PATTERNS FOR BETTER USE CASES SPECIFICATION

ALBERTO RODRIGUES DA SILVA, INESC-ID & Instituto Superior Técnico, Universidade of 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 some styles and templates 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, when specifying functional requirements, as well as by software developers, when designing and implementing the involved system functionality. In spite 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 as a model 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 set of patterns to help 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 these last years, particularly in the designing of MDD languages and respective approaches. These patterns are interconnected among themselves and are the following: (1) DEFINE USE CASE TYPES, (2) KEEP USE CASE AND DOMAIN MODEL CONSISTENT, (3) DEFINE USE CASE WITH DIFFERENT SCENARIO AND INTERACTION BLOCK TYPES, (4) DEFINE USE CASE WITH DIFFERENT ACTION TYPES, and (5) COMPLEMENT USE CASE WITH UI-PROTOTYPES.

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 "*pattern is, in short, 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 thing: "*a pattern it is the rule for making the thing, but it is also, in many respect, the thing itself"* [3]. As suggested in Figure 1, a pattern has both the structure of the problem and of the solution. Additionally, a pattern is a process that explains when and how the structure of the solution should be created from the structure of the problem.

1.1 Design Patterns

The book *Design Patterns: Elements of Reusable Object-Oriented Software* [2] has 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 important source for understanding object-oriented design theory and practice. Up to now, many of these and other design patterns are 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 in not only in design and implementation as well as in other software disciplines, both earlier and later, such as requirements engineering or testing engineering.

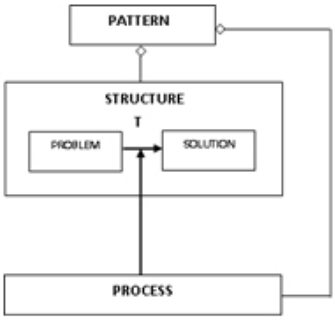

Fig. 1. Basic properties of a pattern: the structure and the process.

1.2 Requirements Patterns

The requirements engineering process is a part of the software development process. Requirements engineering involves two main sub-processes [5]: (1) requirements development (with tasks such as elicit, analyze, specify, and validate software requirements) and (2) requirements management. Requirements engineering is a discipline where business requirements that a system must satisfy need to be gathered, refined using various analysis techniques before being verified against stakeholder and finally specified. The importance of requirements engineering in software engineering has been documented from long ago. 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 requirement patterns are [6]: (1) guide the analysts to understand the problems; (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 the most important is in requirements specification. Since patterns encourage knowledge reuse (and consequently avoid "reinventing the wheel") they show great potential to build software better, faster and at low cost [8]. Requirements patterns involve the developer and end users and affect project managers, business stakeholders, architects and requirements analysts. Business analysts and requirements engineers use patterns to solve their problem more effectively.

Different approaches have proposed using patterns to achieve reuse during requirements engineering. Toval et al. define the SIREN method (Simple Reuse of software requiremENts) to requirements reuse, where a requirement pattern is defined as a sentence in natural language [9]. Wahono et al. define the concept of extensible requirement pattern for web applications [10]. Even other proposals, such as [7, 11, 13, 14], differ in criteria like the scope of the approach, the formalism used to write the patterns, the intended main use of patterns and the existence of an explicit meta-model [15]. The most cited books that discuss the use of patterns in use case modeling are [16, 17]. However, these books are more focused on the use case modeling and structuring than on the specification content of use cases. In this respect, Langlands [18] proposes a set of patterns that define detailed flows of interactions and system actions. Chung et al. [19] and Issa et al. [20] defined proposals for use case patterns as diagrams that can be reused in new software projects

1.3 Requirement Pattern Templates

There are different formats for representing requirement patterns. Generally, requirement patterns follow a general template format with a structure such as [21]: (1) name, (2) also known as, (3) author, (4) problem that define the intent of the pattern, (5) context that describe valid uses of the pattern, (6) forces that arise when the pattern is applied, (7) solution, (8) applicability that describe how and when it can be applied, (9) classification, (10) known uses, (11) examples, and (12) related patterns.

According to Withall there are three types of requirement patterns [7]: (1) functional, (2) pervasive, and (3) affects database. In addition, all of these 37 requirement patterns are divided into eight domains: (1) Fundamental requirement patterns, (2) Information requirement patterns, (3) Data Entity requirement patterns, (4) User Function requirement patterns, (5) Performance requirement patterns, (6) Flexibility requirement patterns, (7) Access Control requirement patterns, and (8) Commercial requirement patterns. Whitall also proposes the following structure of 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.

