Poznań University of Technology
Faculty of Computing

Automatic generation of user manual for web applications

Bartosz Alchimowicz

A dissertation
submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy.

Supervisor:	Jerzy Nawrocki, PhD, Dr Habil.
Co-supervisor:	Mirosław Ochodek, PhD

Poznań, Poland, 2015

ABSTRACT

Context: Elaboration of a good quality user documentation is a time consuming task. Any inconsistency between user documentation and software has a number of negative implications. Users often perceive errors or omissions in user documentation as errors in the application, which results in bad opinion about the product and increases costs of support (according to Mike Markel, the cost of one support call in 2008 was about $32).
User documentation comprises not only user manual but also explanations, including explanations of syntax of fields in web application forms, often called field explanations, for short. These explanations usually have the form of short messages (for example, in HTML5, field explanation is given as attribute `title` of tag `input`).
Time pressure negatively affects quality of user documentation. Thus, the questions arises, whether it is possible to help software vendors by automating at least part of the work associated with the creation of user documentation.

Objective: The aim of this thesis is to investigate the possibility of automatic generation of user documentation whose quality were similar to content created by a human.

Method: Evaluation criteria, commonly referred to as a quality model, are needed to determine whether the quality of an automatically generated document is similar to the quality of a corresponding document prepared by a human. Literature review revealed that the existing quality models for manuals are too detailed and too lengthy (e.g. ISO standards 26512, 26513, and 26514 contain about 700 low-level control questions, which can be regarded as evaluation criteria). Therefore, a new quality model for user manual was proposed, called COCA, which comprises four criteria: Completeness, Operability, Correctness, and Appearance. For quantitative evaluation of operability the *Documentation Evaluation Test* method (DET) has been used, which is based on *Browser Evaluation Test.*
To check feasibility of automatic generation of user documentation prototypes of the two tools have been developed:

- a field explanation generator (an input to the generator is a regular expression describing the syntax of the field), and
- a user manual generator (this generator uses a project's business case, requirements specification, including use cases, acceptance tests, and a working software).

The quality of the generated manual was evaluated using the COCA model and the DET method, while the quality of generated explanations was empirically investigated using the understandability criterion.

Results: The outcome of the research can be summarized in the following statements:

- It is possible to automatically generate field explanations whose quality is no worse than quality of a description provided by a human. In the conducted experiments 84.0% of correct answers have been obtained for field explanations generated by the prototype tool and less than 79% for field explanations written by humans.
- It is possible to generate a user manual whose quality is no worse than a manual written by a human. In the experiment based on the commercial system *Plagiat.pl* percentage of correct answers for the user manual generated by the prototype tool was about 85% and for the original user manual it was almost 83%. The average time spent by a subject on searching the user manual to find the answer to a given question was 2:41 min for the prototype tool, and 2:27 min in the case of the original manual. Although, the generated manual proved "slower", the difference of 10% seems neglectable.

Conclusion:
It is possible to automate significant part of work related to the creation of user documentation. However, it requires maintaining good quality requirements specification and scripts for automated acceptance testing.

Politechnika Poznańska
Wydział Informatyki

Automatyczne generowanie instrukcji obsługi dla aplikacji internetowych

Bartosz Alchimowicz

Rozprawa doktorska

Promotor:	dr hab. inż. Jerzy Nawrocki
Promotor pomocniczy:	dr inż. Mirosław Ochodek

