Improving the Quality of Requirements with Refactoring

Ricardo Ramos[1], Eduardo K. Piveta[2], Jaelson Castro[1], João Araújo[3], Ana Moreira[3], Pedro Guerreiro[3], Marcelo S. Pimenta[2], and R. Tom Price[2]

[1] Universidade Federal de Pernambuco (UFPE) - Brasil
{rar2, jbc}@cin.ufpe.br

[2] Instituto de Informática - Universidade Federal do Rio Grande do Sul (UFGRS)
{epiveta, mpimenta}@inf.ufrgs.br, tomprice@terra.com.br

[3] Universidade Nova de Lisboa (UNL) - Portugal
{ja, amm, pg}@di.fct.unl.pt

Abstract. *Requirements specification can often exhibit some shortcomings, regarding contents and organization of its partial specification elements. Sometimes, modularization is deficient, with modules dealing with too much information, or the same functionality being specified in different modules. Left unchecked, these inadequacies will propagate themselves to the subsequent phases of the software development and cause problems during maintenance. We have been able to identify a collection of typical deficiencies in the specification of structured documents and we propose a collection of refactorings that minimize or remove them. Doing this early in the development process increases requirements modularity and understandability. A case study is conducted to illustrate the use of these refactoring practices on two existing requirement documents.*

1. Introduction

Several approaches are used to express the requirements of a software system and are extensively used in academia and industry. These approaches describe both the interactions between the users and the system and the functionalities to be provided. Requirements are structured and described using, for example, use cases, viewpoints, structured textual descriptions, activity diagrams, sequence diagrams [Li 1999].

Over the past few years, a set of typical issues seems to plague the requirements specifications. For example, requirements that have been abandoned and are no longer meaningful, descriptions that are unnecessarily long and convoluted, information that is duplicated [Wiegers 2003] [Firesmith 2007]. Inspired by the code refactoring literature [Opdyke 1992], we set off to identify a set of bad smells that could indicate potential refactoring opportunities.

The occurrence of these bad smells decreases the overall understandability quality of software, not only during implementation, but throughout the development process [Boehm and Sullivan 2000] and can be minimized by the identification of their symptoms and the removal of their causes. These symptoms may indicate potential problems with the software [Elssamadisy and Schalliol 2002] and can be removed using appropriate refactoring transformations. Their removal in early stages of software development process reduces the costs associated with software changes. These cost reductions could be three to six times more in later stages than during requirements activities [Pressman 2005].

Opdyke (1992) initially coined the term refactoring as the process of improving the design of existing software using systematic transformations, without changing its observable behavior. The term is also used to refer to program restructuring operations aiming to support the design and evolution of object-oriented software. Refactorings express ideas of good style, which can have a significant impact on the maintainability and evolvability of code bases.

There are refactorings and tools for refactoring requirements documents [Xu et al. 2004] [Yu et al. 2004] [Rui et al. 2003] but they focus on specific techniques, such as use case models and do not directly address textual descriptions or other mechanisms used to specify requirements. Moreover, these approaches do not provide any guidelines on how to identify the potential problems. As such, they say nothing about which refactoring can be used to address those issues.

In this paper we describe a generic approach to identify *refactoring opportunities* in requirements descriptions, together with a collection of associated refactorings. The resulting approach can be instantiated to any requirements description technique (e.g., viewpoints [Finkelstein and Sommerville 1996], goals [Lamsweerde 2001]). In this paper we will discuss our ideas using use cases. In summary, our main contributions are:

– To provide a collection of *refactoring opportunities* that can be found in requirements, corresponding to bad smells in the specification: *large requirement*, *complex conditional structure*, *lazy requirement*, *naming problems* and *duplicated activities*. For each opportunity we describe (a) a method to identify occurrences of the problem and (b) the refactorings that can be used to minimize the effects of the problem occurrences.

– A collection of refactorings to manipulate requirements: *extract requirement*, *rename requirement*, *move activity*, *inline requirement*, and *extract alternative flows*. Each refactoring contains the context that suggests the application of the refactoring, the type of solution it provides, a motivation for the transformations, its mechanics (a set of well defined steps) and an example of a refactored description.

To assess the *refactoring opportunities* and the refactorings described in this paper, we conducted a case study with two selected requirements documents: the Health Watcher system[1] [Soares et al. 2002], a web-based health complaint system, and the Order Processing System described by Schneider and Winters (1998).

This paper is structured as follows: Section 2 introduces a collection of refactoring opportunities in use case descriptions. Section 3 presents refactorings to manipulate requirements descriptions. Section 4 describes a case study applied to try out and assess the proposed techniques. Section 5 discusses some related work. Finally, Section 6 draws some conclusions and points out directions for future work.