The patterns proposed in this paper follow a simpler pattern template with the following key elements: pattern name, context, problem, solution, examples, consequence, related patterns, and known uses.

2. AUTHORS BACKGROUND

During this last decade a trend of approaches has emerged considering models not just as documentation artefacts, but as central artefacts in the software engineering process. In addition to 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) approaches and have been mainly focused on the requirements, analysis and design, and implementation disciplines [22][23][24]. If we want 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 patterns proposed in this paper results from these research initiatives and consequently are seen in those involved languages and tools.

2.1 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 and models that might be used in MDD approaches. Taking into account the different types of requirements this project focus mainly in software requirements, those that can more easily be transformed into software design models into ProjectIT's languages such as XIS [28] or XIS-Mobile [29][30].

2.2 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 project SilabReq project focused on the formalization of user requirements and their transformations to different UML models to facilitate the analyses 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, Silab project 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 this project in developing different intensive software system we are established a SilabMDD approach.

Usually, in MDD the implementation is (semi) automatically generated from models. Despite the fact that use cases are narratives, there is no a single standard that specifies what 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. Bearing in mind that the model is expressed through a modeling language and SilabMDD approach uses several integrated DSL it is also language-oriented. SilabMDD approach is use case driven approach but it do not pay much attention to the way in which get use cases. It can be derived from business process, or 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 small SilabReqBCM DSL language. Action in use cases as well as pre-condition and post-condition are specified in context of business conceptual model. Except two of this language, SilabMDD approach use SilabReqUI DSL language which is primary use for specification user interface prototype [33].

2.3 RSLingo Approach

A couple of years agor 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 in two languages and 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 was extended for integrating privacy requirements [41], and has been supported by different tools, namely a MS-Excel template and an Xtext-based editor.

3. OVERVIEW OF THE SET OF PATTERNS

Software-intensive system is a kind of system whose essential functionalities and qualities are realized by software [5]. According to Pohl, there are two classes of software-intensive system: (1) information systems, that collects, stores, transforms, transmits, and processes information; and (2) embedded software-intensive systems, which the software is one part of 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 Making-reports, to help developers write down use cases and manage the relationships between use cases and domain objects to keep consistency between requirements. These patterns are expected to cover over 80% of simple use cases of business application systems. We obtained similar results in the previously mentioned Kostmod 4.0 project [32]. The proposed use case patterns are primary applicable to the context of user requirements specification of information system. Figure 2 overviews of proposed patterns, namely: (1) DEFINE USE CASE TYPES, (2) KEEP USE CASE AND DOMAIN MODEL CONSISTENT, (3) DEFINE USE CASE WITH DIFFERENT SCENARIO AND INTERACTION BLOCK TYPES, (4) DEFINE USE CASE WITH DIFFERENT ACTION TYPES, and (5) COMPLEMENT USE CASE WITH UI-PROTOTYPES.