Poznań, 2015

STRESZCZENIE

Kontekst: Opracowanie dobrej dokumentacji użytkownika jest czasochłonne, a ewentualne niezgodności dokumentacji z oprogramowaniem niosą szereg negatywnych implikacji. Braki lub błędy w dokumentacji są często odbierane przez użytkowników jako błędy w aplikacji, co skutkuje złą opinią o produkcie i prowadzi do wzrostu kosztów wsparcia technicznego (według Mike'a Markela koszt wsparcia telefonicznego dla jednego zgłoszenia wynosił w 2008 roku około $32).
Dokumentacja użytkownika obejmuje oprócz instrukcji obsługi (podręcznika) także objaśnienia, w tym objaśnienia dotyczące składni pól w formularzach aplikacji internetowych, zwane krótko objaśnieniami pól. Objaśnienia te mają zazwyczaj formę krótkich komunikatów (na przykład w języku HTML5 objaśniania pól podaje się jako atrybut `title` znacznika `input`).
Presja czasu negatywnie wpływa na jakość tworzonej dokumentacji użytkownika. Powstaje zatem pytanie, czy można pomóc producentom oprogramowania poprzez automatyzację choćby części prac związanych z tworzeniem dokumentacji użytkownika.

Cel: Celem pracy jest zbadanie możliwości automatycznego generowania dokumentacji użytkownika o jakości zbliżonej do materiałów opracowywanych przez człowieka.

Metoda: Aby móc stwierdzić czy jakość automatycznie wygenerowanego dokumentu jest zbliżona do jakości analogicznego dokumentu wytworzonego przez człowieka potrzebne są kryteria oceny, których zbiór nazywa się potocznie modelem jakości. Analiza literatury wykazała, że istniejące modele jakości dla instrukcji obsługi są zbyt drobiazgowe (np. standardy ISO 26512, 26513 i 26514 zawiera około 700 bardzo szczegółowych pytań kontrolnych, które można by traktować jako kryteria oceny). Dlatego też opracowano nowy model jakości dla instrukcji obsługi, nazwany COCA, obejmujący cztery kryteria: kompletność (ang. *Completeness*), operowalność (ang. *Operability*), poprawność (ang. *Correctness*) i wygląd (ang. *Appearance*). Dla ilościowej oceny operowalności wykorzystano metodę *Documentation Evaluation Test* (DET) bazującą na *Browser Evaluation Test*.
Zbadanie możliwości automatycznego generowania dokumentacji użytkownika polegało na opracowaniu prototypów dwóch narzędzi:

- generatora objaśnień pól (wejściem dla generatora jest wyrażenie regularne opisujące składnię pola) i
- generatora instrukcji użytkownika (generator ten wykorzystuje w swoim działaniu uzasadnienie biznesowe projektu, specyfikację wymagań obejmującą przypadki użycia, testy akceptacyjne oraz działające oprogramowanie).

Jakość wygenerowanej instrukcji obsługi oceniono w oparciu o model COCA i metodę DET, a jakość generowanych objaśnień w oparciu o jedno kryterium jakim jest zrozumiałość.

Wyniki: Z przeprowadzonych badań wynikają następujące wnioski:

- Możliwe jest automatyczne generowanie objaśnień pól o jakości nie gorszej niż opisy opracowywane przez człowieka. W przeprowadzonych eksperymentach uzyskano 84.0% poprawnych odpowiedzi w oparciu o automatycznie wygenerowane objaśnienia pól i mniej niż 79% poprawnych odpowiedzi w oparciu o objaśnienia napisane przez uczestników eksperymentu.
- Możliwe jest wygenerowanie instrukcji obsługi o jakości nie gorszej niż instrukcja opracowywana przez człowieka. W eksperymencie bazującym na komercyjnym systemie *Plagiat.pl*, procent poprawnych odpowiedzi dla instrukcji obsługi wygenerowanej przez prototyp wynosił około 85%, a dla oryginalnej instrukcji obsługi wynosił prawie 83%. Średni czas wyszukania odpowiedzi w instrukcji obsługi na pytanie zadane uczestnikom eksperymentu wynosił 2:41 min dla prototypowego narzędzia, i 2:27 min dla oryginalnej wersji instrukcji. Mimo, że wygenerowana instrukcja okazała się “wolniejsza”, różnica 10% wydaje się być pomijalna.

