Describing Use Cases with Activity Charts

Jesús M. Almendros-Jiménez and Luis Iribarne

Dpto. de Lenguajes y Computación, Universidad de Almería, Spain
{jalmen, liribarne}@ual.es

Abstract. The Model-Driven Development (MDD) describes and maintains models of the system under development. The Unified Modeling Language (UML) supports a set of semantics and notation that addresses all scales of architectural complexity by using a MDD perspective. Use Cases and Activity Charts are two modeling techniques of the UML. The first one helps the designers to identify the requirements of the system discovering its high level functionality. The second one helps them to specify the internal behaviour of a certain entity or subsystem of the software developed, such as a database, a graphical interface, a software component, or any specific software. However, there is not a direct way to relate/model the requirements (use cases) with their internal behavior (activity charts). In this paper we present a method for describing use cases with activity charts. Our technique also allow us to identify the two main use case relationships —include and generalization— by means of activity charts. As a case study, we will show how to use the activity charts to describe graphical user interfaces (GUI) from use cases. In particular, we will show an Internet book shopping system example.

1 Introduction

The general processes applied in the development of systems handle the use of solutions based on spiral methodologies [Boe88, Nus01]. These solutions are focused on an iterative use of practices of analysis and design (A&D) and the building of rapid prototypes of several parts (i.e., the business, data and presentation logic) and stages (analysis, design and coding) of the system.

In general, a development process begins with the high level elicitation and description of the requirements of the domain to be modeled. These requirements are then systematically and progressively refined under development. In the process, rapid prototypes of models of the system are simultaneously developed, which are continuously revised by the designers. These revisions change the state of the collection of requirements, modifying or removing the requirements and/or dependencies between requirements, or adding some news one. The spiral model allows us to model the presentation, data and business logic in a concurrent and iterative way. Nevertheless, although this concurrent development method does not imply an independent development between its parts —since there exist a connection— it gets the development of rapid prototypes of models, driving it on the bit parts of the system. The development of

U. Kock Wiil (Ed.): MIS 2004, LNCS 3511, pp. 141–159, 2004.

these rapid prototypes may involve different platforms (such as J2EE or .NET), domains (such as real-time systems, database, graphical interfaces, or software components) and tools (such as IDL, XML or OCL). As a result of this, the *Model-Driven Development* style (MDD) [OMG03] is generating increasing interest due to the needs of methodologies getting a rapid development and a direct connection between models.

The *Unified Modeling Language* (UML) [OMG03] is an A&D feature that supports a set of semantics and notation to address all scales of architectural complexity under MDD perspectives. For instance, the *Use Cases* and *Activity Charts* are two modeling techniques of the UML. The first one helps the designers to identify the requirements of the system, discovering its high level functionality. The second one helps them to specify the internal behaviour of a certain entity or subsystem of the software developed, such as a database, a graphical interface, a software component, or any specific software. However, a direct way to relate/model those requirements identified in the use cases with their internal behaviors modeled in the activity charts is not immediate.

In order to address this gap between models, this paper presents a method that proposes how to describe use cases into activity charts. This method can be considered as a concrete proposal to describe/connect models in the MDD arena, use cases and activity charts in this case. The technique also allows us to identify the two main use case relationships (*include* and *generalization*) by means also of activity charts. In order to clarify the method, this paper presents a case study that puts into practice our approach. This case study handles the use case model for designing *Graphical User Interfaces* (GUI). In particular, we will show a design example of an Internet book shopping system. However, this is just a case study of our approach which is enough general to be used for specifying other system views from the use case model, such as data and business logic, which can complete the system's views following the UML philosophy.

The rest of the paper is organized as follows. Section 2 includes background information on what drove the requirements and the design rationale in the Unified Modeling Language (UML). Section 3 describes a general method for describing use cases with activity charts, and the identification of use case relationships. Section 4 describes an specialization of our method oriented to the design of GUI. Then, Section 5 presents an Internet Book Shopping example that illustrates our method. Finally, Section 6 discusses some conclusions and future work.

2 Discussion of the Unified Modeling Language (UML)

The UML helps the designers working on analysis and design (A&D) with a consistent language for *specifying*, *visualizing*, *constructing*, and *documenting* the artifacts of software systems. One of the primary goals of the UML is to enable *meaningful exchange* of model information between tools, agreement on semantics and notations (for instance, providing IDL specifications as a mech-

anism of model interchange between OA&D tools). The choice of what models and diagrams to create has a profound influence upon how a problem is attacked and how a corresponding solution is shaped. Every complex system is best approached through a small set of nearly independent views of a model. No single view is sufficient.

The UML diagrams provide *multiple perspectives* of the system under analysis or development. The underlying model integrates these perspectives so that a self-consistent system can be analyzed and built. The diagrams, along with supporting documentation, are the primary artifacts that a designer sees, although the UML and supporting tools will provide for a number of views. UML looks for techniques, including component technology, visual programming, patterns and frameworks. UML seeks techniques to manage the complexity of systems as they increase in scope and scale. In particular, the UML recognizes the need to solve iterative architectural problems. One of the key motivations in the minds of the UML developers was to create a set of semantics and notation that adequately addresses all scales of architectural complexity, across all domains. The primary goals of the UML are:

(1) Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models;
(2) Furnish extensibility and specialization mechanism to extends the core concepts;
(3) Support specifications that are independent of particular programming languages and development processes;
(4) Support higher-level development concepts such a components, collaborations, frameworks and patterns.

