applet. In our case study, we have only adopted four Java graphical components: `JTextArea`, `JList`, `JLabel` and `JButton`. Nevertheless, other graphical elements could be easily considered in the activity diagram since they are modeled as state or transition stereotypes.

Graphical components can be classified as input (a text area or a button) and output components (a label or list). Input/output components are associated

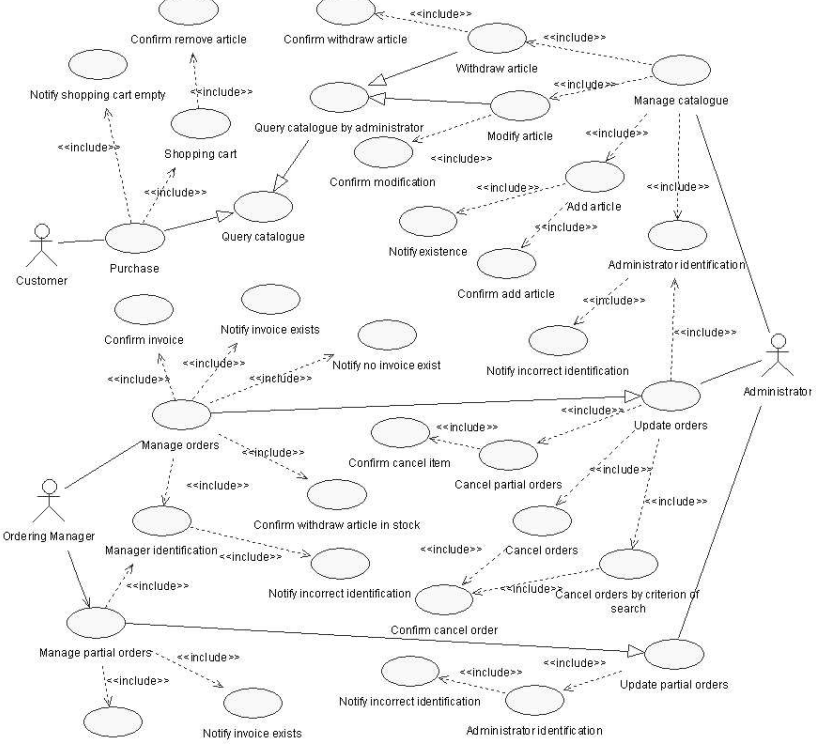


Figure 1. The Internet Shopping use case diagram

with terminal states and transitions by using the appropriate stereotype, for instance, `JTextArea`, `JList`, `JLabel` stereotypes are associated with states and `JButton` stereotype to transitions. Since the graphical behaviour concerns to states and transitions, next we will describe them separately.

States can be stereotyped or not. Stereotyped states represent terminal states which can be labeled by the `<<JTextArea>>`, `<<JList>>` and `<<JLabel>>` stereotypes. For instance, Figure 2 (a) shows the activity diagram for the `Purchase` use case. The behaviour shows how the customer begins the purchasing process of querying, adding or removing articles of the shopping cart. After a usual purchasing process, the shopping system requests the customer a card number and a postal address to carry out the shipment, whenever the shopping cart is not empty. This diagram shows the graphical and behavioural content of the applet window where the purchases can be carried out. The activity diagram is composed of four states. Two of them are terminal states, since they correspond to graphical elements. They are stereotyped (`<<JTextArea>>`) and labeled by a text related to the graphical element. Two other states have been described in a separate activity diagram in order to structure better the design. The name of a separate activity diagram should be the same as the one of the state.

According to the rules of the proposed methodology, if an state is not labeled with a stereotype, this means that the state is described in another activity diagram. This new diagram can either represent the behaviour of another use case or simply a way of allowing a hierarchical decomposition of the original activity diagram. For example, in the activity diagram associated with the `Purchase` use case, there appear two non-terminal states: `Manage shopping cart` and `Notify shopping cart empty`. At the same time, two activity diagrams are described for both states. All these activity diagrams are also shown in Figure 2.

In the activity diagram of the `Notify shopping cart empty` use case, we can observe how the target use case (being modeled) brings to another activity diagram. The model represents a warning applet window containing only the text `Shopping cart empty` and the button `Close`. In the `Manage shopping cart` activity diagram, the states `Query catalogue` and `Shopping cart` are itemized on independent activity diagrams. Both states would also correspond with an applet, since they appear as use cases in the diagram.