Konkluzja: Możliwa jest daleko idąca automatyzacja prac związanych z wytwarzaniem dokumentacji użytkownika pod warunkiem, że pracom *stricte* programistycznym towarzyszy dbałość o specyfikację wymagań i skrypty automatyzujące testy akceptacyjne.

Acknowledgments

This thesis could not have been written without the help of numerous people. I would like to take this opportunity to mention some of them.

First of all, I would like to extend my gratitude to Professor Jerzy Nawrocki. He guided me through the exciting world of research problems and showed that every challenge can be a beautiful adventure. Without his support and feedback I could have never conducted the research presented in this thesis.

Many thanks go to my colleagues, with whom I had the pleasure to collaborate: Magdalena Deckert, Sylwia Kopczynska, Jakub Jurkiewicz, Michał Maćkowiak, Mirosław Ochodek, Konrad Siek, Marcin Szajek, Bartosz Walter, Wojciech Wojciechowicz, Adam Wojciechowski. They were always eager to listen and help me with the problems I encountered.

Experiments are an important part of this thesis. I would like to thank students from Poznań University of Technology for their time. It was with their help that I was able to conduct many experimental evaluations.

I am greatful to the Authorities of the Institute of Computing Science at Poznań University of Technology for creating an environment in which I could develop my skills as a researcher and work with my colleagues.

I would like to thank the Polish National Science Center for supporting my research under the grant DEC-2011/03/N/ST6/03016.

Last but not least, I want to show my appreciation to my parents, sister, girlfriend and friends for their constant support and encouragement during my work on this thesis.

Bartosz Alchimowicz, Poznań, March 2015

List of journal papers by Bartosz Alchimowicz
(sorted by 5-year impact factor)

Symbols:
IF = 5-year impact factor according to Journals Citations Report (JCR) 2013
Ranking = ISI ranking for the Software Engineering category according to JCR 2013
Citations = citations according to Google Scholar, visited on 2015-02-23
Ministry = points assigned by Polish Ministry of Sci. and Higher Education as of Dec. 2014.

1. M. Ochodek, B. Alchimowicz, J. Jurkiewicz, J. Nawrocki: *Improving the reliability of transaction identification in use cases*, Information and Software Technology, 53(8), pp. 885–897, 2011. DOI: 10.1016/j.infsof.2011.02.004
IF: 1.583; Ranking: 24/105; Citations: 6; Ministry: 35.
2. B. Alchimowicz, J. Nawrocki: *The COCA quality model for user documentation*, Software Quality Journal, DOI: 10.1007/s11219-014-9252-4 (available on-line but not assigned to an issue yet)
IF: 0.889; Ranking: 68/105; Citations: 0; Ministry: 20.
3. B. Alchimowicz, J. Jurkiewicz, M. Ochodek, J. Nawrocki: *Building benchmarks for use cases*, Computing and Informatics, 29(1), pp. 27–44, 2010.
IF: 0.331; Ranking: 110/121[1]; Citations: 12; Ministry: 15
4. B. Alchimowicz, J. Nawrocki: *Generating Syntax Diagrams from Regular Expressions*, Foundations of Computing and Decision Sciences, 36(2), pp. 81–97, 2011.
IF: N/A; Ranking: N/A; Citations: 1; Ministry: 9
5. Ł.Olek, B. Alchimowicz, J. Nawrocki: *Acceptance Testing of Web Applications with Test Description Language*, Computer Science, 15(4), pp. 459–477, 2014, DOI: 10.7494/csci.2014.15.4.459
IF: N/A; Ranking: N/A; Citations: 0; Ministry: 8

List of conference papers by Bartosz Alchimowicz
(sorted by the *Ministry* points)

