Use Case and User Interface Patterns for Data Oriented Applications

António Miguel Rosado da Cruz(✉)

Escola Superior de Tecnologia e Gestão, Instituto Politécnico de Viana do Castelo, Av. do Atlântico, s/n, Viana do Castelo, Portugal
miguel.cruz@estg.ipvc.pt

Abstract. Use case driven software development starts, in general, with abstract problem domain descriptions of how the users see themselves using the system being developed, and involves a series of iterative refinement steps that incrementally add detail to the use case model, bringing those descriptions to the solution domain. Use cases involve interactions between human actors and the system state. These interactions are held within interaction spaces, which are modeled through a user interface model. Business applications are in general data-driven, comprising a set of typical functions that the users can make on the system. When a use case driven approach is used to develop data-oriented applications those typical functions pop-up as use case patterns, and their interactions occur within a set of user interface patterns. This paper presents a set of use case patterns and the corresponding user interface patterns typically found in data-oriented business applications. For that, a user interface metamodel and corresponding concrete user interface modeling language are also proposed.

Keywords: Use case model · Use case patterns · Interaction modeling · User interface model · User interface patterns · Model-driven development

1 Introduction

Use cases are present in most software projects, and evolve iteratively since the first analysis activities until the activities of design and coding. Use case driven software development encourages software engineers to follow an approach that is guided by the system functionality. This approach, typically starts with high-level problem domain descriptions of how the users see themselves using the system being developed, and involves a series of iterative refinement steps that incrementally detail the use case model, bringing those descriptions to the solution domain [1]. These refinement steps comprise the simultaneous development of a domain model, which models the domain entities and the structural relations between them [2], and the model of the system state on which the system functionality will act upon. Such a process produces increasingly detailed use case models and domain entity models that must be kept consistent with each other [3].

According to the UML specification [4], a use case, being a *BehavioredClassifier*, specifies some offered behaviors, which involve interactions between its actors and a subject comprising a collection of classifiers. This collection of classifiers that form the subject of a use case may be the system state or a partial view of the system state containing the domain entities affected by the use case behaviors (its functionality). Use case behaviors may be semi-formally specified, in UML, through various means, including state machines, activities, interactions, pre-conditions, post-conditions and natural language text [4].

As the use case model becomes more detailed, with use cases including or being extended by other, more concrete, use cases, its use case specifications become more obvious, and each use case behaviors may be informally inferred from a short description or from the use case name itself. These use case behaviors act on one or more system domain entity instances (its subject or collaborative entity classes) [4], so the use case model needs to be closely related to the system domain model. This proximity is in the sense that the use case behaviors refer to entities from the domain model [5].

Indeed, at platform independent level, use case (UC) and domain models are two sub-models (views) of one and the same system model. The former models a vision of the system functionality, and the latter models a vision of its structural features [2]. Other relevant model views are the system user interface (UI) view, modeled by a user interface model (UIM), and a behavioral view, which is, in this approach, divided between class methods and invariant constraints in the domain model and through use case behaviors, which may be specified as mentioned above [5].

When confining ourselves to data-oriented applications, which constitute the vast majority of business applications, the use case model tends to present a set of use case patterns that comprise typical functions that the users can make on the system [6]. Similarly, the spaces (user interface) where those use case patterns interactions take place form, at a platform independent level, a set of user interface patterns. This paper's main contribution is the presentation, in Sect. 5, of the user interface patterns related to the use case patterns previously presented in [6], and summarized in Sect. 3. Another major contribution is the proposition of a UI metamodel, aligned with UML, and the corresponding UI modeling concrete notation, in Sect. 4.

The next section addresses some background issues concerning abstract user interface models and the UML use case metamodel. Section 3 presents the mentioned set of use case patterns typically found in business (data-oriented) applications. In Sect. 4, a UI metamodel and the concrete notation for constructing UI models is proposed, and in Sect. 5, the set of abstract user interface patterns corresponding to the use case patterns previously defined, is presented. Section 6 illustrates the relation between use case and UI patterns through a demonstration case. Section 7 overviews related work and, finally, Sect. 8 concludes the paper.

2 Background

This section addresses some background issues, namely user interface models and the Canonical Abstract Prototypes (CAP) notation for modeling abstract user interface models, in the next subsection, and the UML use case metamodel, in Subsect. 2.2.

CAP will be used, in Subsect. 4.2, as a basis for proposing a concrete notation for abstract UI models that conform to the metamodel proposed in Sect. 4.

2.1 Abstract User Interface Models

User Interface model-based development techniques build a more or less declarative User Interface Model (UIM), which is typically composed of various sub-models, or model views. This UIM captures the relevant aspects of the UI and is typically developed using a model-based user interface development environment (MB-UIDE) [7]. Different MB-UIDEs use different kinds of models specified with different kinds of modeling languages.

Typically, a model-based UI development process begins with the construction of a task model [7, 8]. Afterwards, an abstract interaction model (or abstract UI model) is built and at the end of the process a concrete interaction model (or concrete UI model) is constructed.

User Interface Models provide a description of the UI at different levels of abstraction. Platform dependent (concrete) user interface models make use of widgets and functionality that may be specific to one given platform. Platform independent declarative (abstract) user interface models provide an abstract description of the UI, which can be reused and can be refined to more concrete (platform dependent) models.

UIMs can, then, be found at different levels of abstraction during the UI design process. A UIM provides an infrastructure for allowing automated tasks in the UI design and implementation processes.

Canonical Abstract Prototypes is an approach and notation, proposed by Constantine [9], for capturing the presentation aspects of interactive systems. Canonical Abstract Prototypes capture only the abstract presentation aspects of a user interface, by making use of abstract interaction objects (AIO), which are UI elements that don't have a unique concrete representation.

