Supporting Use-Case Reviews*

Alicja Ciemniewska, Jakub Jurkiewicz, Łukasz Olek, Jerzy Nawrocki

Poznań University of Technology, Institute of Computing Science,
ul. Piotrowo 3A, 60-965 Poznań, Poland
Alicja.Ciemniewska@gmail.com {Jakub.Jurkiewicz, Lukasz.Olek,
Jerzy.Nawrocki}@cs.put.poznan.pl

Abstract. Use cases are a popular way of specifying functional requirements of computer-based systems. Each use case contains a sequence of steps which are described with a natural language. Use cases, as any other description of functional requirements, must go through a review process to check their quality. The problem is that such reviews are time consuming. Moreover, effectiveness of a review depends on quality of the submitted document - if a document contains many easy-to-detect defects, then reviewers tend to find those simple defects and they feel exempted from working hard to detect difficult defects. To solve the problem it is proposed to augment a requirements management tool with a detector that would find easy-to-detect defects automatically.

1 Introduction

Use cases have been invented by Ivar Jacobson [13]. They are used to describe functional requirements of information systems in a natural language ([1], [4], [12], [5]). The technique is getting more and more popular. Use cases are extensively used in various software development methodologies, including Rational Unified Process [16] and XPrince [19].

Quality of software requirements, described as use cases or in any other form, is very important. The later the defect is detected, the more money it will cost. According to Pressman [22], correcting a defect in requirements at the time of coding costs 10 times more than correcting the same defect immediately, i.e. at the time of requirements specification. Thus, one needs quality assurance. As requirements cannot be tested, the only method is requirements review. During review a software requirements document is presented to interested parties (including users and members of the project team) for comment or approval [11]. The defects detected during review can be minor (not so important and usually easy to detect) or major (important but usually difficult to find). It has been noticed that if a document contains many easy-to-detect defects then review is less effective in detecting major defects. Thus, some authors proposed to split reviews into two stages (Knight calls them "phases" [15], Adolph uses term "tier"

* This work has been financially supported by the Ministry of Scientific Research and Information Technology grant N516 001 31/0269

[1]): the first stage would concentrate on finding easy-to-detect defects (e.g. compliance of a document with the required format, spelling, grammar etc.) and the aim of the second stage would be to find major defects. The first stage could be performed by a junior engineer or even by a secretary, while the second stage would require experts.

In the paper a mechanism for supporting use-case reviews, based on natural language processing (NLP) tools, is presented. Its aim is to find easy-to-detect defects automatically, including use-case duplication in the document, inconsistent style of naming use cases, too complex sentence structure in use cases etc. That will save time and effort required to perform this task by a human. Moreover, when integrating this mechanism with a requirements management tool, such as UC Workbench developed at the Poznan University of Technology ([20], [21]), one can get instant feedback on simple defects. That can help to learn good practices in requirements engineering and it can help to obtain a more 'homogeneous' document (that is important when requirements are collected by many analysts in a short time).

The idea of applying NLP tools to automate analysis of requirements is not new. First attempts were aiming at building semi-formal models ([3], [10], [24]) and detecting ambiguity ([17], [14], [18]) in "traditional" requirements. Requirements specified as use cases were subject of research done by Fantechi and his colleagues [8]. They have used three NLP tools (QuARS [7], ARM [23] and SyTwo) to automatically detect lexical and semantical ambiguity as well as too long and too complicated sentences in a requirements document specifying Nokia's FM radio player. Our work is oriented towards use-case patterns proposed by Adolph and others [1] and our aim is to extend UC Workbench with automatic detection of easy-to-dected defects.

The next section describes two main concepts used in the paper: use case and bad smell (a bad smell is a probable defect). Capabilities of natural language processing tools are presented in Section 3. Section 4 describes defects in use cases that can be detected automatically. There are three categories of such defects: concerning the whole document (e.g. use-case duplication), injected into a single use case (e.g. an actor appearing in a use case is not described), and concerning a single step in a use case (e.g. too complex structure of a sentence describing a step). A case study showing results of application of the proposed method to use cases written by 4th year students is presented in Section 5. The last section contains conclusions.