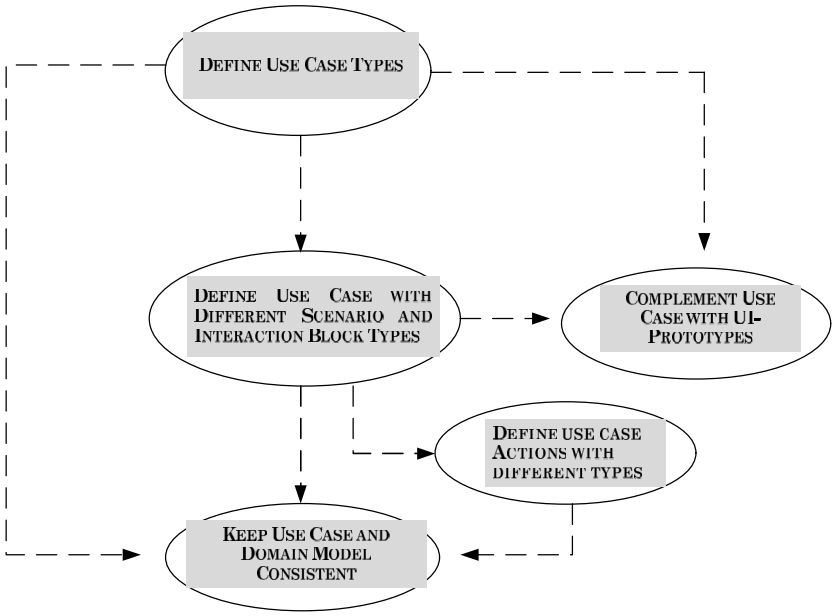

Fig. *2. Overview of the proposed patterns.*

Bearing in mind that use cases describe interactions between actors/users and the system under study, to better describe these interactions we 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 use case type for each use case (see DEFINE USE CASE TYPES). We named a scenario of user-system interactions as an interaction flow that contains one or more interaction blocks (see DEFINE USE CASE WITH DIFFERENT SCENARIO AND INTERACTION BLOCK TYPES). Each *Interaction block* contains one or more *Actor request block* and one *System response block*. An interaction between an user and a system should be described textually with a little formal notation

with different types of use case actions (see DEFINE USE CASE WITH DIFFERENT ACTION TYPE): for each *Actor request block* we recommend specifying actions such as: (1) actor prepares data for system operation (*ActorPrepareData* - APD) and (2) actor calls system to execute system operation (*ActorCallsSystem* - ACS) [31]. On the other hand, for each *System response block* we recommend specifying data that system returns to user as a result of the system operation execution (*SystemReturnsResult* - SRR). Use case actions should be described in the context of the domain model, in a way that during the use case specification we can continuously inspect and check the consistency between the use case model and the domain model (see KEEP USE CASE AND DOMAIN MODEL CONSISTENT). User interacts with the system through graphical user interfaces. Therefore, the specification of user interfaces should still be defined in the scope of use case specifications. This two models (use case model and user interface model) need to be considered as a whole complement (see COMPLEMENT USE CASE WITH UI-PROTOTYPES*)*. Figure 3 shows the conceptual model underlying the proposed patterns.

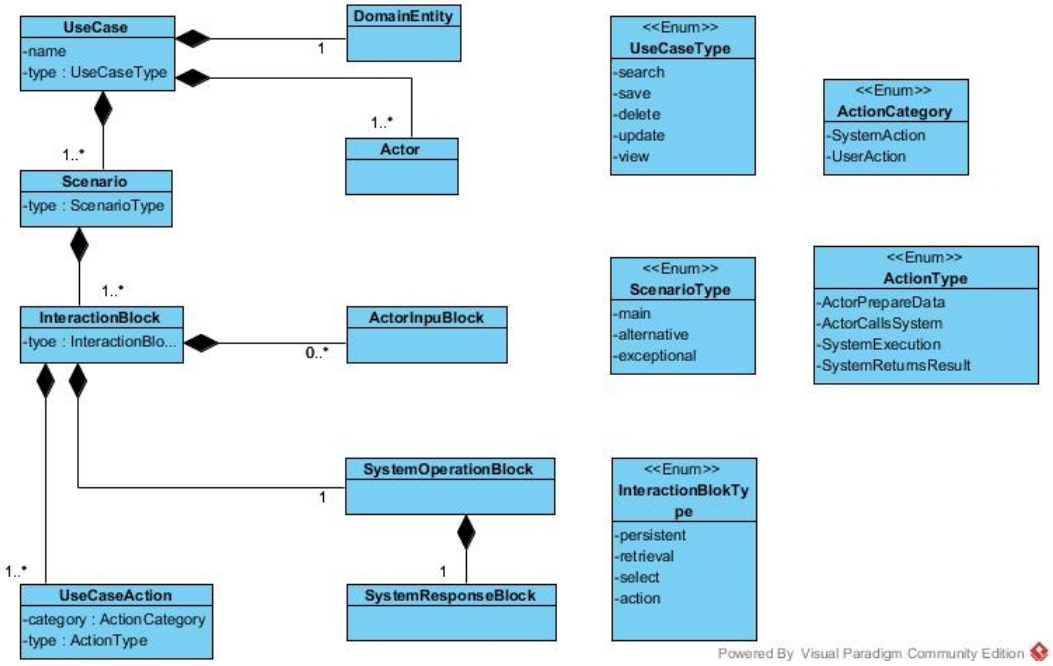

Fig. 3. The conceptual model underlying the proposed patterns.