CAP is based on 3 extensible generic universal symbols [9]:

1. Material (or generic container): represents information, data or other objects shown to the user during a task.
2. Tool (or generic action/operation): represents UI objects that can be used to manipulate, control or transform materials.
3. Hybrid (or active material): represents UI components with characteristics both from materials and tools like, for instance, editable fields or lists of selectable items.

Figure 1 shows the main symbols of the canonical abstract notation, which allow the development of abstract user interface models (AUIM) like the one shown in Fig. 2. The figure shows a prototype of a selectable list and the output of detailed information about the selected list item. The symbol ⋙ represents repetition and, in the example, it means that the aligned elements in the Film Clips selectable collection are repeated in every line.

2.2 Use Cases in the UML Metamodel

Use cases can be used both for modeling the external requirements of a subject and the functionality offered by a subject. In both cases the subject can be the system domain model or a subset of it. Moreover, use cases can also be used to specify the requirements the subject (the system) poses on its environment, by defining how the actors should interact with the subject [4].

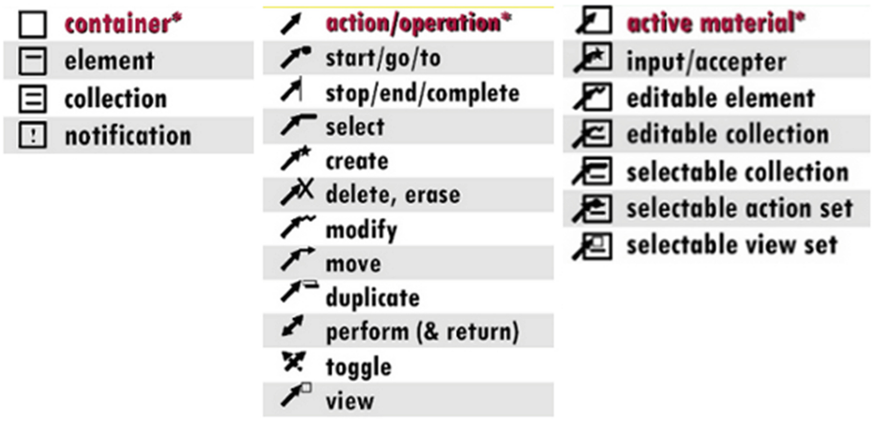

Fig. 1. Canonical abstract prototypes symbols (adapted from [9]).

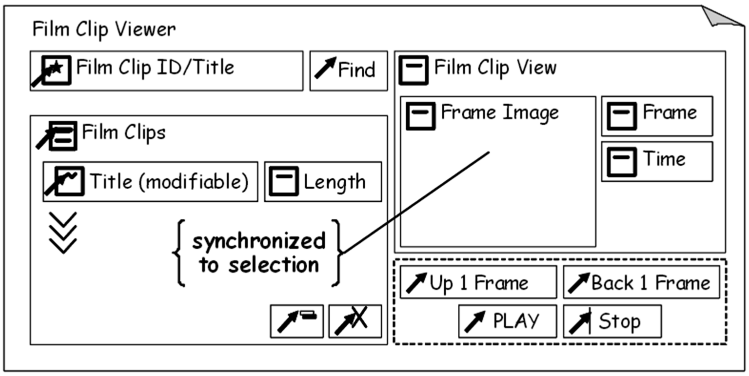

Fig. 2. Example of a CAP for a Film Clip Viewer (taken from [9]).

The UML metamodel for use cases (see Fig. 3) supplies two use case relations, namely Extend and Include, which allow the modeler to organize a use case behaviors into further refined behaviors that are included in a bigger, more complex, use case, and optional or conditional behaviors that may extend the bigger use case by the actors' option or when certain pre-conditions hold.

Besides those two relations, as a (*Behaviored*) Classifier, a use case may also specialize another use case through an Inheritance relation. A use case that inherits from another use case, inherits all its features (included use cases, associated Domain Model Classifiers and Features, etc.).

Use cases comprise behaviors that can be instantiated within an interaction. Those behaviors consist of a specification of events that may occur dynamically over time [4]. On a behavior invocation, the actual sequence of events that occur, and are consistent with the behavior specification, is called an execution trace [4]. An execution is, then, an instance of a Behavior.

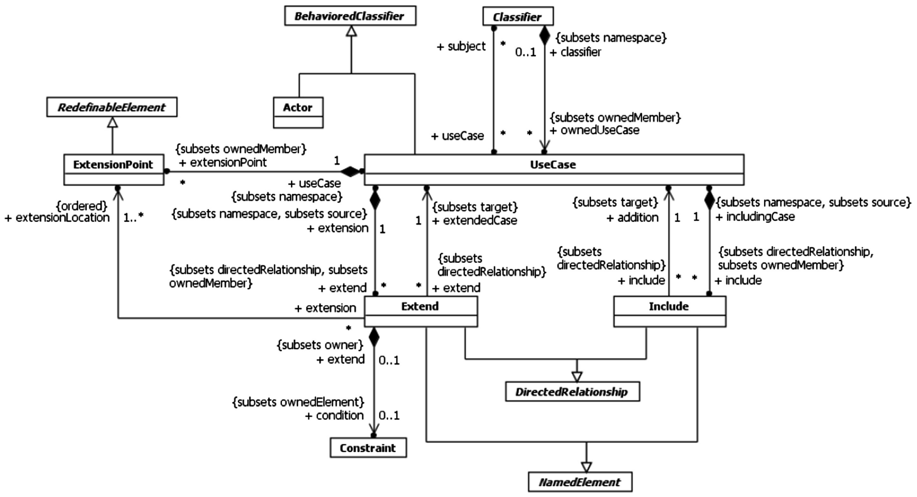

Fig. 3. Use cases portion of the UML metamodel (taken from [4]).