2 Use Cases and Bad Smells

Use cases are getting more and more popular way of describing functional requirements ([1], [4], [12]). In this approach, scenarios pointing up interaction between users and the system are presented using a natural language. Use cases can be written in various forms. The most popular are 'structured' use cases. An example of structured use case is presented in Figure 1. It consists of the main scenario and a number of extensions. The main scenario is a sequence of

steps. Each step describes an action performed by a user or by a system. Each extension contains an event that can appear in a given step (for instance, event 3.A can happen in Step 3 of the main scenario), and it presents an alternative sequence of steps (actions) that are performed when the event appears.

UC1. Search a product
Main Actor: Customer
Main Scenario:
1. Customer chooses search option.
2. System shows search box.
3. Customer enters search criteria, and asks for results.
4. System shows a list of found products.
5. Customer chooses one of the products.

Extensions:
3.A. Search criteria are invalid.
    3.A.1. System marks invalid fields, and asks for correction.
    3.A.2. Go back to step 3.

Fig. 1. An example of a use case in a structured form.

The term 'bad smell' was introduced by Kent Beck and it was related to source code ([9]). A bad smell is a surface indication that usually corresponds to a deeper problem in the code. Detection of a bad smell does not have to mean, that the code is incorrect; it should be rather consider as a symptom of low readability which may lead to a serious problem in the future.

In this paper, a bad smell is a probable defect in requirements. Since we are talking about requirements written in a natural language, in many situations it is difficult (or even impossible) to say definitely if a suspected defect is present or not in the document. For instance, it is very difficult to say if a given use case is a duplication of another use case, especially if the two use cases have been written by two people (such use cases could contain unimportant differences). Another example is complexity of sentence structure. If a bad smell is detected, a system displays a warning and the final decision is up to a human.

3 Natural Language Processing Tools for English

Among many other natural language processing tools, the Stanford parser [2] is the most powerful and useful tool from our point of view. The parser has a lot of capabilities and generates three lexical structures:

- probabilistic context free grammar (PCFG) structure - is a context-free grammar in which each production is augmented with a probability
- dependency structure - represents dependencies between words

– combined structure - is a lexicalized phrase-structure, which carries both category and (part-of-speech tagged) head word information at each node (Figure 2)

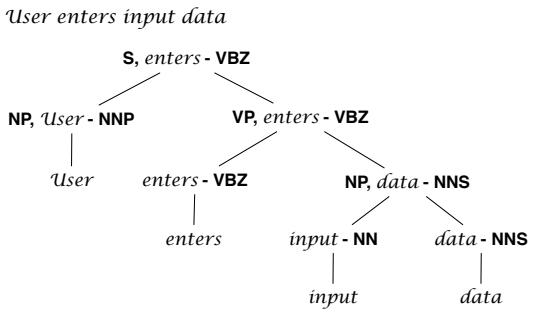

Fig. 2. An example of a Combined Structure generated by the Stanford parser.

The combined structure is the most informative and useful. To tag words it uses Penn Treebank set. As an example in Figure 2 we present structure of a sentence "User enters input data". In the example the following notation is used: S - sentence, NP - noun phrase , VP - verb phrase, NN - noun, NNP - singular proper noun, NNS - plural common non, VBZ - verb in third person singular.

Moreover, the Stanford parser generates grammatical structures that represents relations between individual pairs of words. Below in Figure 3 we present typed dependencies for the same example sentence, as in the above example. Notation used to describe grammatical relations are presented in [6].

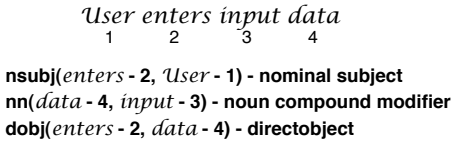

Fig. 3. An example of a typed dependencies generated by the Stanford parser.

The ambiguity of natural language and probabilistic approach used by the Stanford parser cause problems with automatic language analysis. During our research we have encountered the following problem. One of the features of the English language is that there are some words which can be both verbs and nouns. Additionally the third person singular form is composed by adding "s" to the end of the verb base form. In a similar way the plural form of a noun is built. This leads to the situation when the verb is confused with the noun. For example word "displays" can be tagged as a verb or a noun. Such confusion

may have great influence on the further analysis. An example of such situation is presented in Figure 4.