1. B. Alchimowicz, J. Jurkiewicz, M. Ochodek, J. Nawrocki: *Towards Use-Cases Benchmark*, Lecture Notes in Computer Science. vol. 4980, pp. 20–33, 2011.
Citations: 3; Ministry: 10
2. J. Nawrocki, M. Ochodek, J. Jurkiewicz, S. Kopczyńska, B. Alchimowicz: *Agile Requirements Engineering: A Research Perspective*, In: SOFSEM 2014: Theory and Practice of Computer Science, ed. by Geffert, Viliam and Preneel, Bart and Rovan, Branislav and Štuller, Július and Tjoa, AMin, vol. 8327, pp. 40–51, Springer. Lecture Notes in Computer Science. 2014.
Citations: 0; Ministry: 10
3. M. Ochodek, B. Alchimowicz, J. Jurkiewicz, J. Nawrocki: *Reliability of transaction identification in use cases*, Proceedings of the Workshop on Advances in Functional Size Measurement and Effort Estimation, pp. 51–58, 2010.
Citations: 0; Ministry: 0

[1] Category: Artificial Intelligence

Contents

List of Abbreviations

A

ATG	Automatic Text Generation

C

COCA	Quality model for user documentation; the name is an abbreviation from the following characteristics: Completeness, Operability, Correctness and Appearance

D

DSL	Domain-Specific Language

F

DET	Documentation Evaluation Test

G

GUI	Graphical User Interface

I

IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
IT	Information Technology

N

NFR	Non-functional Requirement
NLG	Natural Language Generation
NLP	Natural Language Processing
NLU	Natural Language Understanding
NoRT	Non-functional Requirements Template

TeCT	Technical Constraint Template

P

PDF	Portable Document Format
POS	Part of Speech

S

SE	Software Engineering
SRS	Software Requirment Specification

V

TDL	Test Description Language

U

UC	Use Case
UCDB	Use Cases Database
UM	User Manual
UML	Unified Modeling Language™

V

VDM	Vienna Development Method

Chapter 1

Introduction

1.1 Problems concerning user documentation in software projects

User documentation is one of the deliverables in software development projects. The aim of user documentation is to "describe, explain, or instruct how to use software" [67, 70]. This is accomplished by writing documentation for people who perform certain roles as users of an application (e.g. the user manual can contain chapters aimed at different end-users, like the chief accountant or the warehouse manager). This aim can also be realized by enriching an application with on-line explanations, e.g. the explanation of field's syntax in HTML5 forms.

The creation of a high quality documentation is an expensive and time consuming task [79, 145]. Jones states that, for each function point[1] (estimated during size estimation) it is necessary to write 0.425 of a page in a document [79], while guidelines by Sun Technical Publications describes that 3-5 hours are required to create one page of a user manual [145]. Unfortunately, common pressure to minimize costs marginalizes tasks other then code development. The situation is additionally hindered by low reuse of already written documentation. According to Jones, only 15% of the documentation can be reused, while code reuse has been estimated for about 36% (depending on the computer language) [79].

Consequently, the quality of user documentation is often low and readers are often dissatisfied [110, 111]. But irritation among readers is not the only issue; inaccurate user documentation can lead to financial loses. For example, when documentation is unavailable, or is of low quality, a company may be forced to organize training courses. Furthermore, users who cannot solve problems on their own may disturb

[1]A function point is a unit of functional size measurement in Function Point Analysis.

their co-workers, thus reducing the effectiveness of the company. The same problem may occur when an available feature is not explained. This can lead to unwanted side effects, like removing all records in a database.

Unexpected losses are not limited to customers. End-users often ask vendors to help them solve their problems, which results in higher support costs [98, 137]. Moreover, information about a feature in documentation which are not available in software, may be reported as a bug in an application. All of this can negatively affect the budget of a vendor, or the opinion about the software and the vendor.