With regard to (4) the UML should be tailored as new needs are discovered, and for specific domains, specializing the concepts, notations, and constraints for particular application domains.

2.1 Use Case Models

In the UML, one of the key tools for behaviour modeling is the *Use Case* model, originated from the *Object-Oriented Software Engineering* (OOSE) [JCJO92]. Use cases are a way for 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. 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. Actors always model entities that are outside the system. The required behaviour of the system is specified by one or more use cases, which are defined according to the needs of actors. Each use case specifies some behaviour, possibly including variants, that the system can perform in collaboration with one or more actors. Use cases define the offered behaviour of the system without reference to its internal structure. These behaviours —involving interactions between the actor and the system— may result in changes to the

state of the system and communications with its environment. A use case can include possible variations of its basic behaviour, including exceptional behaviour and error handling. Each use case specifies a unit of useful functionality that the system provides to its users, i.e., a specific way of interacting with the system. The behaviour of a use case can be described by means of *interaction diagrams (sequence and collaboration diagrams)*, *activity charts*, and *states diagrams*, or by pre-conditions and post-conditions, as well as natural language text where appropriate. Which of these techniques to use depends on the nature of the use case behaviour as well as the intended reader.

From a pragmatic point of view, use cases can be used for the specification of the (external) requirements on an entity, and for the specification of the functionality offered by an (already realized) entity. Moreover, the use cases also indirectly states the requirements that the specified entity imposes in its users, i.e., how they should interact so the entity will be able to perform its services. One actor can communicate with several use cases of an entity, i.e., the actor may request several services of the entity, and one use case communicates with one or several actors when providing its service. In the case where subsystems are used to model the system's containment hierarchy, the system can be specified with use cases at all levels, and use cases can be used to specify subsystems and classes. In addition, actors representing potential users describe the particular system views of each user, and inheritance between actors is used for specifying common (inheritance of) view of the developed system.

2.2 Generalization and Include Relationships

One of the most controversial elements of the use case model along the UML development has been the Use case relationships named *inclusion*, *generalization* and *extension*, introduced by Jacobson [Jac03]. These relations have an unstable semantics along the UML development. They have received several interpretations [GLQ02, Sim99, MOW03, MOW04], reflecting a high degree of confusion among developers. Our approach for specifying use cases with activity charts is also concerned with the study of use case relationships. In particular, we are interested in the study of the relationships inclusion and generalization. The aim of our work is to provide a more formal definition of use case relationships in terms of their specification by means of activity charts. Our idea is to compare activity charts in order to compare use cases. Such a comparison is abstract and is defined in term of states and transitions included in the activity charts, and interpreted as generalization and <<*include*>> relationships. This topic of research has not been enough explored, although there exist some works which have studied it [Ste01, OP99, Sim99]. On one hand, in [Ste01] use cases are described by means of state machines, and inclusion relationship is studied. On the other hand, in [OP99, Sim99] the cited and extend relationship proposed by Jacobson are discussed but without providing a formal description.

From a pragmatic point of view, an inclusion relationship between two use cases means that the behaviour defined in the target use case is included at one location in the *sequence of behaviour* performed by the base use case. A use case

may be included in several other use cases, and a use case may include several other use cases. In that sense, the included use case represents encapsulated behaviour which may easily be reused in several use cases. A generalization relationship between use cases implies that the child use case contains the sequences of behaviour and participates in all relationships of the parent use case. The child use case may also define new behaviour sequences, as well as additional behaviour, and specialize existing behaviour of the inherited ones. A use case may have several parent use cases, and a use case may be a parent to several other use cases. Therefore, inclusion and generalization relationships are closely related with well-known concepts on object-oriented A&D, such as *encapsulation* and *inheritance*. Up to now, this interpretation has been intended for UML experts, but it needs particular interpretations depending on the diagrams used for specifying use cases.

Due to the iterative perspective of the model-driven development of the UML, the use case view of a system can be refined in the development process, and therefore, inclusion and generalization can also be used for *tracing* the development. System designers can include, generalize and specialize certain use cases which were described in early stages of the development process. In summary, use case model can be used for documenting both the final system and the development process, enabling the maintenance of the system. From the point of view of future system designers, use case model allow the knowledge of the system (components) structure and behaviour and the relationships of integrated components: tasks, requirements, services, external needs, behaviour, and so on. The integration of a new system with the older one should take into account the particular requirements specified on the use case model.

2.3 Activity Charts

Activity charts basically describe the set of *states* (and the corresponding *transitions*) which a given entity follows when a given service is required. Therefore, activity charts can be used for specifying the *internal behaviour* of a certain entity or subsystem of the developed software, such as a database, a graphical interface, a software component, or any specific software. Depending on the nature of the entity to be specified, the states can describe *internal states*, that is, states in which no interaction with the environment is achieved, and *external states* which involves communication with actors and other entities. In addition, transitions can be also subdivided following the same criteria, that is, *internal transitions* achieved by the entity and *external transitions* with participation of the outside world.

