DOCUMENTS PROCESSING IN THE REQUIREMENTS MANAGEMENT SYSTEM

Tomasz Wydra[1], Konrad Grzanek[2]

[1]Academy of Management, Łódź, Poland
tomasz.wydra@hotmail.com

[2] IT Institute, Academy of Management, Łódź, Poland
kgrzanek@spoleczna.pl, kongra@gmail.com

Abstract

Requirements analysis is a highly critical step in software life-cycle. Our solution to the problem of managing requirements is an embedded domain-specific language with Clojure playing the role of the host language. The requirements are placed directly in the source code, but there is an impedance mismatch between the compilation units and electronic documents, that are the major carriers of requirements information. This paper presents a coverage for this problem.

Key words: Requirements engineering, knowledge discovery, documents processing, PDF

1 Source-code Oriented Requirements Management

According to surveys (like [1]) on software quality, software development is still more art than science. Researches point out poor requirements management as one of the most important factors when discussing the possible reasons. Requirements analysis is a highly critical step in software life-cycle [2], [3]. The proper and effective requirements management saves the overall project costs due to the following reasons:

- Requirement errors typically cost well over 10 times more to repair than other errors.
- Requirement errors typically comprise over 40% of all errors in a software project.
- Small reductions in the number of requirement errors pay big dividends in avoided rework costs and schedule delays.

Moreover, the requirement managements errors are the most common errors in software projects.

Our previous work [3] describes a requirements management system for the Clojure programming language [4]. The system allows a programmer to put the requirements directly into the source code and to manage them using standard compilation techniques as well as doing it interactively using Lisp REPL (Read-Eval-Print Loop). Our unique approach is inspired by a homoiconicity of the languages from the Lisp family of programming languages (as stated in [3]). Our solution is an embedded domain-specific language with Clojure playing the role of the host language. This DSL wins the following for the analysts, designers and programmers (after [3]):

- Editing source code is a primary activity every programmer undertakes on every work-day. Putting the act of reading/writing the requirements into source code increases the comfort of this – sometimes boring – activity.
- It also affects the designers and other people not involved directly in the implementation phase, because it opens an effective channel of communication between – for instance – a system analyst and a coder; the analyst writes a requirement directly in a compilation unit, the programmer reads it and perform further steps to gain the required functionality.
- The presence of requirements in compilation units allows to interweave them (their definitions formally speaking) with source code snippets being their direct implementations or implementation parts. This point is especially important because an act of locating requirements in pure (not instrumented with requirements or requirement-related tags) source code is a tedious and hard to solve problem. Further works on this can be found in [5, 6].
- A compilation unit keeping some requirements may be tracked and managed by a source management and revision control system, such as Git [7]. An immediate consequence is the ability to manage the requirements versions, because a requirement change is a change in the compilation unit. All version control system's goodies, including the possible encryption and the overall robustness of a distributed versioning system are there to be used.

2 The Requirements and Implementations Abstraction

The model of the requirements management system presented here consists of two major types of data called ***REQ-infos*** and ***IMPL-infos***. Objects of those types are essentially maps with the properties as specified in the following tables:

Table 1. REQ-info properties

:label	A label of the requirement.
:doc	A documentation part of the requirement describing some feature or presenting some fact.
:file	A name of the compilation unit (source file) in which the requirement was defined.
:ns	A Clojure name-space in which the requirement was defined.
:line	The line number in ***:file*** in which the requirement was defined.
:score	A search score. The property is present only in the requirement records retrieved via querying.

Table 2. IMPL-info properties

:label	A label of the requirement for which the implementation was defined.
:file	A name of the compilation unit (source file) in which the implementation was defined.
:ns	A Clojure name-space in which the implementation was defined.
:line	The line number in :file in which the implementation was defined.
:score	A search score. The property is present only in the implementation records retrieved via querying[1].

REQ-info objects (***REQ-infos***) represent requirements and the ***IMPL-infos*** – the implementations. Mutual relations between the objects of both kinds form a graph. The graph may undergo further processing and searching to build a more comprehensive overview of what has to be implemented and what has already been achieved, especially when the number of requirements reaches thousands (not even mentioning number of ***IMPL-infos*** in such case).

1 Indexing and querying the REQ-infos and IMPL-infos is performed using the Lucene text search engine. More on this can be found in [3].

