A PATTERN LANGUAGE FOR USE CASES SPECIFICATION

ALBERTO RODRIGUES DA SILVA, INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
DUŠAN SAVIĆ, SINIŠA VLAJIĆ, ILIJA ANTOVIĆ, SAŠA LAZAREVIĆ, VOJISLAV STANOJEVIĆ, MILOŠ MILIĆ, Faculty of Organizational Sciences, University of Belgrade

Use cases describe a set of interactions between actors/users and the system under study. These interactions should be described textually according to some style and template to be simultaneously readable, consistent and verifiable. The main reason for these qualities is that use cases specification should be used both by business analysts and requirements engineers, when specifying functional requirements, as well as by software developers, when designing and implementing the involved system functionality. In spite of the popularity of use cases, there are not many patterns and guidance to help produce use cases specifications in a systematic and high-quality way. Furthermore, in the context of Model Driven Development (MDD) approaches, use cases specifications can be also considered models with precise syntax and semantics to be automatically validated or used in model-to-model or model-to-text transformations.

In this paper we propose a pattern language to improve the quality of use cases specifications. These patterns are concrete guidelines that have emerged from the research and industrial experience of the authors throughout the years, particularly in the designing of MDD languages and approaches and in applying them in concrete applications. These patterns are interconnected among themselves and are the following: (P1) DEFINE USE CASE TYPES, (P2) KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL, (P3) DEFINE USE CASE WITH DIFFERENT SCENARIO AND INTERACTION BLOCK TYPES, and (P4) DEFINE USE CASE WITH DIFFERENT ACTION TYPES.

Key Words: Use cases, Requirements Specification, Requirements Patterns

1. INTRODUCTION

Some years ago Alexander Christopher noted that a **pattern** *"describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"* [1]. On the other hand, Gamma et al. defined design patterns as "*descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*" [2]. From these definitions it is understandable that patterns have two important parts: a problem and a solution, where the solution can be reused several times in different problem domains. Still, Alexander pointed out that a pattern is *"at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing*" [1]. Coplien told a similar idea: "*a pattern is the rule for making the thing, but it is also, in many respect, the thing itself*" [3].

The book *Design Patterns: Elements of Reusable Object-Oriented Software* [2] launched an avalanche of best practices and share knowledge in the world of software engineering and put the design patterns in the center of software design and development in a very practical way. It is one of the most

EuroPLoP '15, July 08-12, 2015, Kaufbeuren, Germany
© 2015 ACM. ISBN 978-1-4503-3847-9/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2855321.2855330

important source for understanding object-oriented design theory and practice. Up to now, many of these and other design patterns were made and used in the development of numerous software systems. A **design pattern** is primarily seen as a generic solution that can be applied several times in different problem and situations. In addition, design patterns provide great flexibility in the program during its maintenance and upgrades. Design patterns belong to solution space patterns that are mainly used by developers to decide on the system design and code structure [2, 4]. But, for many years the concept of patterns was adopted by researchers and practitioners and used not only in design and implementation as well as in other software disciplines, such as requirements engineering or testing engineering.

Requirements engineering (RE) is a part of the software development process that traditionally includes two main processes [5]: requirements development (with tasks such as elicitation, analyze, specification, and validation of requirements) and requirements management. RE is a discipline where business and technical requirements need to be gathered and refined using various analysis techniques before being verified by the stakeholders and finally specified. The importance of RE in software engineering has been documented and extensively discussed in the literature. Errors produced at this stage, if undetected until a later stage of software development, can be very costly as reported in several studies [42, 43].

When the concept of patterns emerged, their primary focus was on the solution space. A trend for considering patterns in the problem space has slowly but steadily risen [6]. Therefore, requirements patterns have gained popularity to help users in identifying, analyzing and structuring requirements of a software system. A **requirements pattern** is defined as a "*guide for writing a particular type of requirement*" [7], or "*a reusable experience based framework that aids a requirements engineer write or model better quality requirements in the least possible time*" [6]. The major goals of requirements patterns are [6]: (1) guide the analysts to understand the problem; (2) provide a common framework to define requirements with which software products can be better evaluated, designed, built and tested; and (3) be able to trace the design of the system back to the original business objectives. The application of patterns is important in all requirements activities but mainly in requirements specification [8]. In particular, the proposed patterns should be used by requirements engineers, business analysts or developers to solve their problems more effectively.

There are different formats for representing requirements patterns. Generally, requirements patterns follow a general template format with a structure such as [21]: name, also known as, author, problem that define the intent of the pattern, context that describes valid uses of the pattern, forces that arise when the pattern is applied, solution, applicability that describes how and when it can be applied, classification, known uses, examples, and related patterns. However, the patterns proposed in this paper follow a simpler **pattern template format** with the following key elements: pattern name, context, problem, solution, examples, consequences, related patterns, and known uses.

The proposed pattern language is primary applicable to the context of user requirements specification of information system, and consists in a coherent set of the following rules: (P1) DEFINE USE CASE TYPES, (P2) KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL, (P3) DEFINE USE CASE WITH DIFFERENT SCENARIO AND INTERACTION BLOCK TYPES, and (P4) DEFINE USE CASE WITH DIFFERENT ACTION TYPES.

2. BACKGROUND

During this last decade a trend of approaches has emerged considering models not just as documentation artifacts, but as central artifacts in the software engineering process. Beyond the benefits of facilitating and sharing a common and coherent vision of the system under study, models also allow – through complex techniques such as meta-modeling, model transformation, code generation or model interpretation – the creation or automatic execution of software systems based on those models. Several of those approaches have been classified as **Model-Driven Development (MDD)** and have been mainly focused on the requirements, analysis and design, and implementation disciplines [22][23][24]. To better support the software development life cycle's activities – ranging from requirements specification to the use of generative programming techniques –, we need to specify requirements in a more rigorous way, not just informal text specifications but also more formal text or graphic-based models. To manage these challenges we have been involved in several research projects and initiatives as we briefly introduce in the following paragraphs. The pattern language proposed in this paper results from these research initiatives and, consequently, are seen in those involved languages and tools.

2.1 Authors Background

2.1.1 ProjectIT Approach

The goal of ProjectIT is to provide a complete software development workbench with support for project management, requirements engineering, analysis, design, and code generation activities [25][26][27]. ProjectIT-Requirement is the component of the ProjectIT architecture that deals with requirements engineering. The main goal of the ProjectIT-Requirements is to develop a textual language for the definition and documentation of requirements, which, by raising their rigor, facilitates the reuse of models that might be used in MDD approaches. Taking into account the different types of requirements this project focus mainly in software requirements, as these can more easily be transformed into software design models into ProjectIT's languages such as XIS [28] or XIS-Mobile [29][30]. (Further information is available at https://github.com/MDDLingo).

2.1.2 SilabMDD approach

SilabMDD approach [31] emerged as a key result of Silab Project which was initiated in 2007 in the Software Engineering Laboratory at Faculty of Organizational Sciences, University of Belgrade. The main goal of this project was to enable automated analysis and processing of software requirements in order to achieve automatic generation of different parts of a software system. In the beginning, Silab Project has been divided in two main sub-projects SilabReq and SilabUI that were being developed separately. Initially this SilabReq project focused on the formalization of user requirements and their transformations to different UML models to facilitate the analysis process and to assure the quality of software requirements. On the other hand, SilabUI project focused on automatic generation of user interfaces based on use cases specification. When both subprojects reach desired level of maturity, they integrated in a way that some results of SilabReq project can be used as input for SilabUI project. As a proof of concept, the Silab has been used for the Kostmod 4.0 [32] project, which was implemented for the needs of the Royal Norwegian Ministry of Defense. After several years of using Silab in developing multiple intensive software systems we defined the SilabMDD approach.