In our approach, an activity diagram is used for specifying the set of behaviour sequences of a given use case, allowing the detection of included use cases (included sequences of behaviour) as well as more general and particular use cases (generalizing and specializing sequences of behaviour). With this aim, we need to define an abstract definition of similar properties on activity diagrams, named inclusion and generalization. Given that activity diagrams handle states and transitions, which describes behaviour sequences, the inclusion and general-

ization relationships between activity diagrams are defined in terms of behaviour sequences. The case of inclusion is simpler describing *subsequences of behaviour*, but generalization uses in its definition an abstract *replacement relationship*. It can be identified with the usual concept of replacement of object-oriented A&D, that is, a certain component offers the same functionality (eventually, adding new functionality) as another one, and therefore the second one can be replaced by the first. This generic view of replacement induces a generic generalization relationship on activity diagrams, which induces the same use case relationship. Therefore use case modeling offers an object-oriented system view.

3 A Method to Describe Use Cases with Activity Charts

In this section, we will show how to describe use cases with activity charts. With this aim, we present a method based on the identification of use case relationships from the description of activity diagrams.

3.1 Identifying Actors and Use Cases

Firstly, the developer describes the functional requirements of the system by means of a use case diagram. A use case diagram consists of a set of actors (users and external systems) and use cases.

Relationships between actors are *generalizations*. An actor p is more general than another actor q whenever q can interact with the system as p and additionally, can interact in more cases. The developer should also identify the set of use cases related to each actor that will represent the set of tasks to be achieved by the actor. Relationships between actors and use cases are called *associations*. In our method, generalizations and associations may be identified in the first step, but they can be refined later. This first developing process will get a (still non formal) use case diagram.

3.2 Describing Use Cases with Activity Charts

Once the actors and use cases associated with each actor have been specified, the developer should provide in a second step a set of activity charts to describe each use case in the use case diagram. They may be basically specified in the early stages of the development process and refined later.

Activity charts are graphs linking *states* by means of *transitions*, which are arrows connecting an *origin state* and an *end state*. There are two special states: *initial* and *final* states. Initial (resp. final) state is the starting (resp. end) point of the activity chart. Each state may represent a system state and transitions can represent actor interactions (user events, external system calls) or internal system behaviour (execution steps and threads).

Transitions are labeled by means of *conditions/actions*, representing conditions to be hold and actions to be achieved for state change. There can be *diamonds* between transitions describing alternative paths depending on a *boolean*

condition. An state can also be described by means of a separate activity chart, describing (sub)states and transitions performed in the state. In this case, the state is called *non-terminal state*, and otherwise it is called *terminal states*.

3.3 Identifying Activity Chart and Use Case Relationships

The third (and last) step consists on the identification of use case relationships from the described activity charts. The relationships between uses cases are <<*include*>> dependencies, together with generalizations. Here, the developer should apply the following rules to compare activity charts, which induces the <<*include*>> and generalization relationships between use cases. This provides a new more formal and refined use case diagram in which there have been specified the cited use case relationships. Both activity diagrams and use case diagram can be refined in later stages of the development process when the knowledge of the system requirements is refined. That is, use case relationships can be late detected due to the refinement of the activity diagrams.

Identifying Activity Chart Relationships. Firstly, we have to assume that terminal states and transitions of activity charts can be compared by means of a (reflexive) *replacement relationship* $\sqsubseteq$ in such a way that two (terminal) states satisfies $s \sqsubseteq s'$ (or two transitions $\lambda \sqsubseteq \lambda'$) whenever s' can be replaced by s (or λ' can be replaced by λ). The replacement relationship can express similar semantics (or behaviour). The replacement relation should be decided by the developer.

This replacement relationship induces a *replacement relationship on activity charts* in such a way that an activity diagram a' can be replaced by a if the states and transitions of a' can be replaced by the states and transitions of a.

In addition, activity charts can be compared by means of an *inclusion relationship*. Inclusion can be intended as an activity chart that includes another one, but without changing the behaviour, that is, states and transitions are not modified, and neither new states or transitions can be added over the included activity chart. In practice, if an activity chart a includes an activity chart a' then one of the states of a is a', although a may implicitly include all the states and transitions of a'. For simplicity, we assume the first case.

Obviously, some activity charts may not be compared by means of replacement and inclusion relations, this means they do not describe "similar" activities. However, some activity diagrams can be compared by means of a combination of replacement and inclusion and not by a single one. In this case, in order to be compared, there should be defined *intermediate* activity charts (decomposing the activity charts), in such a way the original activity diagrams could be compared through *a chain of relationships*. In the whole development process, intermediate activity diagrams can be detected when specifying new users interactions of refinement of the existent ones.

Identifying Use Cases Relationships. Once we can compare activity charts by means of the above relationships, the use cases —which are described by means of activity charts— can also be compared by means of the following relationships.