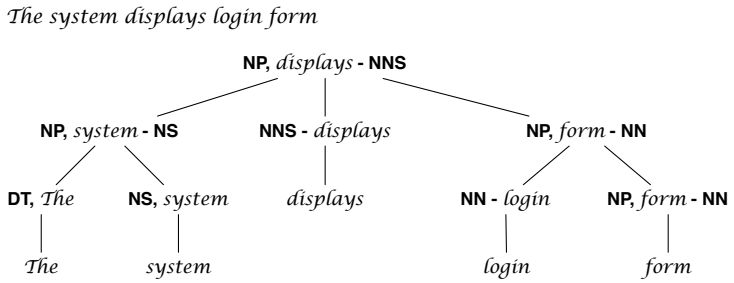

Fig. 4. An example of an incorrect sentence decomposition.

In our approach we want to give some suggestions to the analyst about how the quality of use cases can be improved. When the analyst gets the information about the bad smells, he can decide, whether these suggestions are reasonable or not. However, this problem does not seem to be crucial. In our research, which involved 40 use cases, we have found only three sentences in which this problem occurred.

4 Defects in Use Cases and their Detection

Adolph [1] and Cockburn [4] presented a set of guidelines and good practices about how to write effective use cases. In this context "effective" means clear, cohesive, easy to understand and maintain. Reading the guidelines one can distinguish several types of defects in use cases. In this Section we present those defects that can be automatically detected. Each defect discussed in the paper contains a description of automatic detection method. They have been split into three groups presented in separate subsections: specification-level bad smells (those are defect indicators concerning a set of use cases), use-case level bad smells (defect indicators concerning a single use case), and step-level bad smells (they concern a step - a use cases consists of many steps).

4.1 Specification-Level Bad Smells

At the level of requirements specification, where there are many use cases, a quite common defect which we have observed is use-case duplication. Surprisingly, we have found such defects even in specifications prepared by quite established Polish software houses. The most frequent is duplication by information object. If there are two different information objects (e.g. an invoice and a bill), analysts have a tendency to describe the processes which manipulate them as separate use cases, even if they are processed in the same way. That leads to unnecessary

thick documentation (the thicker the document, the longer the time necessary to read it). Moreover, it is dangerous. When someone finds and fixes a defect in one of the use cases, the other will remain unchanged, what can be a source of problems in the future. There are two sources of duplicated use cases:

- **Intentional duplication**. An analyst prefers that style and/or he wants to have a thick document (many customers still prefer thick documents - for them they look more serious and dependable, which is of course a myth). Some of such analysts perhaps will ignore that kind of warning, but some other - more proactive - may start to change their style of writing specification.
- **Unintentional duplication**. There are several analysts, each of them is writing his own part of the specification and before the review process no one is aware of the duplications. If this is the case, the ability to find duplicates in an automatic way will be perceived as very attractive.

Detection method is two-phased. In the first stage a signature (finger print) of each use case is computed. It can be a combination of a main actor identifier (e.g. its number) and a number of steps a use case contains. Usually a number of steps in a use case and a number of actors in the specification are rather small (far less than 256), thus a signature can be a number of the integer type. If two use cases have the same signature, they go through the second stage of analysis during which they are examined step by step. Step number j in one use case is compared against step number j in the second use case and so-called step similarity factor, s, is computed. Those similarity factors are combined into use-case similarity factor, u. If u is greater than a threshold then two use cases are considered similar and a duplication warning is generated.

A very simple implementation of the above strategy can be the following. We know that two similar use cases can differ in an information object (a noun). Moreover, we assume that most important information about processing an information object is contained in verbs and nouns. Thus, step similarity factor, si, for steps number i in the two compared use cases can be computed in the following way:

- If *all the corresponding verbs and nouns* appearing in the two compared steps *are the same*, then $s_i = 1$.
- If all the corresponding verbs are the same and *all but one corresponding nouns are the same* and a difference in the two corresponding nouns has been observed for the first time, then $s_i = 1$ and InfObject1 is set to one of the "conflicting" nouns and InfObject2 to the other (InfObject1 describes an information object manipulated with the first use case and InfObject2 is manipulated with the second use case).
- If all the corresponding verbs are the same and *all but one corresponding nouns are the same* and the conflicting nouns are InfObject1 in the first use case and InfObject2 in the second, then $s_i = 1$.
- In all other cases, s_i for the two analyzed steps is 0.

Use-case similarity factor, u, can be computed as a product of step similarity factors: $s_1 * s_2 ... * s_n$.