The question arises, whether it is possible to improve the quality of user documentation in ways other than increasing its budget. One could consider automatic generation of documentation. There are some attempts to automatize some components of technical documentation, but most of them are dedicated to technical staff as the target audience [52, 101, 118], and other readers seem to be neglected. The question about evaluation criteria can be raised as well, since generation of documentation can be beneficial only if the resulting content meets readers' requirements. To ensure good quality, one can use evaluation methods proposed in literature, however most of them suffer from many issues. Some of them provide a number of low-level quality criteria, (e.g. ISO 2654*nm* series concerning user documentation) which make evaluation quite a time consuming task. Other approaches use non-orthogonal criteria (e.g. *eight measures of excellence* by Markel [98]), which makes it difficult to draw conclusions.

1.2 Aim and scope

To improve the quality of user documentation, it was decided to investigate the following research question:

QUESTION 1. Is it possible to automatically generate user documentation for web applications whose quality is similar to that of content created by a human?

The term user documentation is used to refer to many types of documents which are dedicated to many roles. It was decided to limit the research by defining the following assumption:

ASSUMPTION 1. *User documentation is evaluated from the end-users' point of view.*

It was decided that the focus should be on end-users, since they appear to be the largest group using web applications. These people often have a limited amount of IT knowledge or experience, therefore well written documentation could be very beneficial to them and their employers. From the end-users' standpoint, the user manual and field explanations appear to be the most important.

Contemporary Internet applications use forms as a way of transferring data from the end-user to the server. A form is a collection of fields—an end-user is expected to enter data (text) into this type of field. Some of the fields have non-trivial syntax which is specified by a programmer as a regular expression. Such syntax should be explained to the end-user (for instance, in HTML5 there is an attribute called "title", which contains text to be displayed to the end-user). A *field explanation* is a piece of information (narrative text or graphics) presented to an end-user to explain what is a syntactically correct input to a given field of a form.

ASSUMPTION 2. *User documentation comprises user manual and field explanations.*

A quality model is necessary in order to compare the quality of two user manuals, thus the following research questions are considered in the thesis:

QUESTION 2. Which evaluation criteria for user documentation should be taken into consideration?

QUESTION 3. What is the average quality of commercially available user manuals?

To automatically generate user documentation, one must know what is to be generated and which input data can be reused to save effort and time.

QUESTION 4. What kinds of software artifacts are widely available in software projects?

QUESTION 5. What is the expected content of user documentation and how to generate it on the basis of artifacts available in a project?

Moreover, linguistic issues are also taken into account.

ASSUMPTION 3. *The focus is on the English language, but the issue of multiple-language user documentation is also considered, where appropriate, with special attention paid to the Polish language.*

Many parts of this thesis are based on previously published papers or reports. Each chapter that is based on published material includes a structured abstract summarizing its main contribution.

The thesis is organized as follows: Chapter 2 presents basic information about user documentation. Chapter 3 describes the methods and tools of Natural Language Processing which were used when working on the thesis. Before beginning work on methods of automatic generation of user documentation, one needs to decide on evaluation criteria necessary to check whether the quality of generated documentation is comparable to that of handmade one. Those criteria compose a quality model which is discussed in Chapter 4. The most important part of this thesis are Chapters 5 and 6. The former aims at generating field explanations on the basis of regular expressions, and it proved quite a challange. The latter presents initial research concerning generation of a complete user manual on the basis of earlier produced artifacts, such as business case, use cases, and test cases. Chapter 7 summarizes the results of the thesis.

1.3 Conventions

The following typographical conventions are used in the thesis:

`code`	Program code, statements, variables, classes.
KEY TERM	Margin notes about key terms and concepts appearing in the corresponding paragraph.
[5]	Citations to books and articles, and references to other sources of information.

In the case of mathematical symbols and variables, their meaning is explained in the place where they are introduced.

A number of people participated in the research. To emphasise their contribution, the pronoun “we” is used in the thesis. However, despite the help and inspiration I have received, I am taking full responsibility for all research and results described in this thesis.