— A use case u *includes* a use case v whenever the activity chart of u includes the activity chart of v.
— A use case u is more general (*generalizes*) than other v whenever there exists an activity diagram u' such that u can be replaced by u' and v includes u'.

In the UML there are some additional information about the use case diagrams like roles, multiplicity, directionality, and *extend* dependencies, but they will not be considered in our approach yet.

4 A Case Study for GUI Design

In order to clarify the cited concepts, in this paper we present a case study that puts into practice our approach. This case study handles the use case model for designing *Graphical User Interfaces* (GUI). Our general method can be specialized depending on the nature of the system to be developed or the part of the system to be build. Although we have decided to apply our method for designing GUIs (that is, to describe the presentation logic of the system), however, it does not means that the first view of the system to be developed is the user interface. Following the UML philosophy, many views of a system can be built in the early stages and refined in later steps, such as business and data logic views. Each stage complements each other, providing a multiple view perspective of the developed system. We have chosen the user interface view as case study given that by its simplicity the main concepts of our approach (i.e., inclusion and generalization) can be detected in user interface design.

Graphical user interfaces have become increasingly dominant, and the design of the "external" or visible system has assumed increasing importance. This has resulted in more attention, being devoted to usability aspects of interactive systems, and a necessity of tool development that supports the design of the presentation logic. Models and notations are required to describe user tasks, and map these tasks on to the user interface. The user interface (as a significant part of the 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 some recent approaches [Kov98, dSP03, dSP00, EK00, EKK99, Nun03, BNT02] which have addressed this problem. The proposals of [dSP03, dSP00, Nun03] 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 methodology 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 a correspondence between use case relationships and GUI components.

Another similar work to our contribution is [EK00, EKK99] 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.

In our case study, activity charts are used for describing user interaction. In particular, they describe the presentation logic of an *applet-based system* (similar to [EK00, EKK99]), in which a set of applet windows are shown to the user, and the user interacts with them in order to put and get data from the system. With respect to this choice, our aim is not to constraint the implementation to a particular window system but the acceptance of the *JAVA swing classes* between developers allows us to assume the reader is familiarized with the implementation details of the GUIs. Our framework can be adapted to others user-interface development technologies with a bit of effort, adding new UML stereotypes for both input and output components, and making similar mappings between use cases and other kind user windows.

4.1 Specializing Activity Charts

In the activity charts, states represent outputs to the user which are labeled with UML *stereotypes* representing visual components for data output. Transitions represent user inputs which are labeled with UML stereotypes representing visual components for data input and choices. Once activity diagrams describe each user interface for each actor in the use case model, the inclusion relationship describes subsequences of interaction with the graphical interface and the generalization relationship inheritance of common (eventually, more particular) tasks and interactions between several users.

The activity charts can be specialized for describing user interaction as follows:

— Each state of the activity chart describing a use case necessarily falls in one of the two following categories:
  - A terminal state is labeled with a *UML stereotype* representing an *output GUI component*. Therefore, they are also called *stereotyped states*.
  - A non-terminal state is not labeled and is described by means of an activity chart. The non-terminal states can be "use cases" of the use case diagram or not.

— Each *transition* in the activity chart of a use case 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*.

With respect to the relation of replacement, it is useful for instance to reuse some design artifacts of a model (i.e., a collection of states, transitions, stereotypes), or change the functionality of an artifact. In the case study the stereotyped states can be replaced whether the *output GUI component* can be replaced. For instance, a list with two columns can be replaced by another 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.

4.2 Building GUIs

Now, we can build our GUI from the use case diagram and the set of activity charts following the next rules:

— Each *actor* representing a user in the use case diagram is an *applet*[1]. Actors representing external systems are not considered for visual component design.
— The *generalization relationships* between two actors p and q (p generalizes q) corresponds with *inheritance* of the applet represented by q from the applet representing p[2].
— Each *use case* in the use case diagram is an *applet*[3].
— The *generalization relationship* between two use cases u and w (u generalizes w) corresponds with *inheritance* of the applet representing w from the applet representing u.
— The *<<include>>* *relationship* between two use cases u and w (u includes w) corresponds with the *invocation* from the applet representing u of the (sub)applet representing w[4].
— In the non-terminal states case, the use case diagram can specify *<<include>>* or generalization relationship between the non-terminal state and the use case, and we follow these rules:
  - In the *<<include>>* relationship case, the non-terminal state is also an applet and contains the GUI components in the associated activity diagram.
  - In the generalization relationship case, the non-terminal state is also an applet containing the GUI components in the associated activity diagram, but the use case also contains these GUI components.
— The non-terminal states —which does not appear in the use case diagram— are not applets, and the GUI components in the associated activity diagrams are GUI components of the applet of the use case.
— The conditions of the transitions of an activity chart are not taken into account for the GUI design.

[1] For some complex user interfaces, the applet class could be replaced with other specialized swing classes like the frame class, which enables to build a single window in which the complete functionality of each actor is presented.
[2] When using frames we can use inheritance of frames.
[3] When using a frame for implementing an actor, a use case connected with an actor can be implemented either with a separate applet or using a window area, and an applet built in the main frame.
[4] When using frames, the invocation can be replaced by the built-in of the applet.