2. Refactoring Opportunities

Fowler (2000) introduces the expression "bad smells" as indications of deficiencies that can appear in software artifacts. In this paper, we use the expression *refactoring opportunity* to refer to the types of bad smells that can appear in requirements artifacts.

However, *refactoring opportunities* should not be seen as exact rules allowing automatic application of refactorings. The requirements engineer needs to decide about the trade-offs in changing the system requirements and needs to choose which refactoring is more adequate for each opportunity.

In this section, we describe *refactoring opportunities* in requirements as well as associated refactorings that might be used for each defined opportunity. We consider the following *refactoring opportunities*: *large requirement*, *complex conditional structure*, *lazy requirement*, *naming problems* and *duplicated activities*. A brief textual description and some refactorings that could help in the solution of the identified problem are also provided but the refactorings themselves are described in Section 3.

2.1. Large Requirement

Large requirements occur when (i) a requirement is trying to handle several concerns[2] at the same time or (ii) there are many alternative flows and steps [Firesmith 2007] [Alexander and Stevens 2002].

Use the *Extract Requirement* refactoring (Section 3.1) to extract information related to a given concern and insert it into a new requirement. This operation could be

[1] Is available at http://aosd.di.fct.unl.pt/ea-icse2007/case_study.html

[2] A concern is a matter of interest in the software [Dijkstra 1982] e.g., security, performance, information retrieval.

repeated for each major concern addressed by this large requirement. If the flows or other components of a requirement could be moved to another requirement, it could be used the *Move Activity* refactoring (Section 3.3).

After extracting or relocating requirements, we sometimes need to rename them to better express the intention of the newly created one or of the one that was modified. In this case, the *Rename Requirement* refactoring (Section 3.2) could be used to provide more appropriated names.

This refactoring opportunity is particularly important when there is a limit for the size of each requirement, set by the organization's Software Quality Assurance Team.

2.2. Complex Conditional Structure

Complex conditional structures occur when a requirement has large or complex expressions or needs several other requirements to be complete. A requirement can contain complex conditional structures due to its nature or simply because it is poorly described. The second situation shows up when there are nested conditional structures or long sequences thereof. [Jacobson et al. 1997].

This refactoring opportunity can be handled with the *Extract Requirement* refactoring (Section 3.1) to create a new requirement from a conditional structure or with the Move Activity refactoring (3.3) to move a conditional structure to a new requirement in the same context.

2.3. Lazy Requirement

Lazy requirements correspond to one of the following cases: (i) when a requirement describes a small number of responsibilities or the impact of the requirements is unclear; (ii) when the size of a requirement is too small, in terms of its components; (iii) when a requirement does not capture all the activities related to a given concern, or is incomplete [Firesmith 2007] [Sommerville 1997].

These problems happen when only the steps of a flow are described and the output is ill-defined or unclear, and also, when after some modifications are performed, the requirement is no longer appropriate. Other situations are due to the incomplete nature of the requirements descriptions, as well as requirements descriptions which were delayed (perhaps due to low priority) and not described in sufficient detail [Alexander and Stevens 2002].

Such lazy requirements can be improved using the *Move Activity* refactoring (Section 3.3) to rearrange the steps of the requirement into other requirements. Also, if a requirement is isolated and is not being used by any other requirement or is not associated with any user or system, we may simply delete it.

If some requirement is not being directly used by any stakeholder or subsystem and is only referenced by one or a few other requirements, a possibility is to use the *Inline Requirement* refactoring (Section 3.4) to move the features of this requirement to the ones that need it.

A lazy requirement can simply be an incomplete requirement. In this case, there is no need for refactoring. The requirements engineer only needs to include the missing information.

2.4. Naming Problems

Meaningless or inconsistent names are situations where the requirement's name bears no relation with the concept described or the same name is used for different concepts [Wiegers 2003].

Several approaches for describing structured requirements documents use some kind of naming mechanisms to express the intent of the requirement [Sommerville 1997]. Some authors describe problems with meaningless names in requirements [Alexander and Stevens 2002] and usually provide some guidelines, such as *use verb + noun phrase (active verb in imperative mode)* when possible. Inconsistent names occur when two requirements are represented by the same identifier but have different meanings. Other problems include excessive use of synonyms, absence of clear definitions, partially overlapping concepts described by different names with no clear definitions and usage of names with different meanings.