Usually, in MDD the implementation is (semi-)automatically generated from models. Despite the fact that use cases are narratives, there is not a single standard that specifies what a textual specification of use case should be. In SilabMDD approach we develop SilabReq DSL language that should be used for use case specification. It requires a rigorous definition of the use case specification, particularly description of sequences of action steps, pre- and post-conditions, and relationships between use case models and domain models. SilabMDD approach is use case driven approach but it does not pay much attention to the way in which use cases are elicited. They can be derived from business processes or from plain text requirements. If requirements are expressed in some form of model as in RSLingo using with RSL-IL (see more below) it is possible to automatically using appropriate transformation to deliver use cases. Throughout the specification process use cases are specified using SilabReqUC DSL language and continuous inspection business conceptual model. For business conceptual model description we developed the small SilabReqBCM DSL language. Action in use cases as well as pre-condition and post-condition are specified in context of business conceptual models [33].

2.1.3 RSLingo Approach

A couple of years ago the ProjectIT-Requirements evolved to a more flexible approach named RSLingo [34]. RSLingo is a linguistic approach for improving the quality of requirements specification, based on two languages and on the mapping between them: the RSL-PL and the RSL-IL. RSL-PL (Pattern Language) [35] is an extensible language for defining linguistic patterns to be used with natural language processing and text extraction tools to automatically produce RSL-IL specifications from requirements written in natural language. On the other hand, RSL-IL (Intermediate Language) [36] is a formal language with a fixed set of constructs for representing and conveying a large number of RE-specific concerns. Recently, and still in the context of the RSL-IL language, we introduced the problem of combinatorial effects based on the evidence of many dependencies that explicitly or implicitly exist among the elements commonly used on system requirements specification (SRS) [39]. We proposed and discussed a set of practical recommendations to help defining a SRS template that may better prevent (to some extent) that combinatorial effects problem [40]. Currently RSL-IL language is extended for integrating privacy requirements [41], and is supported by different tools, namely a MS-Excel template and an Xtext-based editor. (Further information is available at https://github.com/RSLingo).

2.2 Withall's Requirements Patterns

Stephen Withall defined 37 requirements patterns in his book "Software Requirement Patterns" [7]. The objective of these extensive number of requirement patterns is to guide the requirements engineers on how to specify common types of requirements, to make it quicker and easier to write them, and to improve the quality of them. These requirement patterns are applied at the level of an individual requirement meaning they are mainly *linguistics patterns* and pragmatic guidance to how better write these sentences.

Withall's requirements patterns are divided into eight domains: Fundamental, Information, Data Entity, User Function, Performance, Flexibility, Access Control, and Commercial requirements patterns. Whitall also proposed the following structure for defining these software requirements patterns [7]: (1) basic details with information about domain, author, classification and, if exists, related patterns; (2) the context in which it can be applied; (3) discussion; (4) content that describes what is necessary to state that type of requirement; (5) template; (6) some examples; (7) extra requirements; (8) how to use the pattern for implementation purposes; and (9) how to use the pattern for testing purposes.

2.3 Cruz's Pattern Language for Use Case Modeling

António Cruz identified the most common use case patterns found on use case models of data oriented applications (i.e., information systems), namely the following *use case patterns* [44]: Manage an entity instance, Manage dependent related entity instances, Manage independent related entity instances, Manage dependent related entity collections, and Manage independent related entity collections. These patterns are well-recognized from the different types of relationships (e.g., master, master-detail or master-reference relationships) established from the entities defined at a domain model level and also from the common CRUD operations found on these entities.

In addition Cruz proposed a pattern language for facilitating the use case modeling with a fine grained use cases (without overcrowding the model with many use cases), but also without losing the relation to the standard UML, because the proposed use case pattern language constructs intermingled with the standard UML use case notation, as every construct can be converted to a standard UML use case pattern and vice-versa [44].

Comparing with our pattern language, Cruz's is focused on the use case modeling while ours is on use cases specification. Both proposals recommend the alignment and consistency between use cases and domain models and, in that way, they can be complementary. However, our proposal emphasizes the both aspects of modeling and textual specification of requirements (inside the oval) while Cruz's emphasizes only the modeling aspects (at the oval level).

3. OVERVIEW OF THE PATTERN LANGUAGE

Software-intensive systems are a particular class of systems whose essential functionalities and qualities are realized by software. According to Pohl, there are two classes of software-intensive systems [5]: (1) *information systems*, that collect, store, transform, transmit, and process information; and (2) *embedded software-intensive systems*, in which the software is part of the system and it is strongly integrated with hardware. Nakatani et al. [38] emphasized that *business application systems* or *information systems* mainly perform operations such as storage, retrieval, updating and deleting of information. These characteristics point out that most common use cases can be classified into several categories such as: data storage, data retrieval, data updating, data creation, documents creation or document transmission. In their research, 53 out of 58 use cases were classified into these six categories. They also developed tools [38] that show basic use case patterns, such as: Create, Read, Update, Delete and Reports-related, to help developers write down use cases and manage the relationships between use cases and domain objects (or entities) to keep consistency between requirements. These patterns are expected to cover over 80% of simple use cases of business application systems. Similar results were also obtained from our Kostmod 4.0 project [32].

This pattern language is primary applicable to the context of user requirements specification of information system. Figure 1 shows the proposed patterns and respective relationships, namely:

(P1) DEFINE USE CASE TYPES,

(P2) KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL,

(P3) DEFINE USE CASE WITH DIFFERENT SCENARIO AND INTERACTION BLOCK TYPES, and

(P4) DEFINE USE CASE WITH DIFFERENT ACTION TYPES.

The numbers in the arrows represent the logical order of application these patterns and also how they are presented in the following sections.

These patterns should preferentially be used together. However, in simpler situations you may only need to adopt the first two patterns: (P1) DEFINE USE CASE TYPES and (P2) KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL. In this situation you only have to define for each use case what should be its type and identify the respective domain entity.

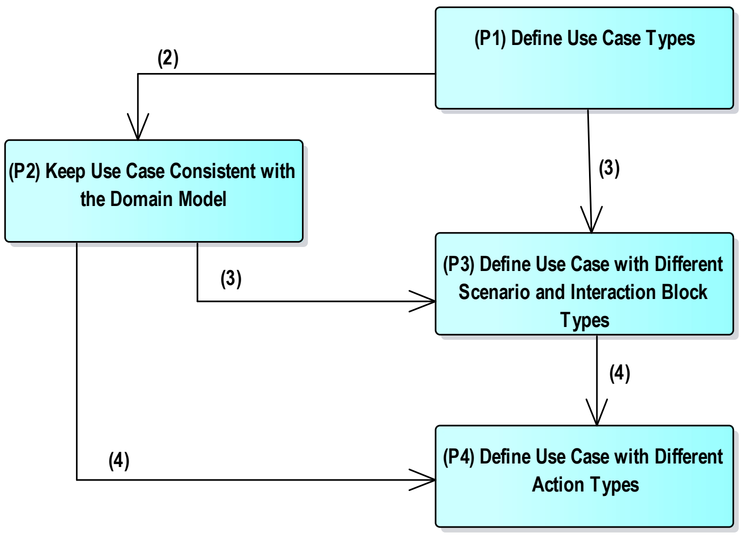

(P1) Define Use Case Types
(2)
(P2) Keep Use Case Consistent with the Domain Model
(3)
(3)
(P3) Define Use Case with Different Scenario and Interaction Block Types
(4)
(4)
(P4) Define Use Case with Different Action Types