A system use case model acts upon the system domain model, whose instance forms the system state.

Use cases define behaviors that may modify the system state. Those behaviors occur within interactions, which form the space where actors interact with the system, in the scope of a use case.

A use case behavior can be seen as an orchestrator of its owned subject's behavioral features. In a model-driven setting, a use case behavior can be defined, for instance, through Alf, OMG's proposal for Action Semantics concrete notation [10].

The simpler types of use case behavior consist of calling CRUD (create/retrieve/update/delete) operations over domain entities (the use case subject), user-defined operations enclosed in methods within the use case subject, and navigational operations over domain entities that are available within each use case.

The use case model also identifies the actors (user roles) that have access to each use case (system functionality), thus providing authorization information about the system.

When focusing on data-oriented applications, a set of use case patterns, and associated behavior and domain model entities (subjects), can be identified, as addressed in the next section.

3 Use Case Patterns for Data-Oriented Applications

Data oriented applications have as main functionality the management of stored entities' information. Operations in such applications typically include listing the (possibly filtered) instances of an entity, editing entity properties, defining or modifying entities' relationships, etc., and may be grouped in the following use case patterns [3, 6]:

- Manage an entity instance;
- Manage dependent related entity instances;
- Manage independent related entity instances;
- Manage dependent related entity collections;
- Manage independent related entity collections.

This section presents these typical functionality patterns, modeled as use case diagrams, taking the form of use case patterns that can be used in constructing a system's use case model.

Two categories of use cases can be distinguished in the patterns presented herein [3, 11]:

- Independent use cases: can be initiated directly, and so can be linked directly to actors, which initiate them. Independent use cases cannot extend and cannot be included in any other use case.
- Dependent use cases: can only be initiated from within other use cases, called source use cases, because they depend on the context set by these. Dependent use cases extend or are included by the source ones, according to their optional or mandatory nature, respectively.

Some use cases may exhibit characteristics from either of these categories, depending on the use case where the actor-system interaction begins.

3.1 Manage an Entity Instance

Managing an entity instance typically involves listing all or some of the existing instances, and selecting one of those instances for editing (retrieving its information for visualizing, updating or deleting it), or creating a new instance.

"Manage an entity instance" is, thus, a use case pattern comprising three use cases where use cases for creating an entity instance (Create E1; see Fig. 4(a)) and editing an existing instance (Retrieve, Update, Delete E1) are dependent of, and extend, the use case for listing existing instances (List E1).

List E1 may also be extended with a use case for defining filtering criteria. And, of course, Create E1 might also be directly accessed by actors.

As specified in [3, 11], each use case references an entity (its subject) through a tagged value, for consistency between models. All use cases of this pattern refer to the same entity in the domain model (E1).

3.2 Manage Dependent Related Entity Instance

A dependent related entity instance is an instance of an entity E2 that has a "one to one" or a "zero-or-one to one" association with E1 (refer to Fig. 4(b)).

Managing the instance of E2 associated to a given instance of E1 typically involves creating a new related instance (Create Related E2, in Fig. 4(b)), or editing the existing related instance (Retrieve, Update, Delete Related E2).