5 The Internet Book Shopping (IBS) Example

In this section, we will explain a simple example of an Internet book shopping (IBS) that illustrates the functionality of our proposed method. In the IBS there will basically appear three actors: (a) the customer, (b) the ordering manager, and (c) the administrator. A customer actor directly carries out the purchases by the Internet. The customer can also consult certain issues of the product in a catalogue of books before carrying out the purchase. On the other hand, the manager deals with (total or partially) customer's orders. Finally, the system's administrator actor can manage the catalogue of books adding and removing new books in the catalogue or modifying those already existing. The administrator can also update or cancel certain component characteristics of an order or those orders fulfilling certain searching criteria. Furthermore, both the manager and the administrator should identify themselves before carrying out any kind of operation restricted to his/her environment of work.

Considering this framework, in the next sections we will describe an IBS GUI project that illustrates the proposed method, i.e., how to connect the system's use cases description to activity charts. This connection helps the developer to translate and/or connect some analysis features (i.e., use cases descriptions) to/with behavioural descriptions of the system. In our case, these behavioural descriptions only will concern with activity charts.

5.1 Modeling the IBS System Using Use Cases

Let's suppose that the developer of the IBS system wishes to model the *presentation logic* by means of use cases. Initially, the use case diagram contains the identified actors of the system: in our IBS example, the actors are the `Customer`, the `Manager` and the `Administrator`.

To model the presentation logic of the IBS system, the developer should previously identify all those future windows of the system (graphical user interfaces), carrying out some quick outline of their content[5]. The connection of an actor with one or more use cases in the use case diagram will be interpreted as a set of options (menu) on a first window on which the actor will interact with the system. Figure 1 shows some prototypes of those windows of the IBS presentation logic that the developer wants to reach[6].

Initially, the use case diagram, which models the IBS presentation logic, contains three kinds of interfaces identifying the actors of the system. A first non-formal description of the IBS system using a use case diagram is shown in

[5] As we said before, the model-driven development follows an iterative refinement process and therefore, here, we are supposing the designer has a well-defined collection of the GUI requirements.

[6] The windows shown in Figure 1 are only a pragmatic simple example as result obtained after applying our method. We suppose that the designer of the system has decided (for any reason) this window organization. We do not consider whether the distribution is suitable or rationale.

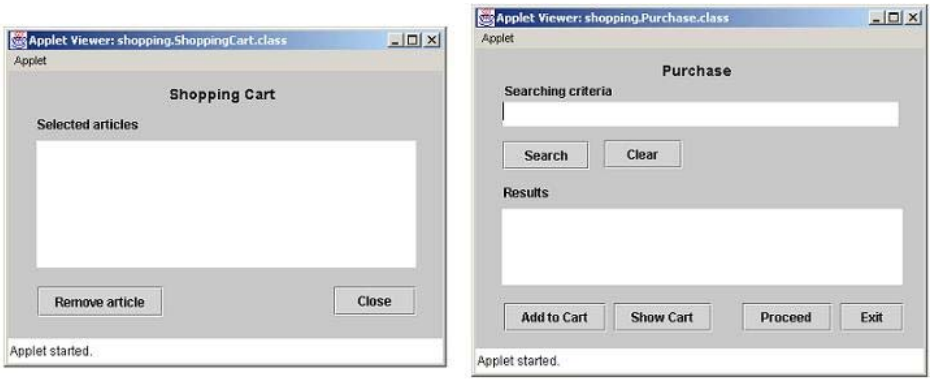

Fig. 1. Some windows of the IBS presentation logic

Figure 2. The diagram also contains all the high-level functions represented by means of use cases. For example, note that the administrator's interface agrees to the *presentation logic* through three possible options (use cases): `Manage catalogue`, `Update orders` and `Update partial orders`. Something similar happens to the remainder actors of the system.

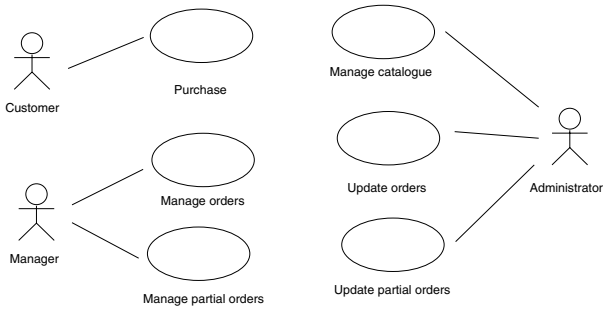

Fig. 2. A non-formal use case description of the IBS example

As we have mentioned in the previous section, the non-formal definition of the system will be refined, causing more precise use cases diagram(s). Figure 3 shows a more complete *presentation logic* definition for the IBS system. In this case, we have chosen to include all the *presentation logic* in a single use case diagram, although it could be itemized in more than one to deal with more structured use case descriptions.