Fig. *1. Overview of the pattern language for use cases specification.*

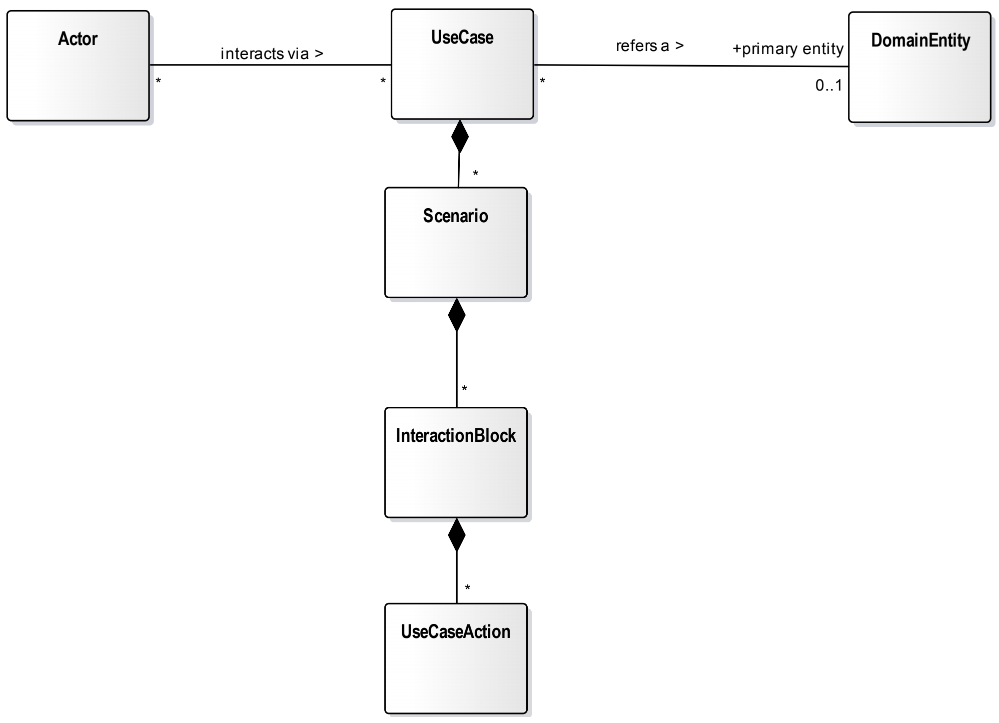


Fig. 2. The conceptual model underlying the pattern language.

Bearing in mind that use cases describe interactions between actors/users and the system under study, to better define these interactions consider use cases specifications as dialog conversations between users and the system. To specify more clearly the goal of use cases it should be recommended to define a use case type for each use case (see DEFINE USE CASE TYPES). Use case should be defined in the context of a domain model, in a way that during the use case specification you can continuously inspect and check the consistency between the use case model and the domain model (see KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL).

The use case's user-system interactions are defined by scenarios supported by one or more interaction blocks (see DEFINE USE CASE WITH DIFFERENT SCENARIO AND INTERACTION BLOCK TYPES). Each *Interaction block* defines one or more *Actor request block* and one *System response block*. Each user-system interaction should be described with a formal notation that requires the adoption of different types of use case actions (see DEFINE USE CASE WITH DIFFERENT ACTION TYPE).

Figure 2 shows the conceptual model underlying these four patterns. Use case *(*UseCase*)* is the key concept and is categorized with a type (UseCaseType), that, for example, can be: create, view, update, delete or search. One or more actors (Actor) interact with a use case. In the scope of information systems the purpose of a use case is to access or manage data entities, and so each use case is related to some domain entities *(*DomainEntity), i.e. an object/entity defined in a domain model. A use case may be defined by one or more scenarios (Scenario). A scenario has a type (ScenarioType) such as: main (MainScenario), alternative (AlternativeScenario), exception (ExceptionScenario). A use case scenario is defined by a set of use case actions (UseCaseAction). These actions are classified in two categories: the actions performed by the actor (ActorAction) and the actions performed by the system (SystemAction). These actions may be organized in interaction blocks *(*InteractionBlock) to better cluster the user-system interactions. Each interaction block has a type (InteractonBlockType) such as: persistent, retrieval, select, and customer-action. Depending on the UseCaseType a use case may contain different types of InteractionBlocks.

4. PATTERNS

The example used to support the discussion of the proposed patterns is based on the *"Billing system" case study* as briefly and partially described below:

The **Billing system** is a business information system to support the management of customers, products and invoices of any organization. This system should provide configuration features and should facilitate the work of administrative managers and operators, i.e. it should allow controlling the organization's invoices and respective payments as well as produce several reports and analytical dashboards. From the requirements specification purpose, the Billing system can be divided into four subsystems, namely:

(1) Customer management subsystem.

(2) Product management subsystem.

(3) Invoice management subsystem, which should include creating invoices, searching and updating existing invoices, printing, sending, exporting invoices and tracking customer's purchases.

(4) General system configuration subsystem, which should include configuring the user of the system, configuring VAT taxes, etc.

[…]

Figure 3 shows a simple version of the Billing system’s domain model, clearly identifying its key concepts and relationships: Customer, Invoice, Invoice-Line, VAT-Category and Product.

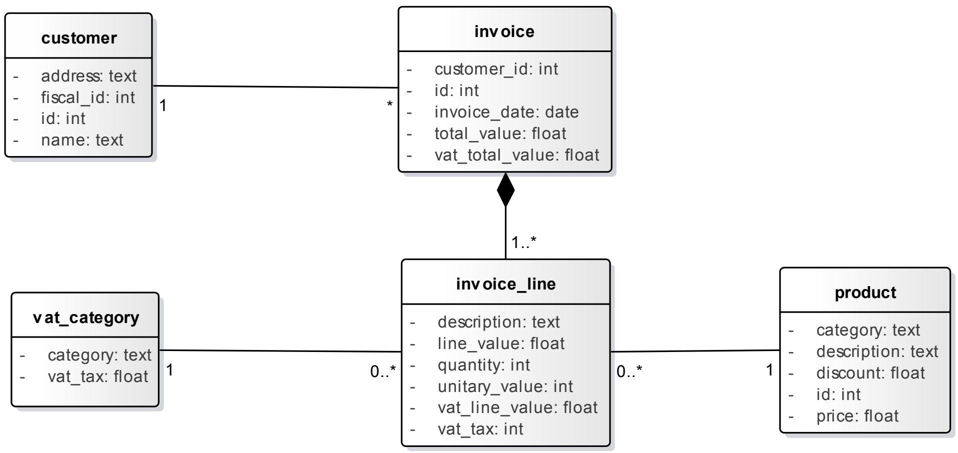


Fig. 3. Billing system’s domain model.

These patterns are described following the pattern template referred in the end of Section 1, namely by considering the following elements: pattern name, also known as, context, problem, solution, example, consequences, related patterns and known uses.