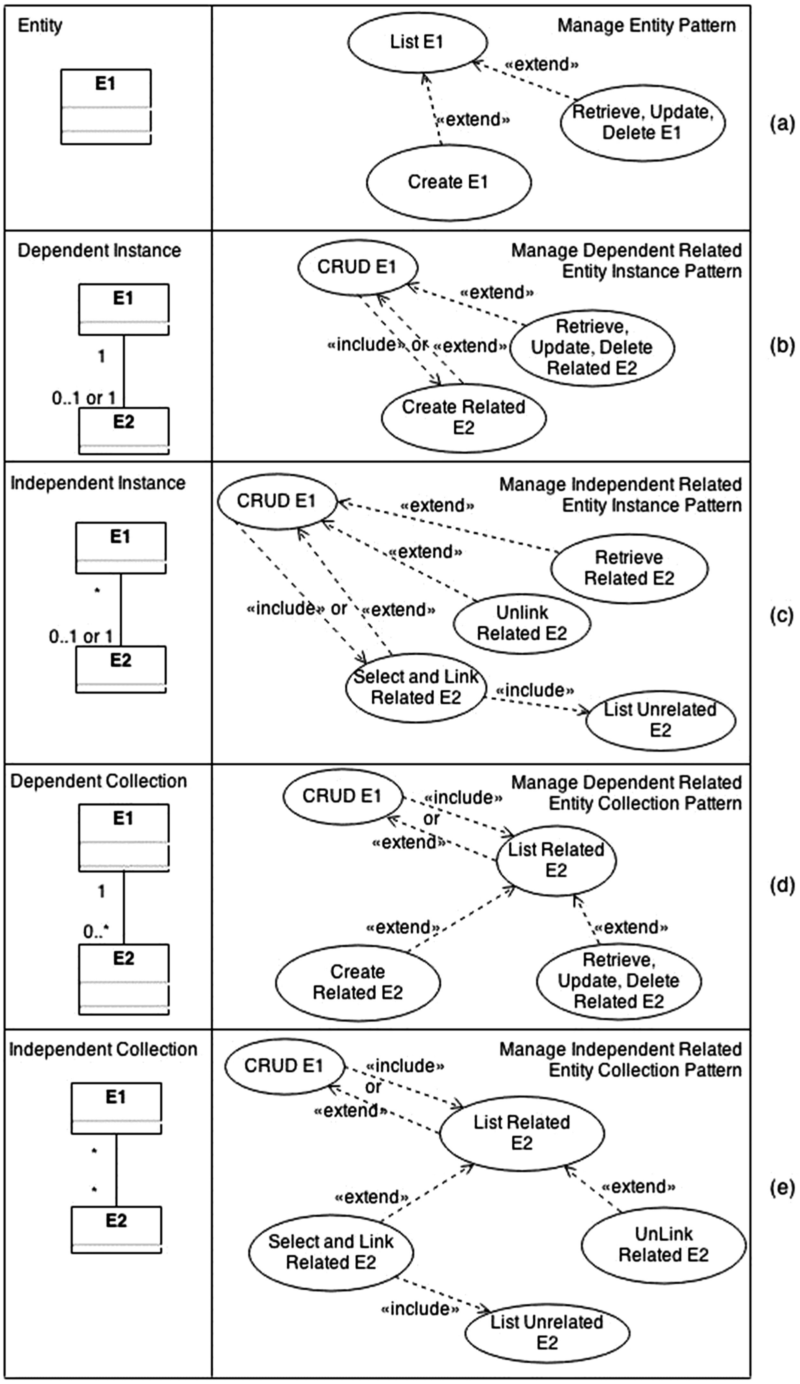

Fig. 4. Use case patterns and the corresponding appropriate patterns in the domain model.

These two use cases are available from within the use case that allows creating or editing the instance of E1 (CRUD E1, in Fig. 4(b)).

"Manage dependent related entity instance" is, therefore, a use case pattern comprising the three use cases referred to above, where CRUD E1 references instance E1, in the case of a "zero-or-one to one" association between E2 and E1, and it needs to reference E1 and E2, in the case of a "one to one" association between the two instances.

The other two use cases need to reference, as subject, both instance E1 and E2, because, creating or updating E2 always demands a related E1.

3.3 Manage Independent Related Entity Instance

An independent related entity instance is an instance of an entity E2 that has a "one to many" or a "zero-or-one to many" association with E1.

Managing the instance of E2 associated to a given instance of E1 typically involves linking (Select and Link Related E2, in Fig. 4(c)) or unlinking (Unlink Related E2) an existing instance of E2, or simply retrieving its information (Retrieve Related E2). These three use cases are available from within the use case that allows creating or editing the instance of E1 (CRUD E1, in Fig. 4(c)).

Use case "Select and Link Related E2" includes a use case for listing existing instances of E2 not related to the instance of E1 being managed (List Unrelated E2).

As a result, "Manage independent related entity instance" is a use case pattern comprising the five use cases referred to above, where CRUD E1 references an instance of E1, in the case of a "zero-or-one to many" association between E2 and E1, and it needs to reference E1 and E2, in the case of a "one to many" association between the two instances.

The other use cases need to reference instances of both E1 and E2, because, creating or updating E2 may imply a related instance of E1.

3.4 Manage Dependent Related Entity Collection

Dependent related entities are the instances of an entity E2 that have a mandatory "to one" association to E1. Managing the collection of instances of E2 associated to a given instance of E1 typically involves listing all or some of the existing related instances, and selecting one of those instances for editing (retrieving its information for visualizing, updating or deleting it), or creating a new related instance.

"Manage dependent related entity collection" is, hence, a use case pattern comprising four use cases where use cases for creating a new related instance (Create Related E2, in Fig. 4(d)) and editing existing related instances (Retrieve, Update, Delete Related E2) extend the use case for listing existing related instances (List Related E2), which in turn extends or is included in a use case where E1 is managed (CRUD E1).

3.5 Manage Independent Related Entity Collection

Independent related entities are the instances of an entity E2 that have an optional shared "to one" or "to many" association with E1. Managing the collection of instances of E2 associated to a given instance of E1 typically involves listing all or some of the existing related instances, and selecting one of those instances retrieving its information or unlinking it, or selecting an existing unrelated instance of E2 and link it to E1.

"Manage independent related entity collection" is, so, a use case pattern comprising five use cases where use cases for selecting and linking a related instance (Select and Link Related E2, in Fig. 4(e)) and unlinking existing related instances (Unlink Related E2) extend the use case for listing existing related instances (List Related E2), which in turn extends or is included in a use case where E1 is managed (CRUD E1). Also, use case "Select and Link Related E2" includes a use case for listing existing instances of E2 not related to the instance of E1 being managed (List Unrelated E2).

4 UI Metamodel and Modeling Notation

Use cases comprise behaviors that occur within interactions between a user playing the role of an actor (user role) and the system. An Interaction occurs within an interaction space. As seen in Sect. 2, interaction spaces may be specified through a User Interface Model, which, just as the use case and domain models we have been addressing, shall be defined in a platform independent manner.

As we are focusing on data-oriented applications, and these typically exhibit form-based user interfaces, the following subsections present a metamodel for developing form-based abstract user interface models (AUIM) at a platform independent level, and the concrete notation for modeling UIs according to the defined metamodel.

4.1 A Metamodel for User Interface Modeling

For enabling the construction of an AUIM, a metamodel for form-based UI modeling is proposed in this subsection. The proposed metamodel (see Fig. 5), extends UML by importing its packages, and is an evolution of the metamodel presented in [5, 11], which has been refined and simplified.

An interaction space (*InteractionSpace* in the figure) is an abstract UI space where interaction between a human actor (user role) and the system takes place, in the context of a use case. An *InteractionSpace* is composed of *InteractionBlocks*, which may contain a set of *DataAIO* elements. Both interaction spaces and interaction blocks may contain *ActionAIO* objects, which may navigate to another interaction space, trigger operations on the user interface (e.g.: *CancelOp*), or execute domain operations, which are behaviors associated to the use cases whose interactions take place within that interaction space, or methods of the domain entities belonging to the subject of those use cases (e.g.: CRUD operations).