The relations between REQ-infos and IMPL-infos are textual and – at least by now – cannot be created fully automatically. The most important notion is a ***label***, a textual identifier representing the ***REQ-info***.

3 Labels and References

To define a ***REQ-info*** inside a source file one has to use a `(defr …)` form that comes with the library. It's list of arguments comes as follows:

```
[doc & {:keys [labels refs do]
        :or   {labels [] refs [] do []}
        :as   options}]
```

The ***doc*** argument is the textual content of a requirement, a documentation part most of the time. ***do*** is a collection of actions to perform on the compile-time, when processing the ***defr*** macro instance, with the (probably) most important ***persist*** action. An example use can be seen below[2]:

```
(REQ[3]/defr
  "4.1 The Kinds of Types and Values

  There are two kinds of types in the Java programming language:
  primitive types (§4.2) and reference types (§4.3). There are,
  correspondingly, two kinds of data values that can be stored
  in variables, passed as arguments, returned by methods, and
 operated on: primitive values (§4.2) and reference values
 (§4.3).

     Type:
             PrimitiveType
             ReferenceType"

  :do [REQ/persist])
```

The example above is a requirement that describes a tiny part of Java type-system coming from The Java Language Specification, Third Edition [8]. In this case a ***label*** is automatically generated for a ***REQ-info***, but in general a programmer or a system engineer is allowed to pass a vector of custom labels. This has the following desirable consequences:

– A label is an identity of a requirement.

2 The REQ library was a subject of substantial changes since it's version described in [3]. This is why the form has a different shape than the ones in the mentioned paper.

3 Possible assuming a form `(:require [kongra.req :as REQ])` was evaluated earlier in the name-space header.

- A ***REQ-info*** may be a description of a whole bunch of requirements, when it contains a number of labels larger than one.
- A requirement may be represented by a collection of REQ-infos and spread all over a large number of places in the code, possibly separate compilation units.
- So the cardinality of a relation between a requirements and ***REQ-infos*** is many-to-many.

Another important, but not explicit property of the ***REQ-info*** is it's collection of references. The ***references*** are labels of the other ***REQ-infos*** referenced from the defined one. The form `(defr …)` possesses a separate parameter called *refs*, but it does not map directly to a ***REQ-info*** property. Instead the REQ library contains some indirect containers for references avoiding an unnecessary *Lucene* indexation of references when storing ***REQ-infos***[4].

Labels and textual content (***doc***) together with ***references*** form a complete graph of informally formulated requirements. Informally because the portions of information are highly human-dependent and their nature does not undergo any regulations other than the syntactic ones. One way to get closer to some automatism in this regard is to:

- Allow the system to generate the labels – a default behavior of the (defr…) form.
- Use the NLP tools to extract dictionary words (1-grams) and then 2-grams, 3-grams and – in general – N-grams out of the doc part of ***REQ-infos*** to build collections of **refs**.

Realization of the two postulates will be a subject of some further work.

4 PDF Processing and the Extraction of Textual Content

Although the system allows and encourages the software engineers to put all requirements (and implementations) information into the compilation units, initially these portions of textual content are placed in some kind of electronic documents, with the PDF format being the most widely used. This section describes one possible solution for the problem of format incompatibility arising between Clojure compilation units and PDF documents. To be more precise here, it presents a way to transform an example portion of textual information present in a document into a shape acceptable from the point of view of creating a `(defr…)` form in a compilation unit.

We have chosen a PDF document processing library for Java called ***iText*** [9]. This library can include extracting information from the document. It

4 This could also be achieved by configuring Lucene. In one way or the other REQ treats references as a kind of the second-class citizens (derivatives) when talking about ***REQ-infos*** model.

operates on the so-called tokens that are tagged to the types of stored information. The following table provides a list of tokens which iText operates on [10]:

Table 3. Overview of the token types

Token type	Symbol	Description
NUMBER		The current token is a number.
STRING	( )	**The current token is a string.**
NAME	/	**The current token is a name.**
COMMENT	%	**The current token is a comment.**
START_ARRAY	[	**The current token starts an array.**
END_ARRAY	]	The current token ends an array.
START_DIC	<<	The current token starts a dictionary.
END_DIC	>>	**The current token ends a dictionary.**
REF	R	**The current token ends a reference.**
OTHER		The current token is probably an operator.
ENDOFFILE		There are no more tokens.