4.1 DEFINE USE CASE TYPES Pattern

Context: You are a requirements engineer, a business analyst or even a developer and you frequently have to specify the requirements of an information system. You want adopt the technique of use cases as your favorite approach to specify these user requirements. You notice that the specification of different information systems based on use cases technique share similar context. This pattern is appropriate when the objective of these use cases is to access and manage data entities, which is very common in information systems.

Problem: You are always in doubt about how to specify and organize your use cases. After specifying several use cases you realize that some use cases emerge and are very similar. However, you do not know how to define and reuse them systematically. Some questions appear from the problem stated, such as: Is it possible to categorize these use cases? What are the most common types? How to make use cases reusable? Is it possible to create templates for scenarios or actions and later add specificity to these general actions?

Solution: As suggested in Figure 4, first, you should define a predefined set of use case types. The majority of use cases can be classified either as **entity-create**, **entity- search**, **entity-view**, **entity-update** or **entity-delete**, or some combination of these types, e.g., **entity-manage**. We may also consider other types of use cases, such as **entity-browse**, **entity-report**, **entity-dashboard**, **entity-import**, **entity-export**, **entity-sync**, or even a general **customer** type. Another good starting for this identification is to research the use case content patterns as discussed by Martin Langlands [18]. Second, you should categorize every use case according to this set of types. Third, you should relate each use case with a primary entity defined in a complementary domain model (as discussed in the pattern KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL).

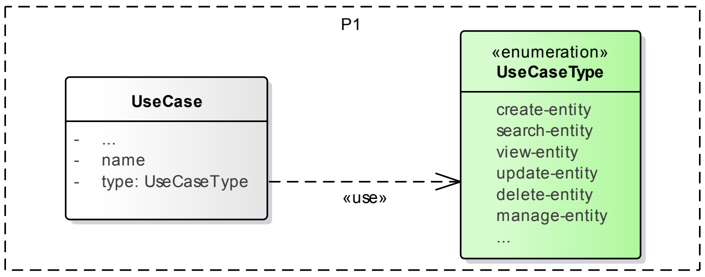


Fig. 4. DEFINE USE CASE TYPES pattern: conceptual solution.

Example: Figure 5 shows a use case model where each use case is categorized by a respective type.

Consequence: As a result of this pattern each use case becomes clearly classified by a type and then prepared to be assigned to a specific domain entity. This is the first step to ensure the consistency between the use case and the domain model (see KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL), and then to define appropriate interaction blocks (see DEFINE USE CASE WITH INTERACTION BLOCKS).

Related Patterns: OBJECT MANAGER, and KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL.

Known Uses: SilabMDD approach follows directly this pattern. XIS and XIS-Mobile languages apply this pattern by classifying use cases based on predefined Boolean tag values (e.g., CreateMaster, ReadMaster, UpdateMaster, CreateDetail, CreateReference).

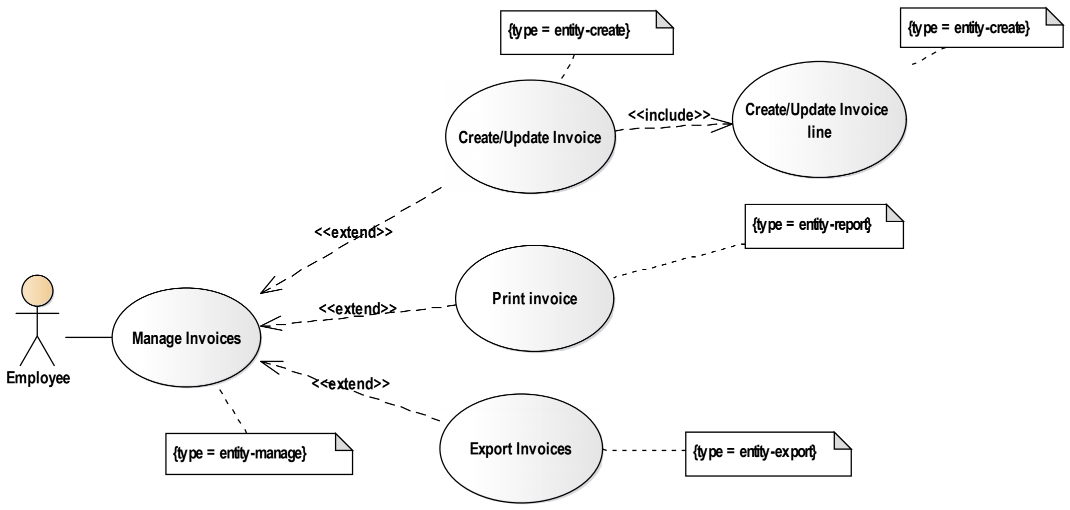


Fig. 5. Use cases specification with use case types.

4.2 KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL Pattern

Context: Use cases are used for identifying and specifying functional requirements. On the other hand, a ***domain model*** provides the foundation for the understanding of the underlining (domain) problem and for a clear understanding of the main concepts of the system under study. Consequently, it is a good practice to define such domain model to better support the system requirements specification, for example, in order that the same concepts and terms are referred consistently throughout the specification of multiple use cases.

Problem: A use case encapsulates both structural and behavioral elements. The structure of a use cases is not separated from its behavior; they are intertwined (overlapped) in use case actions, use case pre-conditions and post-conditions. Use case actions describe the use case behavior, and at the same time they hide its structure because the structure encapsulated in the use case is a part of the domain model. The problem is how to properly ensure this consistency bearing in mind that there are many use cases specified with different level of detail, throughout a requirements specification.

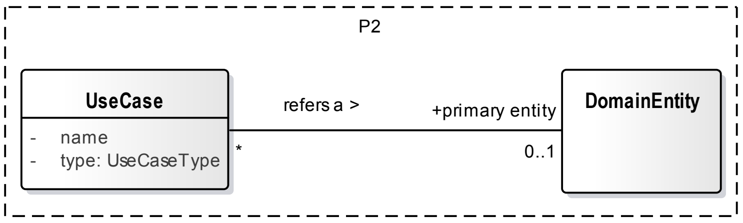


Fig.6. KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL pattern: conceptual solution.

Solution: As suggested in Figure 6, to ensure the consistency between the use cases and the domain model, each part of the use case must be related with the entities defined at that domain model. Each use case should be related to just one primary domain entity, meaning that its behavior is directly related to that domain entity (in the example of the Figure 7 that domain entity is *Invoice* and *Invoice-Line*). That means that all entities that are referenced by any use case action should be related with that entity (*Customer* and *Invoice-Line* entities in the Billing system example). Everything that is specified in use cases should follow some rules. For example, an *input rule* must define which data information the user needs to enter when he wants to create a new invoice. This rule should be specified in the context of the domain model.

Example-A: Figure 7 shows a use case model where each use case is categorized by a respective type and is assigned to a primary domain entity.