The described detection method is oriented towards *intentional duplication*. To make it effective in the case of *unintentional duplication* one would need a dictionary of synonyms. Unfortunately, so far we do not have any.

4.2 Use-Case Level Bad Smells

Bad smells presented in this section are connected with the structure of a single use case. The following bad smells have been selected to detect them automatically with UC Workbench, a use-case management tool developed at the Poznan University of Technology:

- **Too long or too short use cases.** It is strongly recommended [1] to keep use cases 3-9 steps long. Too long use cases are difficult to read and understand. Too short use cases, consisting of one or two steps, distract a reader from the context and, as well, make the specification more difficult to understand. To detect that bad smell it is enough to count the number of steps in each use case.
- **Complicated extension.** An extension is designed to be used when an alternative course of action interrupts the main scenario. Such an exception usually can be handed by a simple action and then it can come back to the main scenario or finish the use case. When the interruption causes the execution of a repeatable, consistent sequence of steps, then this sequence should be extracted to a separate use case (Figure 5). Detection can be based on counting steps within each extension. A warning is generated for each extension with too many steps.
- **Repeated actions in neighboring steps.**
- **Inconsistent style of naming**

The last two bad smells will be described in the subsequent subsections.

Repeated Actions in Neighboring Steps Every step of a use case should represent one particular action. The action may consist of one or more moves which can be taken as an integrity. Every step should contain significant information which rather reflect user intent then a single move. Splitting these movements into separate steps may lead to long use cases, bothersome to read and hard to maintain.

Detection method: Check whether several consecutive steps have the same actor (subject) and action (predicate). Extraction of subject and predicate from the sentence is done by the Stanford parser. The analyst can be informed that such sequence of steps can be combined to a single step.

Inconsistent Style of Naming Every use case should have a descriptive name. The title of each use case presents a goal that aprimary actor wants to achieve.

Main Scenario:
1. System switches to on-line mode and displays summary information about data that have to be uploaded and downloaded.
2. User confirms action.
3. System executes action.

Extensions:
1.A. TMS in unreachable.
    1.A.1. System shows information that there is no connection to TMS.
1.B. There is no data to synchronize.
    1.B.1. System shows information that no data have to be synchronized.
    1.B.2. End of use case.
2.A. TMS does not recognize user's login and password.
    2.A.1. System displays information about the problem and shows the login form.
    2.A.2. User fills the form.
    2.A.3. System saves new data.
    2.A.4. Go to step 2.

Fig. 5. Example of a use case with too complicated extension (bolded).

Wrong:
~~*1. Administrator fills in his user name*~~
~~*2. Administrator fills in his telephone number*~~
~~*3. Administrator fills in his email*~~

Correct:
Administrator fills in his user name, telephone number and email

Fig. 6. Example of repeated actions in neighboring steps

There is a few conventions of naming use cases, but it is preferable to use active verb phrase in the use case name. Furthermore, chosen convention should be used consistently in all use cases.

Wrong: ~~*Title: Main Use Case*~~
Correct: *Title: Buy a book*

Fig. 7. Example of inconsistent style of naming

Detection method: Approximated method of bad smell detection in use case names, is to check whether use case name satisfies the following constraints:

- The title contains a verb in infinitive (base) form
- The title contains an object

This can be done using the Stanford parser. If the title does not fulfill these constraints, a warning is attached to the name.

4.3 Step-Level Bad Smells

Bad smells presented in this section are connected with use-case steps. Steps occur not only in the main scenario, but also in extensions. The following bad smells are described here:

Too Complex Sentence Structure The structure of a sentence used for describing each step of use case should be as simple as possible. It means that it should generally consists of a subject, a verb, an object and a prepositional phrase ([1]). With such a simple structure one can be sure that the step is grammatically correct and unambiguous. Because of the simplicity, use cases are easy to read and understand by readers. Such a simple structure helps a lot when using natural language tools. It is essential to notice that the simpler the sentence is, the more precisely other bad smells can be discovered. An example of a too complex sentence and its corrected version is presented below.