Let's see now certain aspects of the use case diagram shown in the figure. Firstly, we emphasize the use given to the relationships *<<include>>* and *generalizations*. In our method, the *<<include>>* relationship can be used to represent optional or mandatory behaviour. This two kinds of relationships are properly modeled and interpreted in the activity charts associated with both con-

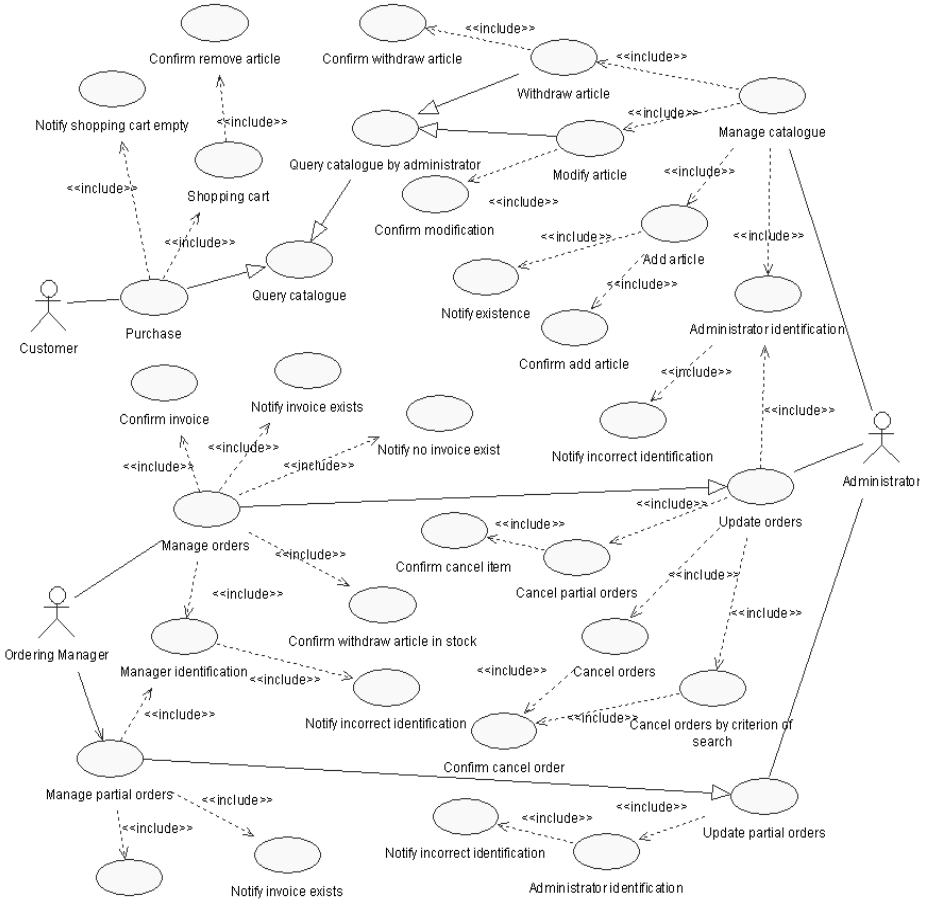

Fig. 3. The refined IBS use case diagram

nected use cases, since use case diagram does not distinguish between hard/soft dependencies (i.e., mandatory/optional relationships). For instance, the use case `Manage catalogue` is an applet that directly depends on four use cases, connected to them by means of a <<*include*>> relation. However, these connections are similar to three of them, and different from the others. The <<*include*>> relationships between the use cases `Withdraw article`, `Modify article` and `Add article` were modeled by the system's designer as relations of optionally (the branches of the use case `Manage Catalogue`'s behaviour go to these states in the activity chart). However, 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 chart) of the use case `Manage Catalogue`.

Therefore, an <<*include*>> relationship can mean that a use case could be considered as a composition of two or more other use cases. For instance, the use case `Manage catalogue` is composed of the use cases `Withdraw article`,

`Modify article` and `Add article` (i.e., applets or frames). An *<<include>>* relationship can also indicate that a use case mandatory depends on another use case to operate. For example, since the administrator should be itself identified before working with the system, the three use cases which he/she directly operates with (i.e., `Manage catalogue`, `Update orders` and `Update partial order`) mandatory depend on the `Administrator identification` use case.

In our case, the relation of *generalization* is intended as an inheritance of behaviour (i.e., GUI components). For example, the use case `Query catalogue` has been established as a generalization of the use case `Purchase`. This relation will mean that, due to own reasons, the developer of the system wishes to model the purchasing process re-using and modifying the functionality of the query process. Note how the use case `Query catalogue by administrator` also inherits from query catalogue and generalizes the use cases `Withdraw article`, and `Modify article` connecting them to a part of the client's *presentation logic* and the administrator side

The distinction between *include* relationships (mandatory, optional) and generalization is established by the system's designer into the activity charts of those include-connected use cases. In the following sections we will only focus on the `Purchase` use case to explain the behaviour of the method.

5.2 Mapping the IBS Features, Use Cases into Activities

As the developer stated, each use case modeling the presentation logic will correspond with an applet (or frame) component. Activity charts 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 chart 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 and output graphical components are associated 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.