Transitions can be labeled by means of *stereotypes*, *conditions* or both together. For instance, a button is connected to a transition by using the `<<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 graphical component (related to button or list states), and the second one is an internal checking condition (not related to the states, but

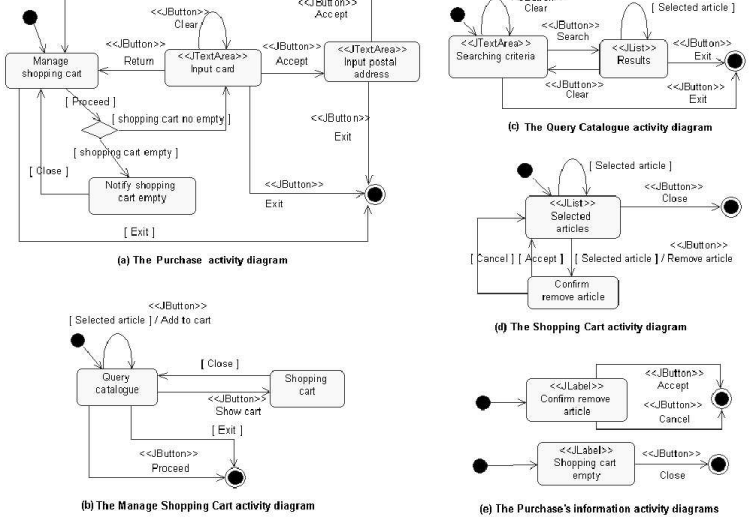


Figure 2. The whole activity diagram of the `Purchase` use case

to the internal process). For example, in our case study the selections on a list are modeled by conditions. Note in the Query Catalogue activity diagram how the list `Results` is modeled by a `<<JList>>` state and a `[Selected article]` condition. Figure 2 shows some transitions (p.e., `[Close]`, `[Exit]` or `[Proceed]`) that correspond with conditions of the kind *user choice*. The `[Exit]` output transition of the state `Manage shopping cart` means that the user has pressed a button called `Exit`, which has been defined in a separate `Manage shopping cart` activity diagram. Nevertheless, the `[shopping cart no empty]` and `[shopping cart empty]` conditions are two *business/data logic* conditions, in which the human factor does not participate.

Condition/action transitions are also useful to model the behaviour of generalization relationships between use cases. Note in the original use case diagram how the `Purchase` use case inherits the behaviour of the use case `Query catalogue` by means of a generalization relationship. This inheritance behaviour is modeled in the Purchase activity diagram as a non-terminal state that includes the behaviour of the Query Catalogue activity diagram. For example, let us observe the behaviour of the query catalogue shown in Figure 2 (c). In this activity diagram, the user introduces the searching criteria in the text area, presses the button `Search` and then the results are shown on a list. After that, the user can select articles in the list, presses a button to exit or try a new search by pressing the button `Clear`. When the Purchase use case inherits the Query Catalogue, it should be possible to interrupt its behaviour. Condition/action transitions can be used to interrupt an inherited behaviour. For example, the query catalogue's behaviour (previously described) is adopted in the activity diagram of the Purchase use case as a non-terminal state called `Query catalogue`. The output transition `[Selected article]/Add to cart` means that the `Add to cart` button at the Purchase applet (use case) can interrupt the query catalogue behaviour whether an article has been selected (condition). Analogously, the output transitions `Proceed` and `Show cart` mean that both buttons can interrupt the inherited behaviour of the query catalogue.

On the other hand, a generalization relationship does not only represent an inheritance of the behaviour as an extension; for instance, the Purchase use case inherits the Query Catalogue use case and increases its behaviour to hold the buttons `Add to cart`, `Show cart` and `Proceed`. However, a generalization relationship can also deal with a **replacement** of behaviour instead of an increase in behaviour. For example, note in the original use case diagram how the Query Catalogue by Administrator also inherits the Query Catalogue. Let us suppose that their behaviours (activity diagrams) are the same, but the results list shown to the customer actor (the `Results` state) is different from that shown to the administrator actor (for instance, `Administrator Results` state). In this case, the system's designer can use the behaviour (activity diagram) of the Query Catalogue use case to model the behaviour (activity diagram) of the "Query Catalogue by Administrator" re-writing (replacing) the results list (p.e., replacing `Results` by `Administrator Results`). This rule of replacement can also be considered on transitions (p.e, replacing a button by another GUI component). Finally, the conditions and "conditions/actions" can be also replaced. In all cases, is a decision of the designer to allow the replacement of states and transitions.