An *InteractionBlock* is associated to one entity class (entity), from the domain model, and may be optionally associated to another class (master_entity) associated with the former, enabling master-detail information in an interaction space, provided

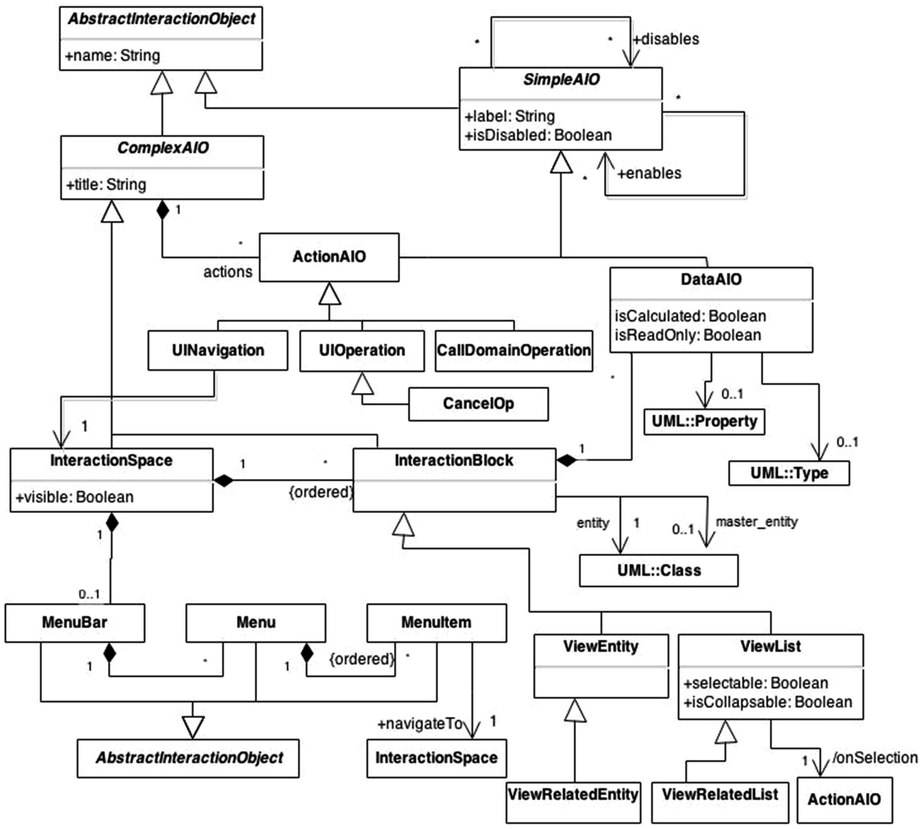

Fig. 5. User interface metamodel.

that in the same interaction space another interaction block is associated to the latter entity as its mandatory entity.

An *InteractionSpace* may contain a menu bar, composed of menus that aggregate menu items, each one of these allowing the navigation to another interaction space.

An *InteractionBlock* has four specializations:

- *ViewEntity,* which represents a *form* associated to an entity in the domain model;
- *ViewList,* which represents a *list* associated to an entity in the domain model;
- *ViewRelatedEntity,* representing a *form* associated to two entities in the domain model that have a dependent instance ("one to one" or "one to zero-or-one") or independent instance ("many to one" or "many to zero-or-one") relation between them, the one side entity being the master_entity and the other side entity being the main entity;
- *ViewRelatedList,* that represents a *list* associated to two entities in the domain model that have a dependent collection ("one to many") relation between them, the one side entity being the master_entity and the other side entity being the main entity, or

an independent collection (“many to many”), being either one of them the master_entity and the other one, the main entity.

An *InteractionBlock* may contain *DataAIO*s, which may have a type and may be associated to properties in the domain model entities. A *SimpleAIO* (*DataAIO* or *ActionAIO*) may enable or disable other *SimpleAIO*s when interacted with.

4.2 UI Model Concrete Notation

In this subsection, a concrete notation for abstract UI models that conform to the previously presented metamodel, is proposed.

Note that the concrete presentation of the final UI is not the goal of an AUIM. This way, the concrete notation for the AUIM shall be simple and leverage the aspects we want to address within this kind of models, that include the user interface interaction spaces, their contents, the relation between UI elements, navigation between spaces or action elements and the behavior triggered by them.

Table 1 shows the proposed concrete symbols for the UI modeling concepts defined in the proposed UI metamodel. The language symbols are borrowed from CAP.

Table 1. Relation between concrete symbols from CAP and the UI modeling language concepts proposed in the metamodel.

CAP symbol	**CAP meaning**	**Proposed Metaclass Identified by symbol**	**Constraints**
□	container	InteractionSpace	
□	container	InteractionBlock	
□	container	ViewEntity / ViewRelatedEntity	
⊟	collection	ViewList / ViewRelatedList	
↗⊟	Selectable collection	ViewList / ViewRelatedList	selectable = true
↗☆	input/accepter	DataAIO	
↗□	editable element	DataAIO	isCalculated = true
⊟	element	DataAIO	isReadOnly = true
➚	action/operation	ActionAIO	
✗	delete, erase	CancelOp (UIOperation)	
➚	start/go/to	UINavigation	
⤢	perform (&return)	CallDomainOperation	

5 UI Patterns for Use Case Patterns' Interactions

Just as data-oriented systems typically include the use case patterns identified in Sect. 3, which can be used in constructing the use case model, also their UIM typically includes a set of UI patterns in which the use case patterns' interactions take place.

Figure 6 presents the previously identified use case patterns and the corresponding UI patterns. The presented UI patterns comprise a set of *InteractionBlocks*, which may be in the same or different *InteractionSpaces*.