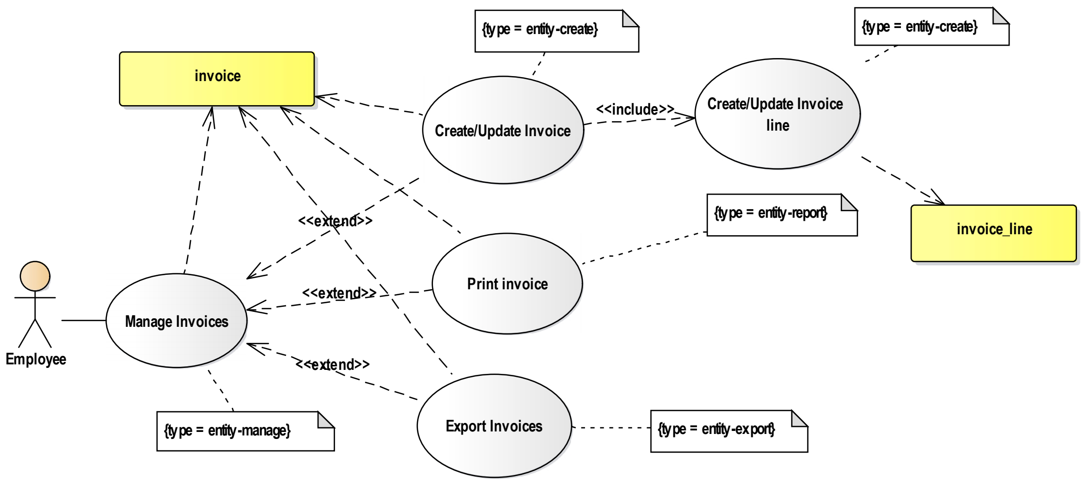
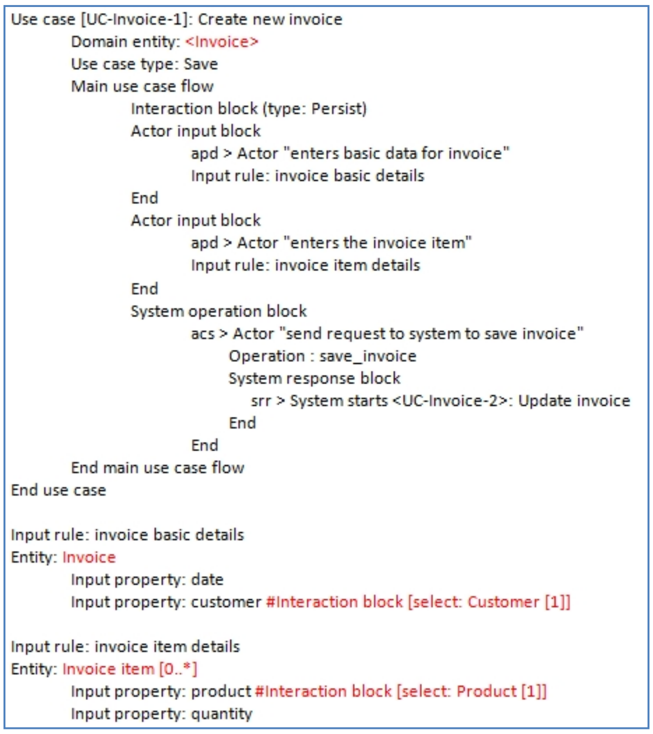


Fig. 7. Use cases specification with use case types and primary domain entities assigned.

Example-B: Figure 8 shows a textual specification of the use case "*Create new invoice"* and its relationship with the (partial) domain model based on the SillabReq language. Each input rule is bound to a *domain entity* and also specifies the type of domain object. For example, the input rule *"Invoice basic detail"* is bound to the Invoice domain entity. Each *input property* should be bound to a *domain entity attribute*. This information has an impact on the system operation that the system should provide so must be somewhere specified.


```
Use case [UC-Invoice-1]: Create new invoice
        Domain entity: <Invoice>
        Use case type: Save
        Main use case flow
                Interaction block (type: Persist)
                Actor input block
                        apd > Actor "enters basic data for invoice"
                        Input rule: invoice basic details
                End
                Actor input block
                        apd > Actor "enters the invoice item"
                        Input rule: invoice item details
                End
                System operation block
                        acs > Actor "send request to system to save invoice"
                             Operation : save_invoice
                             System response block
                                srr > System starts <UC-Invoice-2>: Update invoice
                             End
                        End
        End main use case flow
End use case

Input rule: invoice basic details
Entity: Invoice
        Input property: date
        Input property: customer #Interaction block [select: Customer [1]]

Input rule: invoice item details
Entity: Invoice item [0..*]
        Input property: product #Interaction block [select: Product [1]]
        Input property: quantity
```

Fig 8 "Create new invoice" use case (partial) specification.

Consequence: As a result of the application of this pattern the consistency between use cases and the involved domain model is kept. In addition, both models are defined in parallel resulting in more consistent and complete models. Finally, this pattern also implies that the resulting use cases become defined rigorously and can be used in the context of MDD approaches allowing, for example, model-to-model or model-to-text transformations.

Related Patterns: DEFINE USE CASE TYPES.

Known Uses: SilabMDD approach follows directly this pattern. XIS and XIS-Mobile languages apply this pattern by introducing the concept of "business entities" which are higher-level entities that aggregate one or more domain entities, with just one entity performing as the "master" and other entities with "detail" ou "reference" roles.

4.3 DEFINE USE CASE WITH DIFFERENT SCENARIO AND INTERACTION BLOCK TYPES Pattern

Context: User-system interactions need to be clear enough to be readable and understandable by all stakeholders. On one hand, they should be readable by non-technical stakeholders that can just read and verify them but, on the other hand, they should include enough details for developers to implement the required system operations, for testers to generate or make automated tests, and for user-interface designers to design or generate user-interface prototypes.

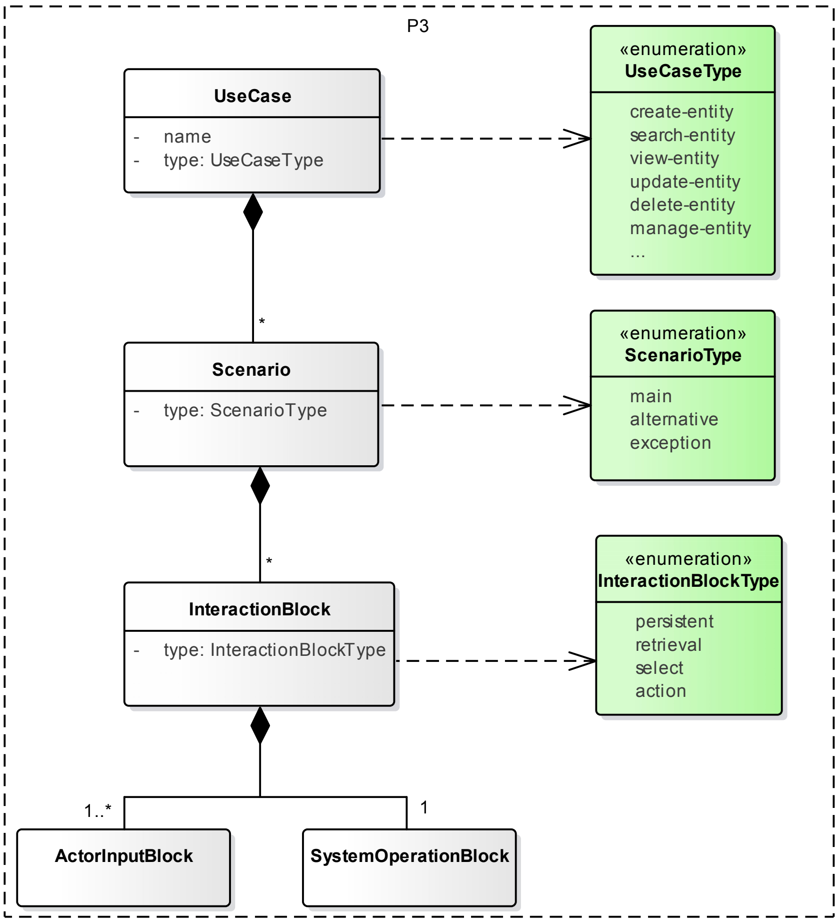


Fig. 9. DEFINE USE CASE WITH DIFFERENT SCENARIO AND INTERACTION BLOCK TYPES pattern: conceptual solution.