Use case *(*UseCase*)* is the central concept of this model. One or more actors (Actor) interact with a use cases. In the scope of information systems the purpose of use case is manipulate the data and so, each use case is related to some main domain entity *(*Domain Entity). A use case has a type (UseCaseType) that can be: *create*, *view*, *update*, *delete* and *search*. 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 are organized in interaction block *(*InteractionBlock) in order to better describe user system interactions. InteractionBlock is used to describe interaction between the user and the system. Each interaction block has a type (InteractonBlockType) such as: *persistent*, *retrieval*, *select*, and *customer-action.* Depending on the UseCaseType, use case can contain a different type of InteractionBlock.

4. PATTERNS

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

> The *Billing System* should be 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 should 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.
> […]

The proposed patterns should be preferentially used together. These patterns are described following the pattern template discussed in Section 1.3, 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

This pattern is **also known as** DEFINE ABSTRACT USE CASE TYPES.

Context: You are a requirements engineer, a business analyst or even developer and frequently you have to design or specify with information systems, and you consider use cases as your tool for user requirements specification. You noticed that use cases in different information systems share similar context. This pattern is appropriate when the scope of these use cases is to manipulate data entities, which is common on information systems.

Problem: You are always in doubt about how to organize these use cases. When you have many use cases but you realized that some type of use cases are specified in a very similar way. However, you don't know how to reuse your use cases. Some questions appear: How to make use cases reusable? Is it possible to create templates for scenarios or actions and later then adding specificity to these general actions?

Solution: The first step is to identify these types of use cases. On one hand, we usually work with many use cases, but, on the other hand, all of these use cases can be classified either as create, search, view, update or delete. A good starting point for this step can be to adopted use case content patterns proposed by Martin Langlands [18]. The second step is to categorize each of your use cases according to theses types. The third step is to relate each use case with just one primary domain entity. These business entities are used by use cases, so use cases manipulate these business entities that are related with them.

Example: The Figure 4 describes use case specification with use case types. Each use case is associated with appropriate type and basic domain entity.

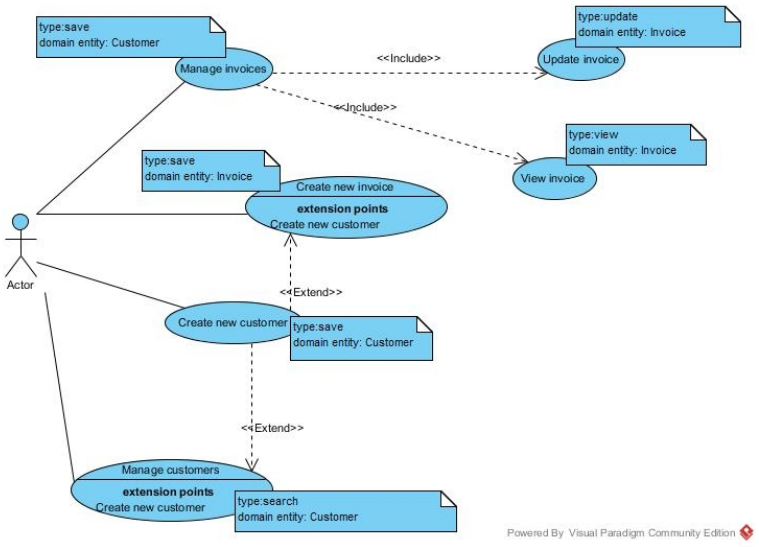

Fig. 4. Use case specification with use case types.

Consequence: As a result of the application of this pattern, each use case use case is bind to appropriate domain model and suitable type. This is the first step in order to ensure consistence between use case and domain model (see KEEP USE CASE AND DOMAIN MODEL CONSISTENT), and defining appropriate interaction block (see DEFINE USE CASE WITH INTERACTION BLOCKS).

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

Known Uses: SilabMDD approach, XIS and XIS-Mobile languages.

4.2 KEEP USE CASE AND DOMAIN MODEL CONSISTENT Pattern

Context: Requirements comes from business system, and use cases can be used for describing them. Domain model provide the foundation for the business system. Domain model, i.e. in business, exist without use cases. On one side, a system under construction may be well-known, so domain model can be defined and specified before defined requirements. On the other side, the system under construction is not well-known, so during requirements specification we need to define domain models as well.