These naming mismatches make requirements harder to understand. The existence and enforcement of a correct nomenclature in a system under developed is a valuable asset. Other types of meaningless names are the use of abbreviations to express a given name. When these kinds of situations occur in the name of a requirement, the developer can use the *Rename Requirement* refactoring (Section 3.2), to give an adequate name to the requirement, thus revealing its purpose [Clements and Northrop, 2001].

2.5. Duplicated Activities

Duplicated activities are a situation that occurs when (i) the same requirement is duplicated in different places in a requirements document or (ii) the same activities or the same pre-post conditions appear in several requirements [Firesmith 2007] [Jacobson 1997].

An activity is present in more than one place offers an opportunity for refactoring. In the simplest case, the steps in a main flow or in an alternative flow are repeated in set of requirements. A possible solution is to use the *Extract Requirement* refactoring (Section 3.1), to remove the duplication and to create a new requirement expressing this specific set of activities. If we have to maintain a relationship between the original requirement and the one that was extracted, we can provide pointers to the latter, thus preserving the original information.

Another common duplication problem occurs when an individual activity appears in several requirements. This duplication could be removed using the *Extract Requirement* refactoring as mentioned above. If the activities are similar, but not exactly the same, we may need to separate the duplicated piece from the rest of the requirement.

A different approach is to use aspects [Kiczales et al. 1997] to express these duplicated activities. The use of aspect-oriented requirements engineering [Rashid et al. 2003] provides a good mechanism to modularize these activities that are scattered among several places in a requirements document and compose them without losing the original information. The use of aspectual requirements [Rashid et al. 2003] [Jacobson 2003] [Jacobson 2005] adds a dependency inversion that provides low coupling between the requirements and the activities in the aspectual requirement.

3. Refactoring Requirements

In this section we define and describe a collection of requirements refactorings. Some of these refactorings were mentioned in Sections 2, providing solutions to the *refactoring opportunities* that were presented. We now describe the following five refactorings: *Extract Requirement*, *Rename Requirement*, *Move Activity*, *Inline Requirement*, *Extract Alternative Flows*.

We use the format recommended by Fowler (2000) for describing refactorings. For each refactoring we provide a context, a solution, the motivation for the application of the refactoring, a set of mechanics to apply the refactoring and an example illustrating the application of the refactoring. Note that although the examples in each individual refactoring are based on use cases, the refactorings could also be used with other techniques for describing requirements. The examples for each refactoring were taken from two requirements documents mentioned in the introduction.

3.1. Extract Requirement

Context. A set of inter-related information is used in several places or could be better modularized in a separate requirement. Or a requirement is too large or contains information related to a feature that is scattered across several requirements or is tangled with other concerns.

Solution. Extract the information to a new requirement and name it according to the context.

Motivation. This refactoring should be applied when there are large requirements that can be split into two or more new requirements. These large requirements include a great deal of information that is difficult to understand. Furthermore it is not easy to locate the needed information quickly [Alexander 2002] [Sommerville 1997].

Mechanics. The following activities should be performed:

1.Create a new requirement and name it.
2.Select the information you want to extract.
3.Add the selected information to the new requirement.
4.Remove the information from the original requirement.
5.Make sure the original requirement is acceptable without the removed information.
6.Update the references in dependent requirements.

Example. In the Health Watcher system [Soares et al. 2002], consider the use case named *Complaint Specification* (Figure 1) that deals with three different types of complaints (animal, food or diverse).

Main flow of events: This use case makes possible for a citizen to register complaints. Complaints can be Animal Complaint, Food Complaint or Diverse Complaint:

1. The citizen informs the kind of complaint;
2. The system registers the kind, date and time of the attendance;
3. The system shows the specific screen for each type of complaint;
4. The citizen provides the data;
5. The system saves the complaint (with the OPENED state), return a code for the attendance, so that the citizen can take note and query for the situation of his/her complaint.

Figure 1. Use case complaint specification

As each type of complaint can have different pre-conditions, different data to be manipulated and different interfaces, the requirements engineer might extract each type as a separated use case. Three new use cases were created (*Register Animal Complaint* (Figure 3), *Register Food Complaint* and *Register Diverse Complaint*). The original use case is modified to reference the extracted use cases (Figure 2).

Main flow of events:
1. The citizen chooses the kind of complaint;
2. Case the citizen chooses the animal complaint.
   2.1. The main flow will follow the one described on [Register Animal Complaint].
3. If the citizen chooses the food complaint.
   3.1. The main flow will follow the one described on [Register Food Complaint].
4. If the citizen chooses the diverse complaint.
   4.1. The main flow will follow the one described on [Register Diverse Complaint].
5. The system saves the complaint.

Figure 2. Use case complaint specification (after the refactoring).