4.3. Step 3: Generating class diagrams

Use cases are translated into classes with the same name as these use cases. The translated classes specialize in a Java *Applet* class. The components of the applet (use case) are described in activity diagrams. A terminal state is translated into that Java *swing* class represented by the stereotype of the state. The Java *swing* class is connected from the container class (i.e., that class working as an applet window in the use case diagram) and

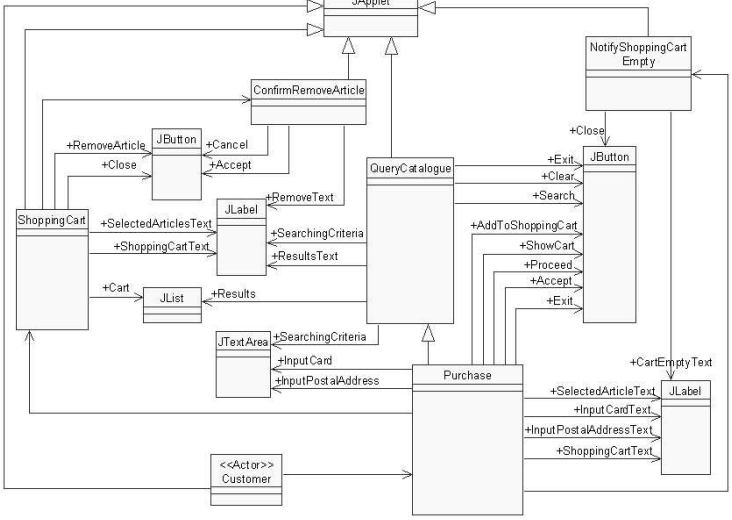


Figure 3. A class diagram obtained from the use cases and activity diagrams

uses an association relationship whose role's name is the one on the terminal state. For example, those terminal states stereotyped as `<<JTextArea>>` are translated into a *JTextArea* class in the class diagram. Something similar happens to the rest of stereotyped states and transitions. The non-terminal states of an activity diagram may correspond to some other use cases (applets) or activity subdiagrams. In the last case, the non-terminal states can be considered an abstract class in the class diagram. Then, it can be described in another class diagram with the same name as that abstract class. Figure 3 shows the class diagram of the customer side.

The class diagram contains six applets, four of which directly specialize in the *Applet* class: the `NotifyShoppingCartEmpty` class, the `Customer` class, the `ConfirmRemoveArticle` class and the `ShoppingCart` class. The other classes inherit the *Applet* class through their super classes. For example, the `Purchase` class inherits the applet class from the `QueryCatalogue` class. These six classes correspond to those five use cases at the customer side in the use case diagram together with the customer actor.

Furthermore, note how the stereotyped states and transitions in the activity diagrams are translated into Java classes in the class diagram. The stereotype name of a transition or state is translated into the appropriate *Java swing* class. The name of the stereotyped state (transition) is translated into an association between the *swing* class and the applet class that contains it. For example, the `<<JButton>>` stereotype of the `Proceed` transition that appears in the `Manage shopping cart` activity diagram (see Figure 2) is translated into a `JButton` class. The transition name (`Proceed`) is interpreted as an association —labeled with the same name— between the `JButton` class and the class containing it (i.e., the `Purchase`). Due to the extension of the resultant class diagram, some classes have not been included in the figure.

4.4. Step 4: Generating the GUI components

Finally, rapid GUI prototypes could be obtained from the class diagram. Figure 4 shows a first visual result of the `Purchase` applet, but without functionality.