The technique used to process the pdf document is to check token by token for the specified pattern. The Xournal editor [11] was used to highlight data to be extracted either as a ***REQ-info doc***, ***label(s)*** or ***ref(s)***, whith the tool called highlighter. Sample selection can be seen at the following Figure 1.

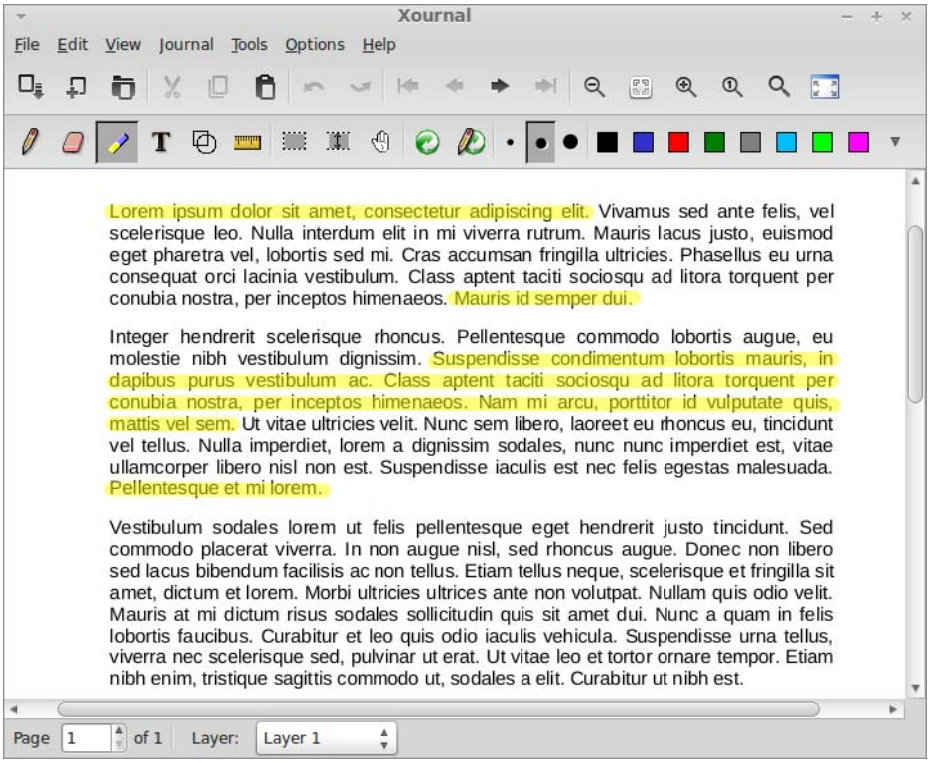

Figure 1. Text highlighted in Xournal editor.

Figure pattern (selection) that editor creates is shown below (1):

$$gs\ x_1\ y_1\ m\ x_2\ y_2\ l\ x_3\ y_3\ l\ \ldots\ l\ x_n\ y_n\ l\ S \tag{1}$$

where:

gs – represents the beginning of the figure
x, y – coordinates of subsequent elements of figure
m, l – tokens specifying circumscribe

The algorithm realizing searching the document is shown at Listing 1.

```
for (int i = 1; i <= pageNumber; i++) {
  byte[] pageBytes = reader.getPageContent(i);
  PRTokeniser tokeniser = new PRTokeniser(pageBytes);
  TokenType tokenType;
  String tokenValue = null;
  int startOfShape = 0;     // 0 = shape not founded,
                            // 1 = the pointer passed
                            //     start of shape
```

```
    while (tokeniser.nextToken()) {
      tokenType = tokeniser.getTokenType();
      tokenValue = tokeniser.getStringValue();
      if (tokenType == PRTokeniser.TokenType.NUMBER) {
        buf.add(tokenValue);
      } else if (tokenType == PRTokeniser.TokenType.OTHER
                  & tokenValue.equals("m")
                  & startOfShape == 0) {
        highlighted.add(buf.get(buf.size() - 2));
        highlighted.add(buf.get(buf.size() - 1));
        startOfShape = 1;
      } else if (tokenType == PRTokeniser.TokenType.OTHER
                  & tokenValue.equals("S")
                  & startOfShape == 1) {
        highlighted.add(buf.get(buf.size() - 2));
        highlighted.add("" + pageNumber);
        startOfShape = 0;
      }
    }
  }
```