A **state** can be stereotyped or not. Stereotyped states represent terminal states which can be labeled by the `<<JTextArea>>`, `<<JList>>` and `<<JLabel>>` stereotypes. For instance, Figure 4 shows the activity chart for the `Purchase` use case. This diagram shows the graphical and behavioural content of the applet window where the purchases can be carried out. The activity chart 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 chart in order to structure better the design. The name of a separate activity chart should be the same as the one of the state.

The activity chart's behaviour, in Figure 4, 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.

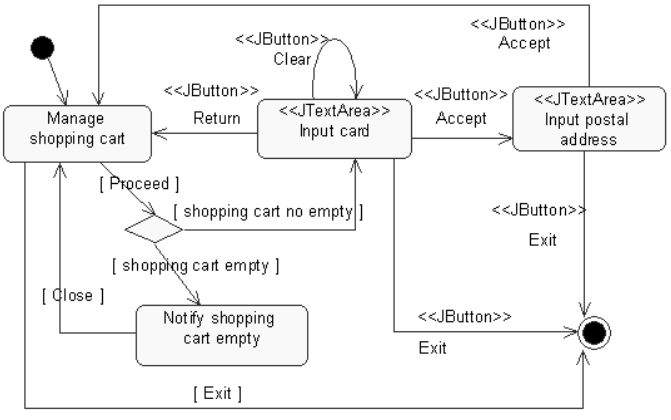

Fig. 4. The activity chart for the `Purchase` use case

According to the GUI developer's rules, if a state is not labeled with a stereotype, this means that the state is described in another activity chart. This new activity chart can either represent the behaviour of another use case or simply a way of allowing a hierarchical decomposition of the original activity chart. For example, in the activity chart associated with the `Purchase` use case (see Figure 4), there appear two non-terminal states: `Manage shopping cart` and `Notify shopping cart empty`. At the same time, two activity charts are described for both states. All these activity charts are shown in Figure 5.

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

Transitions in the activity chart 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 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 chart shown in Figure 5 (b), the list `Results` is modeled by a `<<JList>>` state and the condition `[Selected article]`. Figures 4 and 5 show 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 chart (see Figure 5). 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.

Furthermore, stereotyped transitions (buttons in our example) and conditions connect (non) terminal states to (non) terminal states. As we said before, a condition would be an output of a non-terminal state in case the user interacts with a button or a list component inside the respective non-terminal state. The usual way "condition/action" transition can connect (non) terminal states to (non) terminal states. A condition/action transition between states means which condition should be present to achieve the action. In our case study, an action can only be a button. For instance, to remove an article from the shopping cart, it must previously be selected from the cart list (Figure 5, c).

Condition/action transitions are also useful to model the behaviour of the generalization relationships between use cases in a use case diagram. 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.

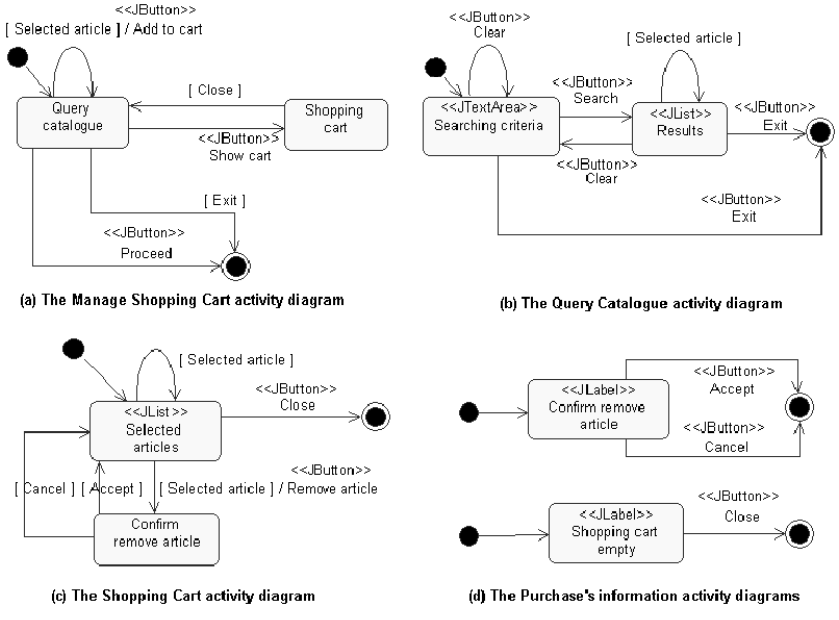

Fig. 5. The whole activity chart of the `Purchase` use case

This inheritance behaviour is modeled in the Purchase activity chart as a non-terminal state that includes the behaviour of the Query Catalogue activity chart. For example, let us observe the behaviour of the query catalogue shown in Figure 5 (b). In this activity chart, 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`. Thus, when the Purchase use case inherits the Query Catalogue use case, 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 chart of the Purchase use case as a non-terminal state called `Query catalogue` (see Figure 5, a). The output transition `[Selected article]/Add to cart` mean 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` at the Purchase applet (use case) mean that both the `Proceed` and `Show cart` 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 charts) 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 chart) of the Query Catalogue use case to model the behaviour (activity chart) 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.