Problem: User-system interactions are usually done through graphical user-interfaces. Therefore, graphical user-interfaces are used by the actor to send request to system to execute a system's operation, as well as, for the system to present results from the execution of system operation. How can we describe this interaction, so that we can separate information related to the user interface from information that is just important to the system's operation, but also to ensure the consistency between them?

Solution: As suggested in Figure 9, the solution is to specify the interactions between the actors and the system using an ***interaction block.*** In that way, a use case scenario should include one or more interaction blocks**.** In addition, an interaction block is composed of several actor input blocks and one system operation block which is associated with a particular system response block.

An **actor input block** includes one or more *ActorPrepareData* actions. Each *ActorPrepareData* action should be bound to particular input rule that describe information that user enters during interaction.

A **system operation block** includes one *ActorCallsSystem* action. Each system operation block is associated with a system response block which specifies the result of system operation execution.

After the execution of the system operation the system returns some results to the actors. In case that the actor sends request to the system to execute:

(1) System operation which type is C, D, or U, system response is appropriate use case which type is view or update (see *DEFINE USE CASE TYPES*). For example: In use case *"Create new invoice"* if use send request to system to execute system operation *"create invoice"*, the system response is use case *"Update invoice"* (update last saved invoice) or response can be use case "*Manage invoices"* (manage all invoices)

(2) System operation which type is R, system response is a respective *"Retrieval"* interaction block

The following types of interaction blocks are defined:

(1) **Persistent interaction block**. This block describes a user-system interaction in which user send request to system to create, delete or update a domain object. Therefore, this interaction block contains zero or more *ActorPrepareData* actions and one *ActorCallsSystem* action.

(2) **Retrieval interaction block**. This block describes user-system interaction when user requires from system to retrieve (find) some domain object or objects (for example search customer). The extension of *Retrieval interaction block* is *Select interaction block***.**

(3) **Select interaction block.** This block presents extension of *Retrieval interaction block* used to describe the interaction between user and system when user wants, after searching for a domain object, to select one or more domain entities and put them available in another use case (for example in use case *"Create new invoice"* user need to select customer for invoice, therefore, actor start some select interaction flow to find customer, and put him on the invoice (select)). This interaction block can be used in conjunction with *Action interaction block*. After selection of some domain object or object actor can request to system to execute some operation (for example, when user select customer, actor can request from system to show details with existing invoices).

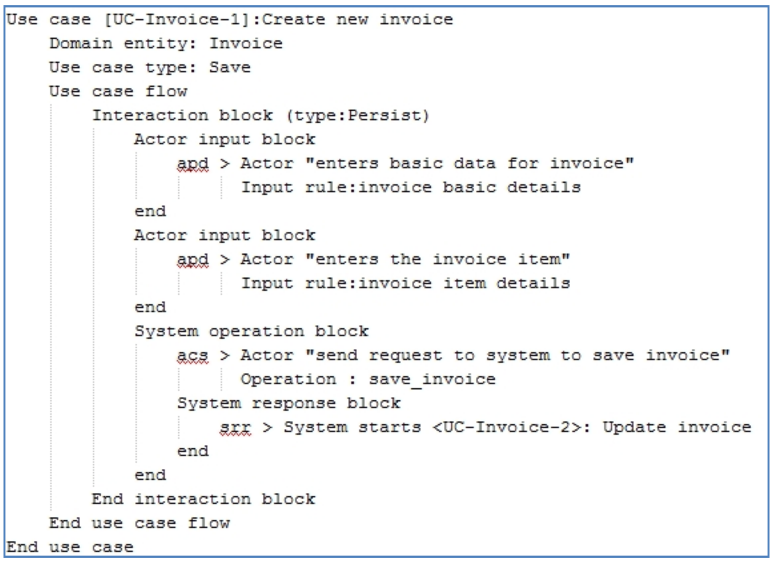
(4) **Action interaction block.** This block is used to describe all possible system operation that actor can request from system on some domain entity object. This block can be used in all types of use cases. This block can be used in description as alternative action block for *System operation block.*

Table 1 presents the constraints between the types of interaction blocks and the types of use cases. Of course this table can be extended with other types of use cases that may be considered.

Table 1. Interaction block types

		INTERACTION BLOCK TYPES			
		PERSISTENT	RETRIEVAL	SELECT	ACTION / SELECT WITH ACTION
USE CASE TYPES	UC: Search		+		+
	UC: Create	+		+	+
	UC: View		+		+
	UC: Update	+	+	+	+
	UC: Delete		+		+

Example: Figure 10 presents the specification of the use case *"Create new invoice"* according to the appropriate interaction blocks. This use case contains one interaction block *(*type *Persistent),* with two blocks *Actor input block* and one *System operation block*. We have two *Actor input block* because they are related to different domain objects. The first one is bound to *Invoice* object, while the second one is bound to *Invoice item* object (see KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL).



```
Use case [UC-Invoice-1]:Create new invoice
    Domain entity: Invoice
    Use case type: Save
    Use case flow
        Interaction block (type:Persist)
            Actor input block
                apd > Actor "enters basic data for invoice"
                          Input rule:invoice basic details
            end
            Actor input block
                apd > Actor "enters the invoice item"
                          Input rule:invoice item details
            end
            System operation block
                acs > Actor "send request to system to save invoice"
                          Operation : save_invoice
                System response block
                    srr > System starts <UC-Invoice-2>: Update invoice
                end
            end
        End interaction block
    End use case flow
End use case
```

Fig.10. "Create new invoice" use case (partial) specification.

Consequence: As a result of the application of this pattern we clearly define system required behavior as a set of system operation that system should provide. Specification of use case at this level also provide ability to constantly check consistency between use case and domain model (see KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL), as well as a good basis for specifying the details of user interfaces in a platform-independent way.

Related Patterns: DEFINE USE CASE TYPES, KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL.

Known Uses: SilabMDD approach follows directly this pattern.

4.4 DEFINE USE CASE WITH DIFFERENT ACTION TYPES Pattern

Context: The integration of use cases in a MDD approach requires rigorous specifications. This means that use case specification should be considered as a model with precise syntax and semantic. On the other hand, use cases as a technique for user requirements specification is popular because they are very useful and readably. Despite the fact that use cases describe user-system interactions, other approaches such as task modeling are used independently or in conjunction with use cases for describing these interactions.

Problem: Different types of actions exist for describing use case scenarios. The problem is that the semantics of these actions is not always clearly defined resulting in a vague level of use case specification. The specification of these actions can range from very informal textual description to very formal, from description of user intention to description of user task and details of user interface. Surely, this depends on the analysts experience and skills as well as on their goals.

Solution: As suggested in Figure 11, use case specification should consider a predefined set of use case action types. Define that set of types to better specify the syntax and the semantics of each use case action. These actions may be classified in two categories: the actions performed by the actor (*ActorAction*), and the actions performed by system (*SystemAction*) [37].