Listing 1. Searching for appropriate tokens

If a pattern is found (1), an index is created which stores information about the location of selection (coordinates with page number) (2):

$$x_1, y_1, x_2, \text{pageNumber}, \ldots, x_n, y_n, x_{n+1}, \text{pageNumber} \quad (2)$$

where:

x_1, - x-coordinate of the beginning of the selection
y_1, - y-coordinate of the beginning of the selection
x_2, - x-coordinate of the end of the selection
pageNumber, - number of page in the document where the selection is placed

After searching the entire document, the method returns a list of the form (2), which is used to build the filter. The filter is a rectangle (position) collected sequentially from the index (2). Then the data are retrieved from the separated area and resulting file is created that contains content originally selected in the PDF document. The method used is shown at Listing 2

```
for (int i = 0; i < highlighted.size(); i+=4) {
  int x1 = (int)Double.parseDouble(highlighted.get(i));

  int y = (int)Double.parseDouble(highlighted.get(i+1));
  int x2 = (int)Double.parseDouble(highlighted.get(i+2));
  int pageNumber =
            (int)Integer.parseInt (highlighted.get(i+3));
  int fontSize = 12;
```

```
  Rectangle size = reader.getPageSize(pageNumber);
  Rectangle rect = new Rectangle(
                       x1, size.getTop()-y-fontSize,
                       x2, size.getTop() - y);

  RenderFilter filter = new RegionTextRenderFilter(rect);
  TextExtractionStrategy strategy;
  strategy = new FilteredTextRenderListener(
                  new LocationTextExtractionStrategy(),
                  filter);

  out.println(PdfTextExtractor.getTextFromPage(
                       reader, pageNumber, strategy));
}
```

Listing 2. Extracting info from selected areas

Below there is an attached screenshot showing the output file (result) of the program on the document with the selected text (Figure 2).

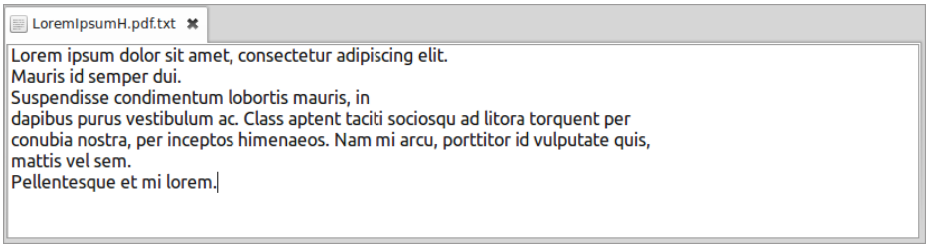

Figure 2. Text document generated from selection

5 Summary

The goal of the paper was to provide a reader with an insight into the essentials of the REQ library, into the directions in which it is going to be enhanced and presented a very interesting solution to a problem of "impedance mismatch" between the compilation units and electronic documents, that are the major carriers of requirements information. The method covers the problem, yet evaluating it's effectiveness, from the point of view of a programmer or requirements engineer should (and will be) be a subject of some further research.

References

1. Davis A. M., Leffingwell D. A., 1995, *Using Requirements Management to Delivery of Higher Quality Applications*, Rational Software Corporation
2. Dardenne A., van Lamsweerde A., Fickas S., 1993, *Goal-directed Requirements Acquisition*, Science of Computer Programming, Vol. 20, pp. 3-50
3. Grzanek K., *Source Code-Oriented Requirements Management*, Computer Methods in Practice, A. Cader., M. Jacymirski, K. Przybyszewski (eds.), Academic Publishing House EXIT, 2012, Warszawa, pp. 21-45
4. *The Clojure Language Website*, 2013, http://clojure.org
5. Eisenbarth T., Koschke R., Simon D., 2003, *Locating Features in Source Code*, IEEE Transactions on Software Engineering, pp. 210-224
6. Eaddy M., Aho A.V., Antoniol G., Gueheneuc Y.G., 2008, CERBERUS: *Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis*, ICPC 2008. The 16th IEEE International Conference on Program Comprehension, pp. 53-62
7. *Git, Website* 2013, http://git-scm.com/
8. Oracle Technology Network, 2013, *The Java Language Specification*, http://docs.oracle.com/javase/specs/
9. *iText, Website* 2013, http://itextpdf.com/
10. Lowagie B., 2007, *iText in Action, Creating and Manipulating PDF*, Manning Publications Co., ISBN 1932394796
11. *Xournal, Website* 2013, http://xournal.sourceforge.net/