Wrong: ~~*The system switches to on-line mode and displays summary information about data that have to be uploaded and downloaded*~~
Correct: *The system displays list of user's tasks*

Fig. 8. Examples of too complex sentence structure

Detection method: Looking at the use cases that were available to us we have decided to build a simple heuristic that checks whether a step contains more

than one sentence, subject or predicate, coordinate clause, or subordinate clause. If the answer is YES, then a warning is generated. Obviously, it is just a heuristic. It can happen that a step contains one of the mentioned defect indicator, but it is still readable and easy to understand. Providing a clear and correct answer in all possible cases is impossible - even two humans can differ in their opinion what is simple and readable. In our research we have encountered some examples of this problem. However, we have distinguished some rules, which can be used to verify, whether a sentence is too complex and unreadable. Below we present the verification rules. The numbers in the brackets show applicability of a rule (in how many use cases a given rule could be applied) and its effectiveness (in how many use cases it has found a real defect approved by a human).

- step contains more than one sentence (2 / 2)
- step contains more than one subject or predicate (2 / 2)
- step contains more than one coordinate clause (8 / 4)
- step contains more than one subordinate clause (7 / 5)

Lack of the Actor According to [1] it should be always clearly specified who performs the action. The reader should know which step is performed by which actor. Thus, every step in a use case should be an action that is performed by one particular actor. Actor's name ought to be the subject of the sentence. Thus, in most cases the name should appear as the first or second word in the sentence. Omission of actor's name from the step may lead to a situation in which the reader does not know who is the executor of the action.

Wrong: ~~*The form is filled in*~~
Correct: *Student fills in the form*

Fig. 9. Example of lack of the actor

Detection method: In the case of UC Workbench, every actor must be defined and each definition is kept within the system. Therefore it can be easily verified whether the subject of a sentence describing a step is defined as an actor.

Misusing Tenses and Verb Forms A frequent mistake is to write use cases from the system point of view. This is easier for the analyst to write in such a manner, but the customer may be confused during reading. Use cases should be written in a way which is highly readable for everyone. Therefore the action ought to be described from the user point of view. In order to ensure this approach, the present simple tense and active form of a verb should be used. Moreover, the present simple tense imply that described action is constant and the system should always respond in a determined way.

Wrong: ~~*Send the message*~~
Wrong: ~~*System is sending an emai*~~
Wrong: ~~*The email is sent by System*~~
Correct: *System sends an email*

Fig. 10. Example of misusing tenses and verb forms

Detection method: Using combined structure from the Stanford parser, the nsubj relation [6] can be determined. It can be checked if the subject is an actor and the verb is in the third person singular active form.

Using Technical Jargon Use case is a way of describing essential system behavior, so it should focus on thefunctional requirements. Technical details should be kept outside of the functional requirements specification. Using technical terminology might guide developers towards specific design decisions. Graphical user interface (GUI) details clutter the story, make reading more difficult and the requirements more brittle.

Wrong: ~~*User chooses the second tab and marks the checkboxes*~~
Correct: *User chooses appropriate options*

Fig. 11. Example of using technical jargon

Detection method: We should create a dictionary containing terminology typical for specific technologies and user interface (e.g. button, web page, database, edit box). Then it is easy to check whether the step description contains them or not.

Conditional Steps A sequence of actions in the use case depends on some conditions. It is natural to describe such situation using conditionals "if condition then action ...". This approach is preferred by computer scientists, but it can confuse the customer. Especially it can be difficult to read when nested "if" statement is used in a use case step. Use cases should be as readable as possible. Such a style of writing makes it complex, hard to understand and follow.

It is preferable to use the optimistic scenario first (main scenario) and to write alternative paths separately in the extension section.

Detection method: The easiest way to detect this bad smell is to look for specific keywords, such as if, whether, when. Additionally, using the Stanford parser it can be checked that the found keyword is a subordinating conjunction. In such a case the analyst can be notified that the step should be corrected.

Wrong:
~~*1. Administrator types in his user name and password*~~
~~*2. System checks if the user name and the password are correct*~~
~~*3. If the given data is correct the system logs Administrator in*~~
Correct:
1. Administrator types in his user name and password
2. System finds that the typed-in data are correct and logs Administrator in
Extensions:
2.A. The typed-in data are incorrect
2.A.1. System presents an error message

Fig. 12. Example of conditional steps

5 Case Study

In this section we would like to present an example of applying our method to a set of real use cases. These use cases were written by 4th year students participating in the Software Development Studio (SDS) which is a part of the Master Degree Program in Software Engineering at the Poznan University of Technology. SDS gives the students a chance to apply their knowledge to real-life projects during which they develop software for real customers.