To develop the IBS project example we have used the Rational Rose for Java tool. 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`.

6 Conclusions and Future Work

In this paper, we have studied a method for describing use cases by means of activity charts based on a Model-Driven Development (MDD) perspective [OMG03]. The use case diagrams help the developer to identify the requirements of the system and to study its high level functionality. The activity charts

allow us to discover new behavioural details of the system or to describe better the already existing. Nevertheless, in this paper we have shown how a direct correspondence between the requirements identified in the use cases with these UML activity diagrams is feasible. The technique also allow us to identify the two main use case relationships (include and generalization) by means also of activity charts. Through a case study, we have shown how our technique can be applied in GUI-oriented component development for rapid prototyping of the external view of the system. Although in our approach we can completely specify the behaviour of the GUI components of the system, in the early stages of the system development a basic behaviour can be specified which could be refined in later stages. We have chosen the JAVA swing classes and components for describing user interfaces due to the acceptance of this technology (*applet*, *frames*, and *events*), however our approach can be adapted to other technologies for GUI building (for instance, *hypertext* and *hyperlinks*). Finally, the use case model together the GUI specification with activity diagrams provide a description of the behaviour and structure of the GUIs of the developed system, and in general can be used for describing a library of reusable GUI components.

As a future work, we firstly plan to apply our general method to other parts of the system (i.e. business or data logic). Secondly, we would like to extend our work to deal with the <<*extends*>> relationship of use cases. Thirdly, we would like to formalize and to incorporate our method in a CASE tool in order to automate it. And finally, we would like to integrate our technique in the whole development process.

Acknowledgements

The authors would like to thank the anonymous referees for their insightful comments, that greatly helped them improve the contents and readability of the paper. We would also like to thank to the attendees of MIS'04 for the suggestions about our paper. This work has been partially supported by the Spanish project of the Ministry of Science and Technology "INDALOG" TIC2002-03968 under FEDER funds.

References

[BNT02] R. Biddle, J. Noble, and E. Tempero. Essential use cases and responsibility in object-oriented development. In *Australasian Computer Science Conference (ACSC2002)*, 2002.

[Boe88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[dSP00] P. P. da Silva and N. W. Paton. User interface modelling with UML. In *Information Modelling and Knowledge Bases XII*, pages 203–217. IOS Press, 2000.

[dSP03] P. P. da Silva and N. W. Paton. User Interface Modeling in UMLi. *IEEE Software*, 20(4):62–69, 2003.

[EK00] M. Elkoutbi and R. K. Keller. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In *International Conference on Application and Theory of Petri Nets (ICATPN 2000)*, pages 166–186. LNCS 1825, 2000.

[EKK99] M. Elkoutbi, I. Khriss, and R. K. Keller. Generating user interface prototypes from scenarios. In *IEEE International Symposium on Requirements Engineering (RE '99)*, page 150. IEEE Computer Society, 1999.

[GLQ02] G. Génova, J. Llorens, and V. Quintana. Digging into use case relationships. In *International Conference on the Unified Modeling Language (UML 2002)*, pages 115–127. LNCS 2460, 2002.

[Jac03] I. Jacobson. Use Cases – Yesterday, Today, and Tomorrow. Technical report, Rational Software, 2003.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: a Use Case Driven Approach*. Addison-Wesley, 1992.

[Kov98] S. Kovacevic. UML and User Interface Modeling. In *International Conference on Unified Modeling Language (UML'98): Beyond the Notation*, pages 253–266. LNCS 1618, 1998.

[MOW03] P. Metz, J. O'Brien, and W. Weber. Specifying use case interaction: Types of alternative courses. *Journal of Object Technology*, 2(2), March-April 2003.

[MOW04] P. Metz, J. O'Brien, and W. Weber. Specifying use case interaction: Clarifying extension points and rejoin points. *Journal of Object Technology*, 3(5), May-June 2004.

[Nun03] N. J. Nunes. Representing User-Interface Patterns in UML. In *International Conference on Object-Oriented Information Systems (OOIS 2003)*, pages 142–151. LNCS 2817, 2003.

[Nus01] B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–117, March 2001.

[OMG03] OMG. Unified Modeling Language Specification, version 2.0 (MDA). Technical report, Object Management Group, June 2003.

[OP99] G. Övergaard and K. Palmkvist. A Formal Approach to Use Cases and Their Relationships. In *International Conference on the Unified Modeling Language (UML'98): Beyond the Notation*, pages 406–418. LNCS 1618, 1999.

[Sim99] A. J. H. Simons. Use cases considered harmful. In *International Conference on Technology of Object-Oriented Programming Languages and Systems (TOOLS-29 Europe)*, pages 194–203. IEEE Computer Society, 1999.

[Ste01] P. Stevens. On Use Cases and Their Relationships in the Unified Modelling Language. In *International Conference on Fundamental Approaches to Software Engineering (FASE'01)*, pages 140–155. LNCS 2029, 2001.