Problem: Each use case encapsulates its structure and behavior. The structure of use cases is not separated from behavior; they are intertwined (overlapped) in use case actions, use case pre-condition and post-condition. Therefore, use case actions describe the use case behavior, and at the same time they hide the structure. The structure encapsulated in the use case is a part of domain model. The problem is how to ensure this consistence (use case structure with domain model) because there are many use cases specified in requirements specification.

Solution: In order to provide consistence between use case structure and domain model, each part of use cases must be related with the concepts of domain model. Each use case should be related to one primary domain entity. Use case is executed over that domain entity (in our example bellow that domain entity is *Invoice* entity). That means that all entities that are referenced in use case actions are related with this entity (in our example that is the case with *Customer*, and *Invoice item*). Everything that we specify in use cases is followed by some rule. For example input rule must define

which data information user need to enter when he want to create a new invoice. This rule should be specified in context of domain model.

Example: The figure bellow shows specification of use case "*Create new invoice"* in context of domain model. Each input rule is bind to domain entity that specifies type of domain object (input rule *"Invoice basic detail"* is bind to Invoice entity object). Each input property should be bind to appropriate domain entity attribute. In case that input properly is reference to another domain entity, we specify appropriate *select interaction block* that describe how user interaction with system before it select domain entity. This is very important because selection of some domain object can be:
(1) Simple (from dropdown list, without specific constraints).
(2) Simple with some criteria for searching (filtering).
(3) Complex with some specific constraint that selected object need satisfied.

Therefore, this information will have an impact on the system operation that system should provide so must be somewhere specified (for example in system operation precondition section).

Consequence: As a result of the application of this pattern, we ensure the consistent between use cases and domain model.

Related Patterns: DEFINE USE CASE TYPES, and COMPLEMENT USE CASE WITH UI-PROTOTYPES.

Known Uses: SilabMDD approach, XIS and XIS-Mobile languages.

```
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 5 *"Create new invoice"* use case specification

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

The pattern is **also known as** DEFINE USE CASE WITH DIFFERENT TYPES OF SCENARIOS AND INTERACTION BLOCKS.

Context: Interaction between user and system need to be clear enough to be readable and understandable for all stakeholders. On one hand, it should be readably by ordinary people that can read and verify it, but on the other hand, it should contains enough details for developer to implement required system operations, tester to generate or make automated tests and user interface designer to develop or generate prototype of user interface.

Problem: Interaction between user and system is usually done through graphical user interface. Therefore, graphical user interface is used by the actor to send request to system to execute a system's operation, as well as, from system to present actor results of execution of system operation. How can we describe this interaction, so that we can separate information related to user interface from information that is important to system's operation, but on the other hand, ensure consistent between them?

Solution: The solution is to describe interactions between user and system using **Interaction block.** Each scenario contains the minimum one **Interaction block. Interaction block** is composed by zero or more **Actor input block,** and one **System operation block** which is associated with particular **System response block**.

Actor input block contains one or more *ActorPrepareData* actions (see Define use case Actions with different types). Each *ActorPrepareData* action should be bind to particular input rule that describe information that user enters during interaction (see Keep use case and domain model Consistent).

System operation block contains specification of one *ActorCallsSystem* action. Each **System operation block** is associated with appropriate **System response block.** The **System response block** specifies the result of system operation execution. Each system operation belongs to one of the following categories (CRUD):

(1) Operation that creates some entity (C).

(2) Operation that deletes some entity (D).

(3) Operation that updates some entity and related references (U).

(4) Operation that searches some entity (R).

After system operation execution, system returns some results to actors. In case that actor sends request 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 appropriate *"Retrieval"* interaction block (see bellow types of interaction blocks)

We identified the following types of interaction blocks:

(1) ***Persistent interaction block***. This block describes 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 interaction between user and system when user wants, after searching domain object, select one or more domain entities and put it 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 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 requested from system on some domain entity object. This block can be used in all types of use cases. This block is can be used in description as alternative action block for ***System operation block***.

Table I describes which type of interaction block can be used in which type of use cases:

Table I. 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 6 below present specification of *"Create new invoice"* use case according to 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 bind to *Invoice* object, while the second one is bind to *Invoice item* object (see *Keep Use Case and Domain Model Consistent*).

```
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. 6. "Create new invoice" use case specification.

Consequence: As a result of the application of this pattern we clear 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 and Domain Model Consistent), as well as, a good basis for specifying user interface details (see Show Use Case's UI-Prototypes).