Let us consider a use case presented in Figure 14. Using the methods presented in Section 4 the following bad smells can be detected:

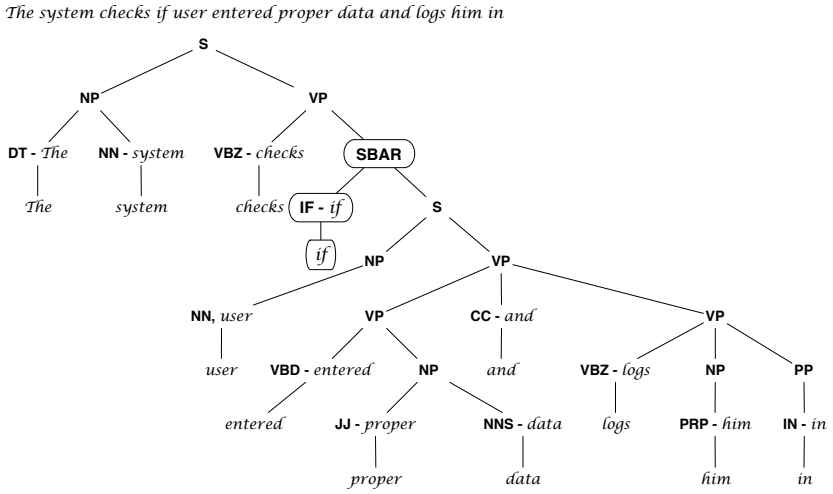

Fig. 13. Example of a use case that contains a condition.

Misusing Tenses and Verb Forms

- **Step:** *3. User fill the form*
- **Tagging** (from the Stanford parser): *User*/NNP *fill*/VBP *the*/DT *form*/NN
  NNP - singular proper noun
  VBP - base form of auxiliary verb
  DT - determiner
  NN - singular common noun
- **Typed dependencies** (from the Stanford parser):
  nsubj(*fill*-2, *User*-1) - nominal subject
  det(*form*-4, *the*-3) - determiner
  dobj(*fill*-2, *form*-4) - direct object
- **Conclusion:**
  From the typed dependencies we can determine the *nsubj* relation between *User* and *fill*. From the tagging it can be observed that *fill* is used in wrong form (proper form would be VBZ - verb in third person singular form).

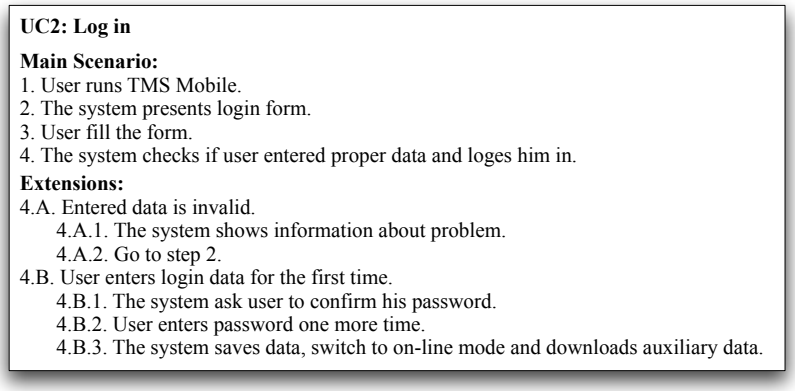

Fig. 14. A use case describing how to log in to the TMS Mobile system.

Conditional Step

- **Step:** *4. The system checks if user entered proper data and loges him in*
- **Tagging** (from the Stanford parser): *The*/DT *system*/NN *checks*/VBZ *if*/IN *user*/NN *entered*/VBD *proper*/JJ *data*/NNS *and*/CC *loges*/VBZ *him*/PRP *in*/IN
  VBZ - verb in third person singular form
  IN - subordinating conjunction
  VBD - verb in past tense
  JJ - adjective

NNS - plural common noun
PRP - personal pronoun
- **Combined structure** (from the Stanford parser): Presented in Figure 13
- **Conclusion:**
  As it can be observed the step contains the word *if*. Moreover from the combined structure we can conclude that the word *if* is subordinating conjunction.

Complicated Extensions

- **Extension:** *4.b User enters login data for the first time*
- **Symptom:** The extension contains three steps.
- **Conclusion:** The extension scenario should be extracted to a separate use case.