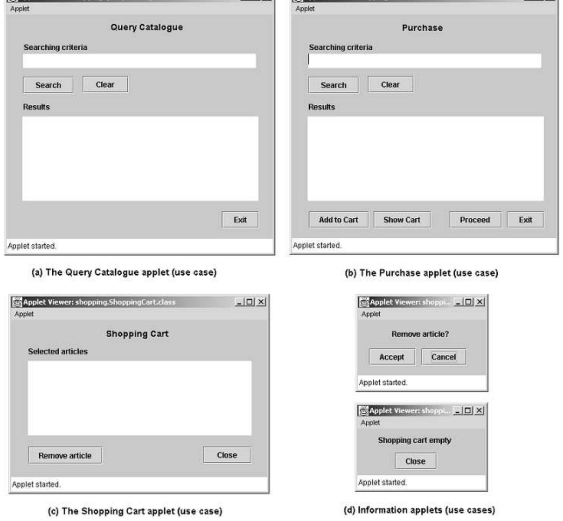
Note how the Purchase window (applet) is very similar to the Query Catalogue window, except that the second one includes three buttons more than the first window. This similarity between applets was reflected in the original use case diagram as a generalization relationship between use cases (applets), here, between the use cases `Query catalogue` and `Purchase`.

The Shopping Cart window (Figure 4, c) appears when the `Show Cart` button is pressed on the Purchase window (Figure 4, b). Note in the original use case diagram, shown in Figure 1, how the button is associated with the window by means of an $<< include >>$ relation between use cases. On the other hand, the two information applet windows (Figure 4, d) are also associated with two buttons: the `Remove article` button in the Shopping Cart window and the `Proceed` button in the Purchase window. Note again how these windows are also described as *include* relations between use cases. Also observe the activity diagrams shown in Figure 2 to track better the behaviour of the example.

For space reasons, we have included here just a part of the GUI project developed for the case study. A complete version of the project is available at `http://www.ual.es/~liribarn/Investigacion/usecases.html`.

5. Formalizing UML-GMM

In this section we will formalize the described method, and provide a formal definition for use case diagrams and use cases. In particular, we will define the use case relationships $<<include>>$ and generalization. We will also define well-formed use case diagram which follows some restrictions. In addition, we will provide an abstract definition of GUI,



(a) The Query Catalogue applet (use case)

(b) The Purchase applet (use case)

(c) The Shopping Cart applet (use case)

(d) Information applets (use cases)

Figure 4. The applet windows of the Customer side

and we will define two relationships between GUI: inclusion and generalization. This will allow us to define a generic transformation technique for use case diagrams into a set of GUI. Finally, we will establish some properties of this transformation technique. Now, let us define a use case diagram as follows:

Definition 1 (Use Case Diagram) *A use case diagram* $UCD = (n, ACT, UC, \rightarrow\!\triangleright, —, \overset{<<i>>}{\dashrightarrow})$ *consists of a diagram name* n*; a finite set* ACT *of actor's names which can be users and external systems* $p, q, r, \ldots$*; a finite set* UC *of use cases* $u, v, w, \ldots$*; and three relations* $\rightarrow\!\triangleright$*,* $—$ *and* $\overset{<<i>>}{\dashrightarrow}$*, where* $\rightarrow\!\triangleright \subseteq (ACT \times ACT) \cup (UC \times UC)$*;* $— \subseteq ACT \times UC$*; and* $\overset{<<i>>}{\dashrightarrow} \subseteq UC \times UC$*; as usual we write* $p \rightarrow\!\triangleright q$*, rather than* $(p, q) \in \rightarrow\!\triangleright$*, and analogously for* $—$ *and* $\overset{<<i>>}{\dashrightarrow}$*.*

Now, we formally define a use case being specified by means of an activity diagram as follows:

Definition 2 (Use Case) *A use case* $u = (n, S, SI, IN, OUT, COND, \rightarrow)$ *consists of: a use case name* n*; a finite set* S *of states which consist of: a finite set* UC *of use cases; a finite set* SS *of stereotyped states of the form* (sn, p) *where* sn *is an state name and* $p \in OUT$*; three special states* SP*, the initial, end and branching states; a finite set* SI *of stereotyped interactions of the form* $[C]/(in, i)$ *where* $C \in COND$*,* in *is an interaction name, and* $i \in IN$*. The condition* $[C]$ *is optional; a finite set* IN *of input stereotypes* $i, j, \ldots$*; a finite set* OUT *of output stereotypes* $p, q, \ldots$*; a finite set* $COND$ *of conditions* $C, D \ldots$*; a transition relation* $\rightarrow \subseteq S \times (SI \cup COND) \times S$*. As usual we write* $A \xrightarrow{\lambda} B$ *rather than* $(A, \lambda, B) \in \rightarrow$*, where* λ *can be* $[C]$ *or* $[C]/(in, i)$*.*