Related Patterns: DEFINE USE CASE TYPES, KEEP USE CASE AND DOMAIN MODEL CONSISTENT, and COMPLEMENT USE CASE WITH UI-PROTOTYPES.

Known Uses: SILABMDD approach.

4.4 DEFINE USE CASE WITH DIFFERENT ACTION TYPES Pattern

Context: The integration of use cases in a Model Driven Engineering process requires more rigorous specifications. This means that use case specification should be consider as a model with precise syntax and semantic. On the other hand, use cases as a technique for user requirements specification became popular, because they are very useful and readably by ordinary people. If we want to more formal describe use cases this fact we must bear in mind. Despite the fact that use cases describe interaction between user and system, another approach such as task modeling is used independently or in conjunction with use cases in description of user-system interaction.

Problem: The use cases are primary narrative approach, but even that there is no standard that specifies what textual specification of use case should look like. Different types of actions can be occurred during use case scenarios execution. The problem is that semantics of these actions is not clear defined, as well as a level of use case formality. 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, it depends on the use case specification analysts experience and skills.

Solution: Use case specification involves specification of different types of use case actions. Define appropriate type for each use case action in order to define better syntax and semantics for each action. These actions are classified in two categories: the actions performed by the actor (*ActorAction*), and the actions performed by system (*SystemAction*) [37]. These categories may be subdivided into subcategories of actions. Furthermore, in the category in which actions are performed by the user, we have subcategories actions types such as: (1.1) actor prepares data for system operation (*ActorPrepareData* - APD) and (1.2) actor calls system to execute system operation (*ActorCallsSystem* - ACS). On the other hand, in the category in which actions are performed by the system, we have categories such as: (2.1) system executes system operation (*SystemExecutes* - SE) and (2.2) System returns the result of the system operation execution to user (*SystemReturnsResult* - SRR).

Each *ActorPrepareData* action should be bind to particular input rule that describe data that user enters. *ActorCallsSystem* action should be specified in *System operation block*. Each *SystemExecutes* action should be categorizes as C, R, U, D and for each of this action specify appropriate *SystemReturnsResult* (appropriate use case or interaction block, see Define Use Case with Interaction Blocks)

Example:

Consequence: Use case actions specified with appropriate level of details with clear semantic.

Related Patterns: PROPERTY LIST, FILED LIST; DEFINE USE CASE TYPES, KEEP USE CASE AND DOMAIN MODEL CONSISTENT, and COMPLEMENT USE CASE WITH UI-PROTOTYPES.

Known Uses: SilabMDD approach.

4.5 COMPLEMENT USE CASE WITH UI-PROTOTYPES Pattern

The pattern is **also known as** ATTACH UI PROTOTYPES TO USE CASE.

Context: Interactions between user and system can be described on different way. For example, these interactions can be described using task modeling or using use cases. The main difference between these approaches is that task modeling is more related to the user interface and more concrete. In order to validate user requirements (specified as use case model) through UI prototype these approaches should be integrated, or we need to extend use case specification with user interface details.

Problem: Good way to validate user requirements is to automatically produce prototype. Use cases are well-known technique for user requirements specification, but we need keep in mind that specifying user interface detail directly in use case should be avoided. The problem is how we can at same time avoid specifying user interface detail in use case actions and obtain user interface prototype directly from use case specification.

Solution: The solution may be, use interaction block as a basic for specification of user interface prototype. Each use case should contain its UI component container, as well as, each Interaction block. UI component container for interaction block should depend on type of interaction block.

UI Interaction component container should hold:

(1) UI component container for specification each *Actor input blocks*. *Actor input block* should be associated with appropriate templates that define how UI widget component are arranged to present input for particular domain object. This container should depend on: (1) type of application (desktop, web or mobile application), (2) does it presents one domain object (for example Invoice) or more domain objects (for example Invoice item), and (3) does it present only one type of object or more (different type of domain) objects (for example view customer details can include view different objects such as Invoice, Deliveries, Customer accounts etc.).

(2) I component container for specification *System operation block*. In this container should be defined component that send request to system to execute appropriate system operation, and specification of another use case that will present to user or interaction block that will be appeared after successful execution of system operation.

Example: Figure 7 presents relation between elements of use case specification and graphical user interface specification. Figure 8 shows the concrete user interface prototype for use case specification *"Create new invoice"*.

Consequence: Use case can be used to validate user requirements using UI prototype.