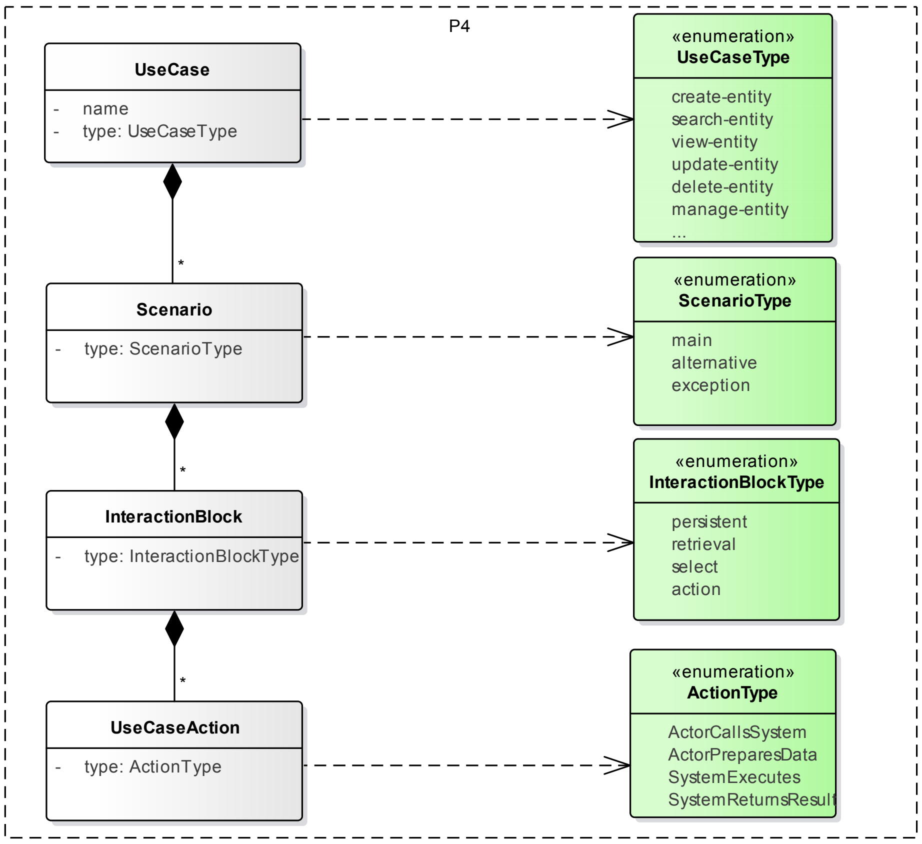


Fig. 11. DEFINE USE CASE WITH DIFFERENT ACTION TYPES pattern: conceptual solution.

These categories may be further subdivided into subcategories. For example, in the category of actions performed by the user, there are the following subcategories: (1.1) actor prepares data for system operation (*ActorPreparesData*) and (1.2) actor calls system to execute system operation (*ActorCallsSystem*).

On the other hand, in the category of actions performed by the system, there are the following subcategories such as: (2.1) system executes system operation (*SystemExecutes*) and (2.2) system returns to the user the result of the system operation execution (*SystemReturnsResult*).

Each *ActorPreparesData* action should be bound to a particular input rule that defines the data that user enters. *ActorCallsSystem* action should be specified in a *system operation block*. Each *SystemExecutes* action can be further categorized according the CRUD pattern, and for each of these actions can also specify the respective *SystemReturnsResult* action.

As already mentioned above, each interaction block contains multiple actor input blocks and just one system operation block. An actor input block may contain several *ActorPreparesData* actions. On the other hand, a system operation block contains just one *ActorCallsSystem* action corresponding to the intended system action. Table 2 summarizes the possible combinations between the different types of use case interaction blocks and the use case actions.

Table 2. Summary of use case actions in the context of the interaction blocks.

Types of use case interaction block	**Interaction block**		
	Actor input block	**System operation block**	
		Mandatory	**Optional**
Persistent	*ActorPreparesData (zero or more)*	*ActorCallsSystem*, related to one create, update or delete operation	More (retrieval system operations)
Retrieval	*ActorPreparesData (one or more)*	*ActorCallsSystem*, related to one retrieval system operations	More (retrieval system operations)
Select	*ActorPreparesData (one or more)*	*ActorCallsSystem*, related to one retrieval system operations	More (retrieval system operations)
Action	NO	*ActorCallsSystem*, related to one create, update or delete system operation	More (create, update or delete system operation)

Example: Figure 10 presents the specification of the use case *"Create new invoice"* with the detail of interaction blocks. This use case contains one interaction block *(*type *Persistent),* with two blocks *Actor input block* and one *System operation block*. In this example, the persistent interaction block contains two *ActorPreparesData* actions, and the system operation block that defines the *create* system action (system operation: *save_invoice*).

Consequence: Use cases should be defined in a clear and precise manner. This requires additional level of rigorous with predefined types and notation for clearly identify use case actions. Different types of use case actions can be defined in the context of different use case interaction blocks, but not all of them should exist in every interaction blocks. Hence, this pattern also allows defining the correct relationships that should exist between interaction blocks and use case actions, depending on their respective types.

Related Patterns: PROPERTY LIST, FILED LIST; DEFINE USE CASE TYPES, KEEP USE CASE CONSISTENT WITH THE DOMAIN MODEL.

Known Uses: SilabMDD approach follows directly this pattern.

5. CONCLUSION

To produce requirements specifications of information systems with high quality, these specifications need to be constantly checked for consistency, completeness and correctness. However, checking requirements based on use cases, usually written in natural language, is a very difficult task. In spite of that, writing use cases to be readable by users and other stakeholders can be improved with concrete guidance and patterns such as the ones proposed in this paper.

In this paper we propose a cohesive set of patterns that makes that possible: Keeping the consistency between use cases model and the domain model is more important when dealing with intensive information systems. The first step for reusing use cases is to identify types (such as create, view, update, delete, search) or to define abstract classes of use cases at a meta or high-level of abstraction. Interactions between users and the system should be defined as user-system dialogs and structured as scenarios and interaction blocks. Considering that these interactions are usually done through graphical user interface, specification of user interfaces could also be defined in the scope of each use cases specification as a complementary model.

Use case specifications using this pattern language are being supported by some approaches and tools that are being developed and put publicly available to the community [28, 29, 31].

ACKNOWLEDGEMENTS

This work was partially supported by Portuguese funds through FCT – Fundação para a Ciência e a Tecnologia, under the projects CMUP-EPB/TIC/0053/2013, UID/CEC/50021/2013 and DataStorm Research Line of Excellency funding (EXCL/EEI-ESS/0257/2012), and by Ministry of Education, Science and Technological Development of the Republic of Serbia, grant number 174031. Finally, thanks to David West, our EuroPLoP'2015 shepherd, for his relevant criticism and suggestions that helped to improve the paper.