Main flow of events:
1. The citizen selects the option register animal complaint;
2. The system registers the kind, date and time of the attendance;
3. The system shows the screen for the animal complaint;
4. The citizen provides the complaint description;
5. The system saves the animal complaint (with the OPENED state), return the code for the attendance, so that the citizen can take note and query for the situation of his/her complaint.

Figure 3. Use case register animal complaint

3.2 Rename Requirement

Context. The name of a requirement is not appropriate for the context, is abbreviated or is used in several places with different semantics.

Solution. Rename the requirement to clearly express its purpose.

Motivation. Good names make communication and understanding system abstractions easier and provide a common vocabulary to the development team [Alexander and Stevens 2002].

Mechanics. The following activities should be performed:

1. Select the requirement you want to rename.
2. Change the name of the requirement.
3. Update the references in dependent requirements.

Example. The use case Complaint Specification (Figure 1) [Soares et al.2002] could be renamed to Register Complaint, following the pattern common to all use cases that describe data insertion in the Health Watcher requirements document.

3.3 Move Activity

Context. A set of activities is better accommodated in another requirement description.

Solution. Move the activity to the desired requirement. If the requirement does not exist yet, create a new requirement with the selected activities using the Extract Requirement refactoring.

Motivation. This refactoring is done to improve modularity and to ameliorate the balance of activities among the requirements. Activities are moved from one requirement to another also when a new requirement is created, either manually or by an Extract Requirement refactoring. This improvement in modularity could lead to a better understanding of the system in the long term [Sommerville 2004].

Mechanics. The following activities should be performed:

1. Select the activities you want to move.
2. Move them to the desired requirement.
3. Update references to these activities if needed.

Example. In the Order Processing System [Schneider and Winters 1998], consider a Login use case (Figure 4) that is concerned with user authentication and also with the functionalities selected by the user. The shaded lines show the activities that could be moved to another use case, responsible for the functionality selection flow.

Main flow of events: 1. The use case starts when the user starts the application. 2. The system will display the Login Screen. 3. The user enters a username and password. . . . 6. The system will display the Main Screen 7. The system will select a function …	… 8. While the user does not select Exit loop 9. If the user selects Place Order then Use Place Order. 10. else if the user selects Cancel Order then Use Cancel Order. 11. . . . end if. 16. The use will select a function. end loop. 17. The use case ends.

Figure 4. Use case login

Other refactorings dealing with moving elements from requirements specifications can be defined. For example, the *Move Actor* refactoring is already defined in [Rui et al.2003], to move actors in use case definitions.

3.4 Inline Requirement

Context. A requirement is referenced and used in only one place or in a few places in such a way that its existence is not justified in terms of maintenance costs.

Solution. Insert the requirements description into the requirements that use it.

Motivation. This refactoring reduces the complexity of the requirements model by merging requirements. Each software artifact demands time and resources to understand and maintain it [Jacobson 2003]. If a requirement is not useful enough to justify its existence, the developer can inline it, merging its responsibilities with other requirements.

Mechanics. The following activities should be performed:

1. Copy the activities (including pre and post conditions if applicable) described in the requirement to all requirements that uses this one.
2. Update the affected requirements to reflect the inlined activities and other requirements information.
3. Remove references to the inlined requirement.
4. Remove the inlined requirement.

Example. In the Health Watcher system [Soares et al.2002], there is a use case named *Change Logged Employee* that is clearly an alternative flow of the *Login* requirement. In this use case, the only activitiy is to redirect the flow back to the *Login* flow. This use case could be inlined into the *Login* use case as an alternative flow, using the Inline Requirement refactoring. Figure 5 shows the use case that will be merged with the *Login* use case. Note that this use case can be considered as a lazy requirement and Inline Requirement is the appropriate refactoring for this situation.

Inputs and pre-conditions:
Logged Employee
Outputs and post-conditions:
First employee signed out and new logged employee
Main flow of events:
1. The employee chooses the option:
[Change Logged Employee].
2. The system shows the login screen, and then this main flow will follow the one described in:
[Login].

Figure 5. Use case change logged employee

3.5 Extract Alternative Flows

Context. A set of activities is not commonly performed in the main information flow and the set could be better modularized in an alternative flow.

Solution. Extract the desired activities to an alternative flow.

Motivation. This refactoring is usually used when the main information flow is bloated by trying to deal with several scenarios at the same time. This kind of situation can make difficult the understanding of the requirement responsibilities and the information flow. The use of alternative scenarios is a way to organize complex flows with several conditional structures or that are too complex [Jacobson et al. 1997].