We denote by $name(u)$ the name of a use case u. Analogously, we define the functions $usecases(u)$ and $transitions(u)$, to get the use cases, respectively, the transitions of a use case. $SI(u)$ —resp. $SS(u)$— denotes the set of stereotyped interactions (in, i) in u —resp. stereotyped states (sn, p) in u—. Finally, we call *exit conditions*, denoted by $exit(u)$, to the interaction names in of transitions $s \overset{[C]/(in,i)}{\rightarrow} s'$ and conditions C of transitions $\overset{[C]}{\rightarrow}$ which go to the end state in a use case.

Now, we have to assume a *(reflexive) replacement relation* $\sqsubseteq$ between stereotyped states, and the same relation for stereotyped interactions and conditions.

The replacement relation $\sqsubseteq$ can be extended to use cases, as follows. Given two 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 $\lambda' \sqsubseteq \lambda$. Assuming this, we can define the inclusion and generalization relationship between use cases as follows. Given two use cases u, v we say that u includes v if $v \in usecases(u)$, and we say that u generalizes v, if there exists $w \in usecases(v)$ such that $w \sqsubseteq u$. Now, we will define the well-formed use cases.

Definition 3 (Well-formed Use Case) *A use case* u *is well-formed if the following conditions hold: for all* $s \xrightarrow{\lambda} t \in transitions(u)$ *then* λ *has the form* $[C]/(in, i) \in SI$ *iff* $s = (sn, p)$*,* $p \in OUT$*, or* $s \in usecases(u)$ *and* s *generalizes* u*; for all* $v \in usecases(u)$ *then there exists* $s \xrightarrow{\lambda} t \in transitions(u)$ *such that* λ *has the form* $[C]$ *or* $[C]/(in, p)$ *for every* $C \in exit(v)$*.*

Well-formed use cases take into account that: (1) an output component should trigger an input interaction; (2) input interactions can be added to a more general use case in order to obtain more particular ones; and finally, (3) the exit conditions of a non-terminal state should be included in the main use case.

According to the previous definition, a well-formed use case diagram includes well-formed use cases, and the <<*include*>> and generalization relationships between use cases in the use case diagram correspond with

Definition 4 (Well-formed Use Case Diagram) *A well-formed Use Case Diagram* $UCD = (n, ACT, UC, \multimap, —, \overset{<<i>>}{\dashrightarrow})$ *satisfies that every* $u \in UC$ *is well-formed; for all* $u, u' \in UC$, $u' \multimap u$ *if* u *generalizes* u'*; and for all* $u, u' \in UC$, $u \overset{<<i>>}{\dashrightarrow} u'$ *if* u *includes* u'.

Now, we will provide an abstract definition of GUI and GUI components. A GUI has a name, a set of GUI which can be invoked (embedded) from it, and a set of stereotyped interactions and states which represent the input and output GUI components.

Definition 5 (GUI) *A graphical user interface (GUI)* $G = (n, W, I, O)$ *consists of a GUI name* n*; a finite set* W *of GUIs; a finite set* I *of stereotyped interactions* (in, i)*; and a finite set* O *of stereotyped states* (sn, p).

GUI can be compared by means of generalization and inclusion relationships. The first one corresponds with the inheritance relationship, and the second one with the invocation (embedding) of GUI.

Given two GUI $(n, W, I, O), (n', W', I', O')$ we say that (n, W, I, O) generalizes (n', W', I', O') if for all $G \in W$, there exists $G' \in W'$ such that G generalizes G'; for all $i \in I$, there exists $i' \in I'$ such that $i' \sqsubseteq i$; and for all $o \in O$, there exists $o' \in O'$ such that $o' \sqsubseteq o$. Given two GUI G and G', we say that $G = (n, W, I, O)$ includes G' if $G' \in W$.

Now, we can formally define our transformation technique which provides a set of GUI for each use case diagram. In order to define our transformation, we need to suppose that <<option>> is a stereotype representing each menu option of a GUI.