Related Patterns: UI Prototypes; DEFINE USE CASE TYPES, and KEEP USE CASE AND DOMAIN MODEL CONSISTENT.

Known Uses: SilabMDD approach, XIS and XIS-Mobile languages.

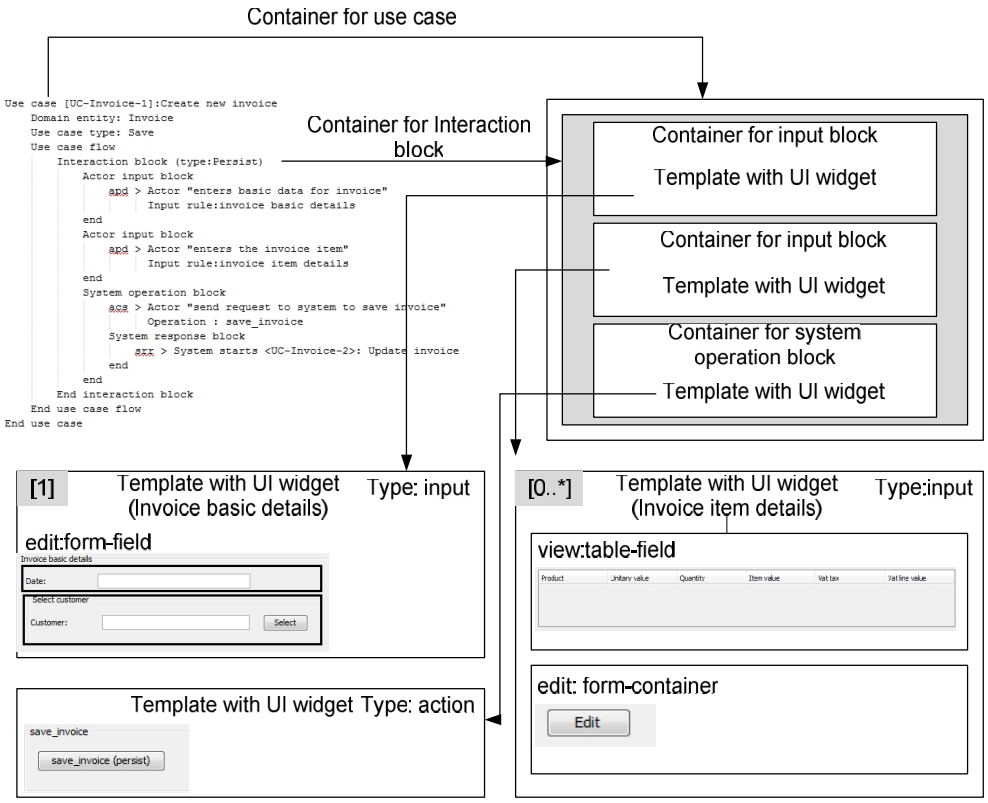

Fig. 7. Relation between elements of use case specification and graphical user interface specification.

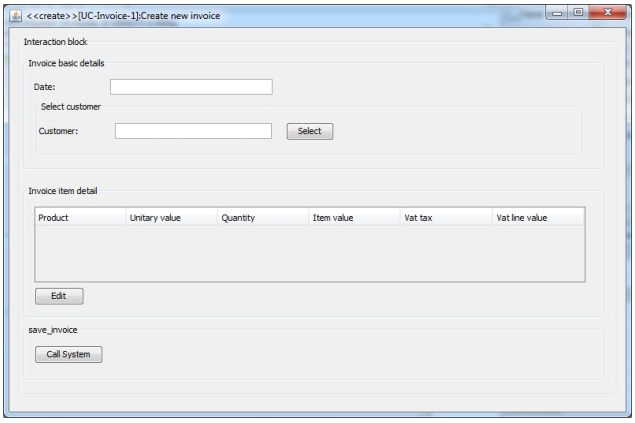

Fig. 8. Simple UI prototype for the use case "Create new invoice".

5. CONCLUSION

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

In this paper we propose a cohesive set of patterns that make 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 use cases at a meta or high-level. Interactions between users and the system should be defined as user-system dialogs and structured as interaction blocks. Considering that these interactions are usually done through graphical user interface, specification of user interfaces could be also defined in the scope of each use cases specification as a complementary model.

The use case specifications using the proposed patterns are being supported by some of the tool prototypes that are being developed and publicly available for 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.