Mechanics. The following activities should be performed:

1.Create a new alternative flow.
2.Select the activities that will be extracted to an alternative flow.
3.Copy these activities to the alternative flow.
4.Update the reference in the main flow to point to the alternative flow.
5.Delete the activities in the main flow.

Example. Consider a use case named *Fill and Ship Order* (Figure 6) [Soares et al. 2002], that deals with more than one scenario in the main flow. An alternative scenario could be extracted, containing the shaded activities and be named to represent this alternative situation (Figure 7).

Main flow of events:
1. The use case starts when the clerk starts the fill-and-ship-order application.
2. . . .
3. While the clerk selects an order
    a. The system display the order
    b. . . .
    c. If there are items not back-ordered and not shipped then
1) For each such item loop
    a. Use Update Product Quantities
    b. . . .
5) . . .
4. The use case ends.

Figure 6. Use case fill and ship order

Alternative Flow - Items not back-ordered and not shipped:
1. For each such item loop
a. Use Update Product Quantities
. . .
5. . . .

Figure 7. Use case fill and ship order alternative flow (after the refactoring)

4. Case Study

The main goal of this case study is to assess the understandability of two requirements documents before and after the application of the refactorings. We use a set of use case metrics [Duran et al.2002] [Ramos et al.2006] to obtain the global picture of the requirements documents being assessed.

The scope. This case study has two phases: the first one is to find *refactoring opportunities*, analyze the situation and apply the appropriate refactoring; the second one is to analyze the results using the metrics selected for the case study.

Each document is from a different domain and was retrieved from different sources. The first requirements document is from the Health Watcher [Soares et al.2002], a web-based health complaint system, aiming to improve the quality of the services provided by health care institutions. This real-system allows the users to register several kinds of health complaints (e.g., complaints against restaurants and food shops), so that an official can investigate the complaints and take the necessary actions. The requirements document contains nine use cases, thirty five steps and two actors. This requirements document is also being used as a benchmark at the Early Aspects Workshop at ICSE (Workshop in Aspect-Oriented Requirements Engineering and Architecture Design)[3].

The second requirements document is the Order Processing System described by Schneider and Winters (1998). This system provides a collection of use cases to a mail based ordering processing system and is composed by thirteen use cases, eighty six steps and seven actors.

The goal of the second phase is to assess the refactoring results. The metrics used in this case study are the following: Number of use cases of the requirements document (NOUC), Number of steps of the use case (NOS), Number of conditional steps of the use case (NOCS), Number of times a use case is included or extends other use cases (NIE) and Number of scenarios of the use case (NOSC). All metrics except NOSC [Ramos et al.2006] are based on Duran et al. [Duran et al.2002].

The size of a structure is proportional the effort needed to understand it [Fenton and Pfleeger 1997]. So, in this case study, we assume that smaller use cases, with less steps and/or conditional structures can be easier to understand. Also, a module usually

[3] http://aosd.di.fct.unl.pt/ea-icse2007.

needs to know about related modules to be fully understood [Sommerville 1997]. So, the number of related modules affects the complexity and understandability of a given module.

At the requirements level, the understandability of a requirement is directly associated with its modularity and the number of concerns in a requirement [Alexander and Stevens 2002]. In this case study, we state that a high number of scenarios in a use case could increase the complexity of this use case.

Instrumentation. The study was performed using two requirements documents. The Health Watcher document uses the template of Cockburn's Use Case [Cockburn 1997] and the Order Process System document uses Schneider and Winters template [Schneider and Winters 1998]. The *refactoring opportunities* we seek are the ones described in this paper. The evaluation results will be compared with metric values. The results are displayed graphically.

The variables. The independent variables are all those that are manipulated and controlled during the study [Wohlin et al.2000]. In this case study the independent variables are the two requirements documents. The dependent variables are all those that are under analysis [Wohlin et al.2000]. In this case study the dependent variables are the number of: use cases, steps, conditional steps and scenarios.

Project. The two requirements documents will be measured before and after the refactoring process. Each document is inspected to identify potential *refactoring opportunities*. For each detected opportunity there is the need to choose the appropriated refactoring to apply in the requirements document. The refactorings used are the ones described in Section 3.

4.1 The Case Study Realization

We discuss first the *refactoring opportunities* that were searched in the Health Watcher requirements document and then the ones in the Order Processing system.

Health Watcher. We found *refactoring opportunities* in three use cases of the respective requirements document: Complaint Specification, Register Tables and Choose Logged Employee.