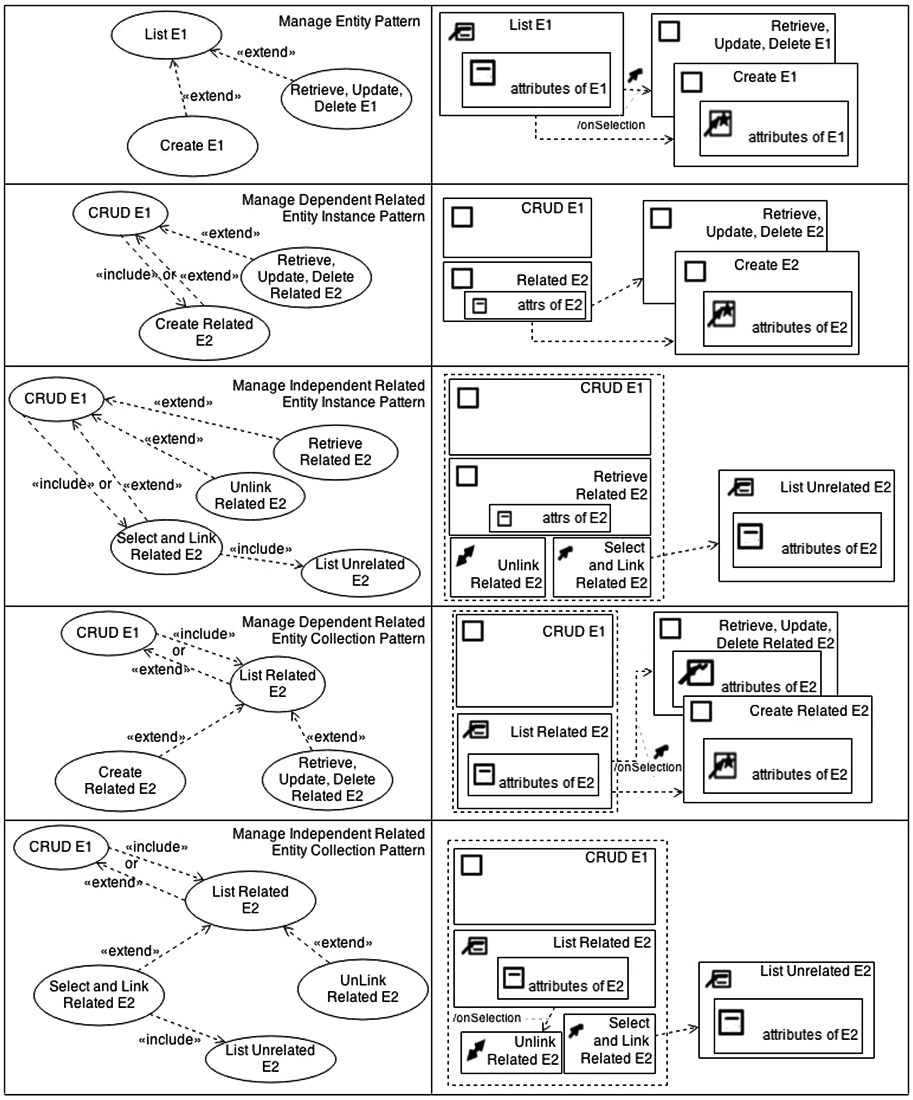
Manage Entity Pattern
List E1
«extend»
Retrieve, Update, Delete E1
Create E1
attributes of E1
/onSelection
Manage Dependent Related Entity Instance Pattern
CRUD E1
«include» or «extend»
Retrieve, Update, Delete Related E2
Create Related E2
Related E2
attrs of E2
Retrieve, Update, Delete E2
Create E2
attributes of E2
Manage Independent Related Entity Instance Pattern
Retrieve Related E2
Unlink Related E2
Select and Link Related E2
«include»
List Unrelated E2
Manage Dependent Related Entity Collection Pattern
List Related E2
Manage Independent Related Entity Collection Pattern
UnLink Related E2

Fig. 6. Use case patterns for data-centered applications and the corresponding UI patterns.

The UI pattern for the “Manage Entity” use case pattern comprises a *ViewList* that lists instances of domain entity E1, from where a *ViewEntity* block for creating new instances of E1, or a *ViewEntity* block for retrieving, updating or deleting an existing instance of E1, may be accessed. Both of these *ViewEntity* blocks will contain *DataAIO*s related to the attributes of E1. *ViewList* for listing instances of E1 may contain *DataAIO*s related to all attributes of E1 or to attributes used for unique identification of instances by the users (e.g.: stereotyped as «ident», as proposed in [11]).

UI pattern corresponding to the “Manage Dependent Related Entity Instance” use case pattern comprises, besides the *ViewEntity* for performing CRUD operations on an instance of E1, a *ViewEntity* for displaying the identification attributes of the related instance of E2, which will contain *UINavigation ActionAIO*s for allowing navigating to *ViewEntity* blocks for creating a new related instance of E2 or for retrieving, updating or deleting the existing related instance of E2.

UI pattern corresponding to the “Manage Independent Related Entity Instance” use case pattern comprises, besides the *ViewEntity* for performing CRUD operations on an instance of E1, a *ViewEntity* for retrieving and displaying all or the identification attributes of the related independent instance of E2. A “Select and Link Related E2” *UINavigation ActionAIO* will allow navigating to a *ViewList* block for selecting and linking a new related instance of E2. An *ActionAIO* is also present for performing the domain operation of unlinking the existing related instance of E2.

Besides the *ViewEntity* for performing CRUD operations on an instance of E1, the UI pattern corresponding to the “Manage Dependent Related Entity Collection” use case pattern, comprises a *ViewList* for displaying all or the identification attributes of the related instances of E2. On selecting an instance of E2 from the list, a *UINavigation ActionAIO* will allow navigating to an interaction space containing a *ViewEntity* block for retrieving, updating or deleting an existing related instance of E2. Other *UINavigation ActionAIO*s will allow navigating to a *ViewEntity* block for creating a new related instance of E2.