REFERENCES

[1] CHRISTOPHER, A., ISHIKAWA, S., SILVERSTEIN, M., JACOBSON, M., FIKSDAHL-KING, I., ANGEL, S., A Pattern Language. Oxford University Press, New York, 1977.
[2] GAMMA, E., HELM,R., JOHNSON, R., VLISSIDES,J., Design Patterns : Elements of Reusable Object Oriented Software, Addison Wesley Professional, 1994.
[3] COPLIEN, J., Software Patterns, SIGS, 1996.
[4] BUSCHMANN F, MEUNIER R, ROHNERT H, SOMMERLAD P & STAL M., Pattern Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996.
[5] POHL, K., Requirements Engineering - Fundamentals, Principles, and Techniques. Springer, 2010.
[6] MAHENDRA, P., & GHAZARIAN, A., Patterns in the Requirements Engineering: A Survey and Analysis Study, 2014.
[7] WITHALL, S. Software Requirements Patterns. Microsoft Press, 2007.
[8] CHENG, B., ATLEE, J. Research Directions in Requirements Engineering, Future of Software Engineering (FOSE '07), pp. 285 – 303, 2007.
[9] TOVAL, J.A., NICOLÁS, J., MOROS, B., GARCIA. F., Requirements Reuse for Improving Information Systems Security: A Practitioner's Approach, Requirements Engineering, 6(4), pp.205-219, 2002.
[10]WAHONO, R. S. , CHENG, J., Extensible Requirements Patterns of Web Application for Efficient Web Application Development, International Symposium on Cyber Worlds (CW), 2002.
[11]ROBERTSON, S., Requirements Patterns Via Events/Use Cases. PLoP, 1996.
[12]DURÁN, A., BERNÁRDEZ, B., RUÍZ, A., TORO, M., A Requirements Elicitation Approach Based in Templates and Patterns, 1999.
[13]MOROS, B., VICENTE, C., TOVAL, A., Metamodeling Variability to Enable Requirements Reuse, EMMSAD, 2008.
[14]J. YANG, L. LIU, Modeling Requirements Patterns with a Goal and PF Integrated Analysis Approach, COMPSAC, 2008.

[15]FRANCH,X., PALOMARES,C., QUER,C., RENAULT,S., LAZZER, F., A Metamodel for Software Requirement Patterns, REFSQ 2010: 85-90, 2010.
[16]ADOLPH, S., BRAMBLE, P., COCKBURN, A., POLS, A., Patterns for Effective Use Cases. Addison Wesley, 2002.
[17]OVERGAARD, G., PALMKVIST, K., Use Cases: Patterns and Blueprints. Addison Wesley, 2005.
[18]LANGLANDS, M., Inside The Oval: Use-Case Content Patterns, Technical report, Planet Project, 2010. Accessed on 2015. http://planetproject.wikidot.com/use-case-content-patterns
[19]CHUNG, L., SUPAKKUL, S. 2006."Capturing and reusing functional and non-functional requirements knowledge: A goal-object pattern approach", IEEE International Conference on Information Reuse and Integration (IRI).
[20]ISSA, A. A. , AL-ALI, A. 2010"Use Case Patterns Driven Requirements Engineering", International Conference on Computer Research and Development (ICCRD)
[21]CHUNG, L., PAECH,B., ZHAO,L., LIU, L., SUPAKKUL.S. 2012., RePa Requirements Pattern Template, International Workshop on Requirements Patterns (RePa'12)
[22] STAHL, T., VOLTER, M., Model-Driven Software Development, Wiley, 2005.
[23] SELIC, B., Personal reflections on automation, programming culture, and model-based software engineering. Automated Software Engineering, 15(3-4): 379-391, 2008.
[24] SILVA, A.R., Model-Driven Engineering: A Survey Supported by a Unified Conceptual Model, in Computer Languages, Systems & Structures, Elsevier (to be published), 2015.
[25]SILVA, A.R., VIDEIRA, C., SARAIVA, J., FERREIRA, D., SILVA, R., The ProjectIT-Studio, an integrated environment for the development of information systems, In Proc. of the 2nd Int. Conference of Innovative Views of .NET Technologies (IVNET'06), SBC and Microsoft, 2006.
[26]SILVA, A. R., SARAIVA, J., FERREIRA, D., SILVA, R., VIDEIRA, C., Integration of RE and MDE Paradigms: The ProjectIT Approach and Tools, IET Software, IET, 2007.
[27]FERREIRA, D., SILVA, A.R., A Controlled Natural Language Approach for Integrating Requirements and Model-Driven Engineering, ICSEA, 2009.
[28]SILVA, A.R., SARAIVA, J., SILVA, R., MARTINS, C., XIS – UML Profile for eXtreme Modeling Interactive Systems, in Proceedings of MOMPES'2007, IEEE Computer Society, 2007.
[29]RIBEIRO, A., SILVA, A.R., XIS-Mobile: A DSL for Mobile Applications, Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC), 2014.
[30]RIBEIRO, A., SILVA, A.R., Evaluation of XIS-Mobile, a Domain Specific Language for Mobile Application Development, Journal of Software Engineering and Applications, 7(11), pp. 906-919, Oct. 2014.
[31]SAVIĆ, D., VLAJIĆ, S., LAZAREVIĆ, S., ANTOVIĆ, I., STANOJEVIĆ, V., MILIĆ, M., SILVA, A. R., SilabMDD: A Use Case Model Driven Approach, ICIST 2015 5th International Conference on Information Society and Technology, 2015.
[32]KOSTMOD4.0 http://rapporter.ffi.no/rapporter/2009/01002.pdf, accessed in January, 2013
[33]SAVIĆ, D., SILVA, A. R., VLAJIĆ, S., LAZAREVIĆ, S., ANTOVIĆ, I., STANOJEVIĆ, V., MILIĆ, M., Use Case Specification at Different Abstraction Level, Proceedings of QUATIC'2012 Conference, IEEE Computer Society, 2012.
[34]FERREIRA, D., SILVA, A.R., RSLingo: An information extraction approach toward formal requirements specifications, Proceedings of MoDRE'2012, IEEE Computer Society, 2012.
[35]FERREIRA, D., SILVA, A.R., RSL-PL: A Linguistic Pattern Language for Documenting Software Requirements, in Proceedings of RePa'13, IEEE Computer Society, 2013.
[36]FERREIRA, D., SILVA, A.R., RSL-IL: An Interlingua for Formally Documenting Requirements, in Proceedings of MoDRE, in the 21st IEEE International Requirements Engineering Conference (RE'2013), IEEE Computer Society, 2013.
[37]JACOBSON I. ET AL. Object-Oriented Software Engineering: A Use-Case Driven Approach, Addison-Wesley1992.
[38]NAKATANI, T., URAI, T., OHMURA, S., TAMAI, T., A requirements description meta-model for use cases, Eighth Asia-Pacific Software Engineering Conf. (APSEC'01), 2001.
[39]VERELST, J., SILVA, A.R., MANNAERT, H., FERREIRA, D., HUYSMANS, *Identifying Combinatorial Effects in Requirements Engineering*. In Proceedings of Third Enterprise Engineering Working Conference (EEWC 2013), Advances in Enterprise Engineering, LNBIP, May 2013, Springer.
[40]SILVA, A.R., VERELST, J., MANNAERT, H., FERREIRA, D., HUYSMANS, P., Towards a System Requirements Specification Template that Minimizes Combinatorial Effects, Proceedings of QUATIC'2014 Conference, IEEE Computer Society, 2014.
[41]CARAMUJO, J., SILVA, A.R., Analyzing Privacy Policies based on a Privacy-Aware Profile: the Facebook and LinkedIn case studies, Proceedings of IEEE CBI'2015, IEEE, 2015.
[42]THE STANDISH GROUP, Chaos Summary 2009 Report, The 10 Laws of Caos, 2009.
[43]Eveleens, L., Verhoef, C., The Rise and Fall of the Chaos Report Figures, IEEE Software, Jan/Feb, 2010.
[44]CRUZ, A.M., A Pattern Language for Use Case Modeling, Proceedings of MODELSWARD 2014, INSTICC Press, 2014.