In the *Complaint Specification* use case (Figure 1) the naming problem and large requirement bad smells were present. We improved this use case by using the Rename Requirement and Extract Requirement refactorings. The use case was renamed to *Register Complaint* (following the same name pattern to all use cases that describe data insertion). Three new use cases were created (*Register Animal Complaint* (Figure 3), *Register Food Complaint* and *Register Diverse Complaint*). The *Choose Logged Employee* use case was inlined into the *Login* use case as an alternative flow.

The *refactoring opportunities* found to the *Register Tables* use case were Large Requirement and Duplicated Activities. We used the Extract Requirement refactoring to create seven new use cases: (*Register Health Unit*, *Register Specialty*, *Update Specialty*, *Register Disease*, *Update Disease*, *Register Symptom* and *Update Symptom*). The use case *Register Tables* and the duplicated activities were removed.

Order Processing System. The use cases wherein *refactoring opportunities* were found are: *Login* (Figure 4), *Get Status* on *Order and Fill and Ship Order* (Fig. 6).

The *Login* use case has the following *refactoring opportunities*: Large Requirement and Complex Conditional Structure. A new use case was created and named *Select Function* and the activities that were not in the login context were moved to this newly created use case, using the Move Activity refactoring. The pre-condition logged user was added to the use cases that were part of login relationships.

The *Get Status on Order* could be seen as a lazy requirement as its responsibilities are unclear. This use case could be seen as an alternative scenario to the *Search for Order* use case. The *Fill and Ship Order* use case was considered a large requirement because it deals with two different concerns in the main flow. An alternative scenario named Items not back-ordered and not shipped (Figure 7) was extracted to encapsulate the activities regarding one of the identified concerns.

The selected metrics were computed before and after the refactoring process and the result values are analyzed in next section.

4.2 Data Analysis

For the *Health Watcher*, the number of use cases and steps increased, while the mean and standard deviation of the other metric values decreased (number of steps, number of scenarios, number of conditional steps and number of inclusions and extensions), as shown in Figure8. The number of use cases went from 9 to 19 and from 37 steps to 70.

In general, the number of use cases is higher, but the use cases are smaller in terms of steps, number of scenarios, conditional structures and inclusions and extensions. They are also more homogeneous regarding these constructions, as the standard deviation decreases for all the selected metrics.

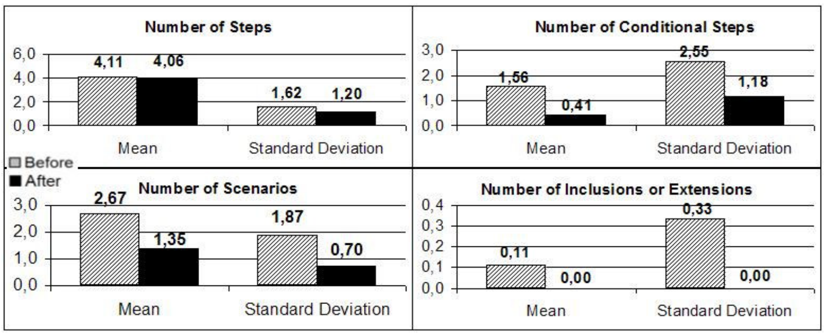

Figure 8. Metric mean and standard deviation values for the Health Watcher.

For the *Order Processing System*, the number of use cases remains the same (13 use cases) and the number of steps decreased from 91 to 88. The mean and standard deviation of the other metric values (number of steps, number of scenarios, number of conditional steps, and number of inclusions and extensions) decreased, as shown in Figure 9.

In general, the number of use cases is the same, but the use cases are smaller in terms of steps, number of scenarios, conditional structures and inclusions and extensions. They are also more homogeneous regarding these constructions, as the standard deviation decreases for all the selected metrics.

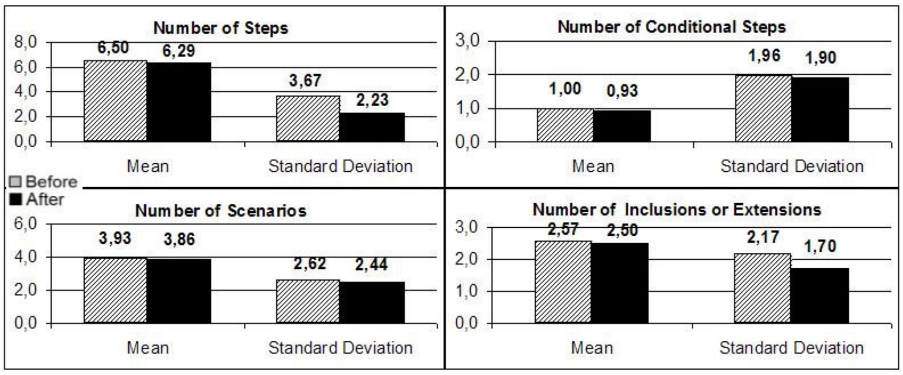

Figure 9. Metric mean and standard deviation values for the Order Processing System