UI pattern corresponding to the “Manage Independent Related Entity Collection” use case pattern comprises, besides the *ViewEntity* for performing CRUD operations on an instance of E1, a *ViewList* for displaying all or the identification attributes of the related independent instances of E2. A “Select and Link Related E2” *UINavigation ActionAIO* will allow navigating to a *ViewList* block for selecting and linking new related instances of E2. An *ActionAIO* is also present for performing the domain operation of unlinking an existing related instance of E2.

6 Demonstration Case

This section shows a demonstration case to illustrate some of the use case patterns, and the corresponding user interface patterns.

Figure 7 shows the partial domain and use case models for a car rental system. The use case model in the figure has three blocks marked, corresponding to three previously identified use case patterns: (A) “Manage Entity”, (B) “Manage Dependent Related Entity Collection”, and (C) “Manage Independent Related Entity Instance”.

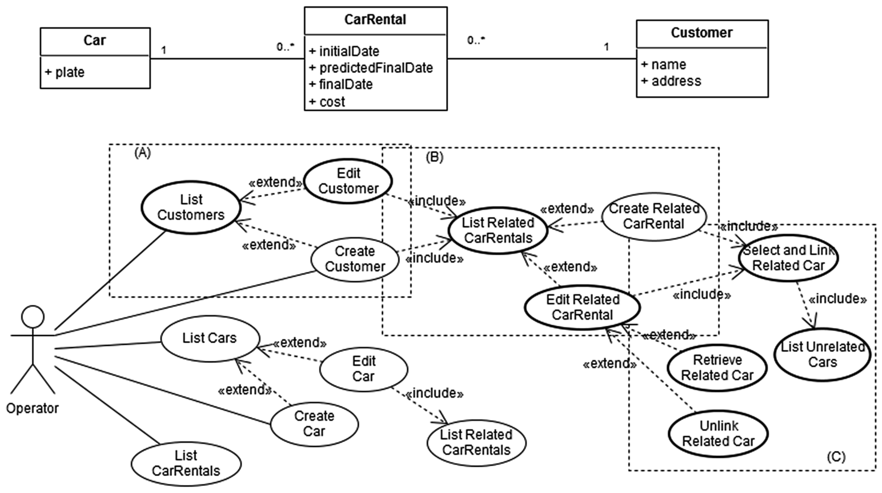

Fig. 7. Car rental partial domain and use case models.

The UI model excerpt corresponding to the bold use cases in the three blocks marked in the use case model is illustrated in Fig. 8. It starts with an interaction space "List Customers" with a selectable *ViewList* associated to Customer, which lists the instances of Customer. Selecting a customer from the list triggers the navigation to an interaction space with two interaction blocks: a *ViewEntity* with editable *DataAIO*s, associated to Customer, and a *ViewRelatedList* associated to CarRentals, with Customer as master entity. The selection of a car rental navigates to interaction space "Edit Related CarRental", that has a *ViewEntity* with read only *DataAIO*s displaying the

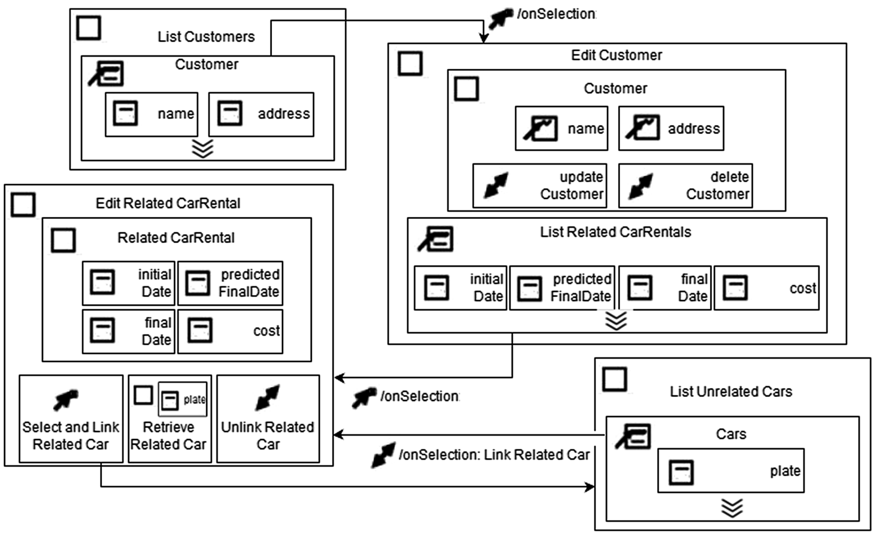

Fig. 8. Partial UI model for the car rental example.

selected car rental details. From there, the user may unlink the Car from the CarRental or navigate to another space with a *ViewList* listing the cars unrelated to the selected car rental, where the selection of a car triggers its linking to the previously selected CarRental.

7 Related Works

A few works that relate abstract UI elements with use cases have been proposed. Radeke *et al.* [12] propose an approach that interactively generates an abstract UI model, and then a concrete UI, by applying UI-patterns to elements of UI sub-models (e.g. task models). The approach is based on the manual selection of patterns, from a repository, that drives the UI model construction.

Costa *et al.* [13] combine CAP and task models to build abstract UI models, and from there obtain concrete Web Interfaces. The approach is based on the specification of a UI model comprising an abstract presentation model, a dialog model and a task model, together allowing the generation of a final concrete UI.

Martínez *et al.* [14] present a methodology for deriving UIs from early requirements existing in an organization's business process model. Their approach involves building a use case model and specifying each use case normal and alternative scenarios, which are then enriched with UI related information. The UI enriched scenario specifications are used in the generation of graphic components of the interface.