Definition 6 (GUI of a Use Case Diagram) *Given a well-formed use case diagram* $UCD = (n, ACT, UC, \multimap, —, \overset{<<i>>}{\dashrightarrow})$*, we define the GUI associated with* UCD*, denoted by* $GUI(UCD)$*, as the set* $\{GUI(p) \mid p \in ACT \; is \; a \; user\}$*, where:*

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

$$where \begin{cases} W = \{GUI(u) \mid p \; — \; u\} \\ I = \{(name(u), << option >>) \mid p \; — \; u\} \\ O = \varnothing \end{cases}$$

$$and$$

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

$$where \begin{cases} W = \{GUI(v) \mid u \overset{<<i>>}{\dashrightarrow} v\} \\ I = SI(u) \bigcup_{\substack{v \in usecases(u) \\ and \; not \; u \overset{<<i>>}{\dashrightarrow} v}} SI(v) \\ O = SS(u) \bigcup_{\substack{v \in usecases(u) \\ and \; not \; u \overset{<<i>>}{\dashrightarrow} v}} SS(v) \end{cases}$$

We can state the following result from our transformation technique.

Theorem 1 *The GUI associated to a well-formed* UCD *satisfies the following conditions: (a) for* $p \in ACT$*,* $GUI(p)$ *includes* $GUI(p')$*; (b) for all* $u, u' \in UC$, $u' \multimap u$ *then* $GUI(u)$ *generalizes* $GUI(u')$*; (c) for all* $u, u' \in UC$, $u \overset{<<i>>}{\dashrightarrow} u'$ *then* $GUI(u)$ *includes* $GUI(u')$*; (d) for all* $p \in ACT$ *and* $u \in UC$, $p —u$*. then* $GUI(p)$ *includes* $GUI(u)$

6. Conclusions and Future Work

In this paper, we have studied a method for mapping use case and activity diagram models into graphical user interfaces (GUI). Through a case study, we have shown how our technique can be applied to the design of the Internet Book Shopping system. As a future work, we firstly plan to extend our work to deal with the <<extends>> relationship of use cases. Secondly, we would like to incorporate our methodology in a CASE tool in order to automatize it. And finally, we would like to integrate our technique in the whole development process.

References

[1] M. Elkoutbi and R. K. Keller. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In *Procs of ICATPN'2000*, pages 166–186. LNCS 1825, 2000.

[2] M. Elkoutbi, I. Khriss, and R. K. Keller. Generating user interface prototypes from scenarios. In *Procs of RE '99*, page 150. IEEE CS, 1999.

[3] G. G´enova, J. Llor´ens, and V´ıctor Quintana. Digging into use case relationships. In *Procs of UML'2002*, pages 115–127. LNCS 2460, 2002.

[4] G. Kösters, H. W. Six, and M. Winter. Coupling Use Cases and Class Models as a Means for Validation and Verifi cation of Requirements Specifi cations. *Requirements Engineering*, 6(1):3–17, 2001.

[5] S. Kovacevic. UML and User Interface Modeling. In *Procs of UML'98*, pages 253–266. LNCS 1618, 1998.

[6] Y. Liang. From use cases to classes: a way of building object model with UML. *Information & Software Technology*, 45(2):83–93, 2003.

[7] N. J. Nunes. Representing User-Interface Patterns in UML. In *Procs of OOIS 2003*, pages 142–151. LNCS 2817, 2003.

[8] G. Övergaard and K. Palmkvist. A Formal Approach to Use Cases and Their Relationships. In *Procs of UML'98*, pages 406–418. LNCS 1618, 1999.

[9] P. Pinheiro da Silva and N. W. Paton. User interface modelling with UML. In *Information Modelling and Knowledge Bases XII*, pages 203–217. IOS Press, 2000.

[10] P. Pinheiro da Silva and N. W. Paton. User Interface Modeling in UMLi. *IEEE Software*, 20(4):62–69, 2003.

[11] A. J. H. Simons. Use cases considered harmful. In *Proc. of TOOLS-29 Europe)*, pages 194–203. IEEE Computer Society, 1999.

[12] P. Stevens. On Use Cases and Their Relationships in the Unifi ed Modelling Language. In *Procs of FASE'01*, pages 140–155. LNCS 2029, 2001.