Using the insights gained during the data analysis and considering that: i) the size of a structure can indicate the effort needed to understand it [Fenton and Pfleeger 1997], ii) the number of related modules affects the complexity and understandability of a given module [Sommerville 1997] and iii) the understandability of a requirement is directly associated with its modularity and the number of concerns in a requirement [Alexander and Stevens 2002], we infer that the understandability Quality was improved in both requirements documents after the applications of the refactorings.

5 Related Work

Rui et al. (2003) describe a meta-model for use case modeling and categorize a list of use case refactorings. We extend their refactoring listing with a detailed collection of *refactoring opportunities* and requirements refactorings definitions, including, for each refactoring: the context for the application for a given refactoring, a possible solution, the motivation to apply the transformations and an example of its practical use.

Yu et al. (2004) explain how refactoring can be applied in order to improve the organization of use case models. They focus on the decomposition of a use case and the reorganization of relationships between use cases. They also describe ten refactorings that could be used to improve the overall organization of use case models, such as inclusion or extension mechanisms introduction, use case deletion or refactorings that manipulate the inheritance tree. While Yu et al. focus on refactoring the use case models, we focus on refactoring requirements descriptions. As a consequence, our refactorings are finer grained than theirs. We also discuss in details the mechanics of each refactoring and possible *refactoring opportunities* in the context of requirements descriptions.

Xu et al. (2004) present a tool that helps in refactoring of use case models and use it to conduct a case study using an ATM application. While they focus on the automation side of the use case refactoring process, we concentrate on providing definitions for deficiencies in use cases scenarios and refactorings in this context. Further research is needed to attain automatic identification of problems in use case descriptions and to support the refactorings proposed in this paper.

Overgaard and Palmkvist (2004) offer a pattern named large use case as a means for describing activities for which a satisfactory separation in several use cases could not be found. We extend their work discussing an opportunity to apply their pattern as a solution to the lazy requirement opportunity and discussing the cases where this large requirement becomes a problem and can be specified as a set of smaller requirements.

6 Conclusions

This paper describes a collection of *refactoring opportunities* that might occur in requirements and provides a collection of refactorings that can be used to improve the quality of requirements where these opportunities appear. For each refactoring, we describe a collection of examples for practical use for both refactorings and the identification of *refactoring opportunities*.

The *refactoring opportunities* that we have identified are not mandatory rules. Their goal is make the requirements more understandable and to improve overall organization of the project, but depending on the significance of quality attributes in each case, the requirements engineer might take actions that slightly differ from the guidelines. Also, the expected granularity of the requirements descriptions might influence the choice of the *refactoring opportunities*. The requirements engineer should take these issues into consideration. To future work we are planning to formalize the *refactoring opportunities* and to develop a tool that supports it.

The term refactoring is originally used on code level for structural changes that preserve behavior or meaning but increase understandability. Refactorings of code are done when there are test suites that can verify that the behavior of the system is preserved after the refactoring. As requirements documents are not executable artifacts and we do not have test cases available, we cannot fully evaluate if the behavior is preserved after the refactoring process. Further research is needed to correctly address this issue.

References

Alexander, I.F., Stevens, R.: (2002) "Writing Better Requirements", Pearson Education Limited.

Boehm, B., Sullivan, K.: (2000) "Software economics: a roadmap", in: ICSE – Future of SE Track. 319–343.

Clements, P., Northrop, L.: (2001) "Software Product Lines", Addison Wesley Professional.

Cockburn, A.: (2000) "Writing Effective Use Cases", Addison Wesley.

Dijkstra, E.W.: (1992) "On the role of scientific thought (EWD 447)", in: Selected Writings on Computing: A Personal Perspective. Springer-Verlag.

Duran, A., Cortes, A.R., Corchuelo, R., Toro, M.: (2002) "Supporting requirements verification using xslt", in: Requirements Engineering Conference, Essen, Germany.

Elssamadisy, A., Schalliol, G.: (2002) "Recognizing and responding to bad smells in extreme programming", in: Proceedings of the 24th International conference on Software Engineering, May 19-25, 2002, Orlando, Florida, USA.

Fenton, N.E., Pfleeger, S.L.: (1997) "Software Metrics: A Rigorous and Practical Approach", PWS Publishing Company.