6 Conclusions

So far about 40 use cases have been examined using our methods of detecting bad smells. Almost every use case from the examined set, contained a bad smell. Most common bad smells were: Conditional Step, Misusing Tenses and Verb Forms and Lack of the Actor. Thus, this type of research can contribute to higher quality of requirements specification.

In the future it is planned to extend the presented approach to other languages, especially to Polish which is mother tongue to the authors. Unfortunately, Polish is much more difficult for automated processing and there is lack of appropriate tools for advanced analysis.

Acknowledgements

First of all we would like to thank the students involved in the UC Workbench project. We would like to thank the IBM company for awarding Eclipse Innovation Grant to UC Workbench project. It allowed students focus on the development work. This research has been financially supported by the Ministry of Scientific Research and Information Technology grant N516 001 31/0269.

References

1. Steve Adolph, Paul Bramble, Alistair Cockburn, and Andy Pols. Patterns for Effective Use Cases. Addison-Wesley, 2002.
2. Advances in Neural Information Processing Systems 15. Fast Exact Inference with a Factored Model for Natural Language Parsing, 2003.
3. Vincenzo Ambriola and Vincenzo Gervasi. Processing natural language requirements. In Automated Software Engineering, pages 36–45. IEEE Press, 1997.
4. Alistair Cockburn. Writing Effective Use Cases. Addison-Wesley, 2001.

5. Larry L. Constantine and Lucy A. D. Lockwood. Software for use: a practical guide to the models and methods of usage-centered design. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.
6. Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In LREC, 2006.
7. F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. The linguistic approach to the natural language requirements quality: benefit of the use of an automatic tool. In Software Engineering Workshop. Proceedings. 26th Annual NASA Goddard, pages 97–105, 2001.
8. Alessandro Fantechi, Stefania Gnesi, G. Lami, and A. Maccari. Application of linguistic techniques for use case analysis. In RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering, pages 157–164, Washington, DC, USA, 2002. IEEE Computer Society.
9. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.
10. Zbigniew Huzar and Marek Łabuzek. A tool assisting creation of business models. Foundations of Computing and Decision Sciences, 27(4):227–238, 2002.
11. IEEE. Ieee standard for software reviews (ieee std 1028-1997), 1997.
12. Ivar Jacobson. Use cases - yesterday, today, and tomorrow. Technical report, Rational Software, 2002.
13. Ivar Jacobson. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 2004.
14. E. Kamsties and B. Peach. Taming ambiguity in natural language requirements. In ICSSEA, Paris, December 2000.
15. John C. Knight and E. Ann Myers. An improved inspection technique. Commun. ACM, 36(11):51–61, 1993.
16. Per Kroll and Philippe Kruchten. The rational unified process made easy: a practitioner's guide to the RUP. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
17. B. Macias and S. G. Pulman. Natural language processing for requirement specifications. In Safety Critical Systems. Chapman and Hall, 1993.
18. L. Mich and R. Garigliano. Ambiguity measures in requirement engineering. In Int. Conf. On Software Theory and Practice, Beijing, China, August 2000.
19. Jerzy Nawrocki, Michał Jasiński, Bartosz Paliświat, Łukasz Olek, Bartosz Walter, Błażej Pietrzak, and Piotr Godek. Balancing agility and discipline with xprince. In Proceedings of RISE 2005 Conference (in print), volume 3943 of LNCS, pages 266 – 277. Springer Verlag, Jan 2006.
20. Jerzy Nawrocki and Łukasz Olek. Uc workbench - a tool for writing use cases. In 6th International Conference on Extreme Programming and Agile Processes, volume 3556 of LNCS, pages 230–234. Springer Verlag, Jun 2005.
21. Jerzy Nawrocki and Łukasz Olek. Use-cases engineering with uc workbench. In Krzysztof Zieliński and Tomasz Szmuc, editors, Software Engineering: Evolution and Emerging Technologies, volume 130, pages 319–329. IOS Press, oct 2005.
22. R. Pressman. Software Engineering - A Practitioners Approach. McGraw-Hill, 2001.
23. W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt. Automated analysis of requirement specifications. In Proceedings of the 1997 (19th) International Conference on Software Engineering, pages 161–171, 1997.
24. Marek Łabuzek. Modelling the meaning of descriptions of reality to improve consistency between them and business models. Foundations of Computing and Decision Sciences, 29(1-2):89–101, 2004.