Elkoutbi *et al.* [15] propose formalizing use cases through a set of UML collaboration diagrams, each corresponding to a use case scenario. Each collaboration diagram message is manually labeled with UI constraints, from which it will then automatically produce message constraints with UI widget information. Elkoutbi's approach is then able to derive UI standalone prototypes for each interface object defined in the domain model.

Elkoutbi *et al.* and Martinez *et al.* approaches are able to produce a UI from the structural, use case and UI behavioral models, but demand the attachment of UI related information (input/output fields and/or widgets) to the use case detail specification, respectively collaboration diagrams and message sequence charts.

None of the surveyed approaches is restricted to data-oriented applications, but they all demand building a complete UIM. By restricting ourselves to data-oriented systems, our approach enables the generation of an UIM, from a system's use case model, easing the process of constructing the UIM. This is made by identifying patterns in use case models and generating the corresponding UI patterns in the UIM. Our approach also provides a concrete UI modeling language that enables modifying and completing the generated UIM, especially for the parts of the use case model that do not form an identifiable pattern and for which a UIM portion is not, consequently, generated.

8 Conclusions

This paper is based on the assertion that a system model has four views: structural view, where informational requirements are modeled through a domain model; functional view, where system functionality, and the user roles that may access it, are modeled through a use case model; user interface view, modeling the spaces where interaction within use cases take place, and a behavioral view, modeling the system behaviors or behavioral constraints.

Use cases, then, control which roles (actors) may access which system functionality, and hold the functionality that is executed in the context of an interaction, through execution traces, by instantiating available behaviors. Use cases may, then, be seen as providing services to the UI, which are based on the CRUD or user defined operations distributed as behavioral features in the system state (the domain model).

This view allowed us to propose the modeling of the abstract UI for use case interactions, based on a proposed UI metamodel and corresponding concrete notation. We also proposed a set of use case patterns and the corresponding UI patterns, which may be used when modeling data-oriented systems. These relations between use case and UI patterns enable the pattern based generation of the abstract UIM from the use case and domain models, as proposed in [5, 11], and for which a model-transformation prototype has been built [5]. In this setting, the proposed UIM concrete language allows the abstract UIM modification after its generation and before generating concrete UIMs for different target platforms [5].

Ongoing work, in the context of project Amalia (**A**gile **M**odel-driven **A**pp**LI**c**A**tion Development Method and Tools), aims at developing a modeling tool for the integrated modeling of the domain, use case and user interface model views of a system. The domain and use case models may be, however, developed using any UML tool, as the UML alone provides the needed mechanisms to associate domain entities and use cases, namely tagged values. In fact, a UML profile can be defined as a convenient and lightweight means, associated with a UML modeling tool, of building the domain and use case models. For developing the UIM, an appropriate modeling tool needs to be constructed, though. And that is one of the ongoing Amalia project's goals.

References

1. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison Wesley, Reading (1998)
2. Frankel, D.S.: Model Driven Architecture - Applying MDA to Enterprise Computing. Wiley Publishing Inc., Indianapolis (2003)
3. Cruz, A.M.R., Faria, J.P.: Automatic generation of user interface models and prototypes from domain and use case models. In: Proceedings of the ICSoft 2009, Sofia, Bulgaria, vol. 1, pp. 169–176. INSTICC Press (2009)
4. OMG: OMG Unified Modeling Language (OMG UML), version 2.5 (2013). http://www.omg.org/spec/UML/2.5/Beta2/
5. Cruz, A.M.R.: Automatic generation of user interfaces from rigorous domain and use case models. Ph.D. dissertation, FEUP, University of Porto, Portugal (2010)
6. Cruz, A.M.R.: A pattern language for use case modeling. In: Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (Modelsward 2014). INSTICC Press, Lisboa, Portugal, January 2014

7. Pinheiro da Silva, P.: User interface declarative models and development environments: a survey. In: Paternó, F. (ed.) DSV-IS 2000. LNCS, vol. 1946, pp. 207–226. Springer, Heidelberg (2001)
8. Dix, A., Finlay, J., Abowd, G., Beale, R.: Human-Computer Interaction, 2nd edn. Prentice Hall, Upper Saddle River (1998)
9. Constantine, L.L.: Canonical abstract prototypes for abstract visual and interaction design. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) DSV-IS 2003. LNCS, vol. 2844, pp. 1–15. Springer, Heidelberg (2003)
10. OMG: Action language for foundational UML (Alf) - concrete syntax for a UML action language, version 1.0.1 (2013)
11. da Cruz, A.M.R., Faria, J.P.: A metamodel-based approach for automatic user interface generation. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 256–270. Springer, Heidelberg (2010)
12. Radeke, F., Forbrig, P., Seffah, A., Sinnig, D.: PIM tool: support for pattern-driven and model-based UI development. In: Coninx, K., Luyten, K., Schneider, K.A. (eds.) TAMODIA 2006. LNCS, vol. 4385, pp. 82–96. Springer, Heidelberg (2007)
13. Costa, D., Nóbrega, L., Jardim Nunes, N.: An MDA approach for generating web interfaces with UML concurtasktrees and canonical abstract prototypes. In: Coninx, K., Luyten, K., Schneider, K.A. (eds.) TAMODIA 2006. LNCS, vol. 4385, pp. 137–152. Springer, Heidelberg (2007)
14. Martinez, A., Estrada, H., Sánchez, J., Pastor, O.: From early requirements to user interface prototyping: a methodological approach. In: International Conference on ASE 2002, pp 257–260 (2002)
15. Elkoutbi, M., Khriss, I., Keller, R.K.: Automated prototyping of user interfaces based on UML scenarios. J. Autom. Softw. Eng. **13**(1), 5–40 (2006)