Finkelstein, A., and Sommerville, I.: (1996) "The Viewpoints FAQ", in: BCS/IEE Software Engineering Journal, Vol. 11(1).

Firesmith, D.: (2007) "Common Requirements Problems, Their Negative Consequences, and Industry Best Practices to Help Solve Them", in Journal of Object Technology, vol. 6, no. 1, January-February 2007, pp. 17-33.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: (2000) "Refactoring: improving the design of existing code", in: Object Technology Series. Addison-Wesley.

Jacobson, I., Griss, M., Jonsson, P.: (1997) “Software Reuse: Architecture, Process, and Organization for Business Success”, in: Addison Wesley.

Jacobson, I.: (2003) “Use cases and aspects - Working seamlessly together”, In: Journal of Object Technology 2(4).

Jacobson, I., Ng, P.W.: (2005) “Aspect-Oriented Software Development with Use Cases”, Addison-Wesley.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: (1997) “Aspect-oriented programming”, in: Aksit, M., Matsuoka, S., eds.: 11th Europeen Conf. Object-Oriented Programming. Volume 1241 of LNCS., Springer Verlag 220–242.

Lamsweerde, A. Van.: (2001) “Goal-Oriented Requirements Engineering: A Guided Tour”, in: 5th International Symposium. on Requirements Engineering. August, Toronto – Canada.

Li, L.: (1999) “A semi-automatic approach to translating use cases to sequence diagrams”, in: TOOLS ’99: Proceedings of the Technology of Object-Oriented Languages and Systems, Washington, DC, USA, IEEE Computer Society 184.

Opdyke, W.F.: (1992) “Refactoring object-oriented frameworks”, PhD thesis, University of Illinois at Urbana-Champaign.

Overgaard, G., Palmkvist, K.: (2004) “Use Cases Patterns and Blueprints”, Addison Wesley Professional.

Pressman, R.: (2005) “Software Engineering: A Practitioner’s Approach”, McGraw-Hill.

Ramos, R.A., Araújo, J., Castro, J., Moreira, A., Alencar, F., Silva, C.: (2006) “Uma abordagem de instanciação de métricas para medir documentos de requisitos orientados a aspectos”, in: 3° Brazilian Workshop on Aspect Oriented Software Development - WASP2006. Florianopolis, Brazil.

Ramos, R. A.; Araújo, J. ; Moreira, A. ; Castro, J. ; Alencar, F. and Penteado, R.: (2007a) “Um Padrão para Requisitos Duplicados”, in: 6th Latin American Conference on Pattern Languages of Programming (SugarLoafPlop’2007), Porto de Galinhas, Recife, Pernambuco , Brazil. (to appear)

Ramos, R. A. ; Alencar, F. ; Araújo, J. ; Moreira, A. ; Castro, J. and Penteado, R.: (2007b) “i* with Aspects: Evaluating Understandability”, in: Workshop on Requirements Engineering, 2007, Toronto, Canada. (to appear)

Rashid, A., Moreira, A., Araújo, J.: (2003) “Modularisation and composition of aspectual requirements”, in: Aksit, M., ed.: Proc. 2nd Int’ Conf. on Aspect-Oriented Software Development (AOSD-2003), ACM Press.

Rui, K., Ren, S., Butler, G.: (2003) “Refactoring use case models: A case study”, in: International Conference on Enterprise Information Systems. April, Angers – France.

Schneider, G., Winters, J.P.: (1998) “Applying Use Cases: A Practical Guide”, in: Addison Wesley - Object Technology Series.

Soares, S., Laureano, E., Borba, P.: (2002) “Implementing distribution and persistence aspects with AspectJ”, in: Proceedings of the 17th ACM conference on Object oriented programming, systems, languages, and applications, ACM Press 174–190.

Sommerville, I., P., S.: (1997) “Requirements Engineering: A good practice guide”, John Wiley and Sons LTD.

Sommerville, I.: (2004) “Software Engineering”, 7th edition. Pearson Education.

Wohlin, C., Runeson, P., H¨ost, M., Regnell, B., Wessl´en, A.: (2000) “Experimentation in Software Engineering: an Introduction”, Kluwer Academic Publishers.

Wiegers, K. E.: (2003) “Software Requirements”, Microsoft Press, Second Edition.

Xu, J., Yu, W., Rui, K., Butler, G.: (2004) “Use case refactoring: a tool and a case study”, in: Software Engineering Conference, 2004. 11th Asia-Pacific. 484–491.

Yu, W., Li, J., Butler, G.: (2004) “Refactoring use case models on episodes”, in: Automated Software Engineering, 2004. Proceedings 19th International Conference on 328–335.