Chapter 2

Selected aspects of creating user documentation for web applications

This chapter addresses those aspects of creating user documentation which are not discussed in further parts of this thesis. The primary focus is on activities which should be taken into consideration by a vendor before (sections 2.1, 2.2, 2.6 and 2.7) and during the creation of user documentation for a web application (sections 2.3 and 2.4, 2.5).

2.1 Audience analysis

AUDIENCE ANALYSIS

Typically, there are several types of stakeholders that can play different roles within the life-cycle of a web application. Some of them install the application on a server, other may be responsible for its configuration and customization, and others still can use it to perform their daily activities. From the perspective of user documentation, it is important to recognize and characterize all of these roles. Such an analysis allows vendors to select the appropriate type of user documentation (see Section 2.2), the most suitable vocabulary (either a natural language or a controlled one, e.g., Simplified Technical English [27]), decide what to describe, how detailed the description should be, etc.

The following traits of prospective users may be taken into consideration while creating user documentation [61, 67, 145]:

- domain knowledge and experience,
- technical knowledge and experience,

- education level (including training and language level[1]),
- performed tasks and their frequency, and
- disabilities.

Some potential sources of such information are *Software Requirement Specification* and *Stakeholder Requirements Specification* [61, 72].

2.2 Basic types of user documentation

According to [67, 145], the most popular types of user documentation are as follows:

USER MANUAL

- *User manual*—explains how to use the software;
- *Installation manual*—describes how to install and configure software on servers and for clients;
- *Programmer manual*—explains how to extend the functionality of software, e.g., by creating plugins (if possible);
- *System administration manual*—informs how to maintain software and troubleshoot different kinds of problems;
- *Reference manual*—provides details concerning the software, such as supported functions, computer language syntax, etc.

Furthermore, a user documentation should contain explanations of syntax of fields in application forms [67].

User documentation can also be divided by its usage mode [67]:

- *instructional mode*—teaches how to perform tasks using the software;
- *reference mode*—provides details for users who are familiar with the software's functions.

FIELD EXPLANATIONS

ISO/IEC Std 26514:2008 points out also the importance of explanations of syntax of fields in forms, often called field explanations, for short.

In the context of web applications, an *instructional mode user manual* can be used by many classes of users, including IT-laymen, while other documents are more suited to advanced users (e.g. administrators and programmers).

[1]For example, according to *Common European Framework of Reference for Languages: Learning, Teaching, Assessment*, see [5]

Different organizations use various naming conventions to refer to user documentation, for example, some prefer the term *guide*, others *manual* or *documentation* [34, 145].

2.3 Style guides and standards

To improve the quality of user documentation, a number of guidelines were created (in some way, they are similar to code conventions used by programmers). The most popular style guides are:

STYLE GUIDES

- *Microsoft Manual of Style* by *Microsoft Corporation* (see [34])
- *The IBM Style Guide* by *International Business Machines Corporation* (see [37])
- *Read Me First! A Style Guide for the Computer Industry* by *Sun Technical Publication* (see [145])

In addition, there is a family of ISO/IEC/IEEE 265*nn* standards:

STANDARDS

- *ISO/IEC/IEEE Std 26511:2012 - Systems and software engineering – Requirements for managers of user documentation*—focuses on documentation management, such as planning and controlling methods (see [74])
- *ISO/IEC/IEEE Std 26512:2011 - Systems and software engineering – Requirements for acquirers and suppliers of user documentation*—describes agreements, requirements, constraints and other issues concerning user documentation which are important from the acquirer's and supplier's point of view (see [71])
- *ISO/IEC Std 26513:2009 / IEEE Std 26513-2010 - Systems and software engineering – Requirements for testers and reviewers of user documentation*—presents exemplary methods of evaluation (see [68])
- *ISO/IEC Std 26514:2008 / IEEE Std 26514-2010 - Systems and software engineering – Requirements for designers and developers of user documentation*—concentrates on activities carried out by a development team (see [67])
- *ISO/IEC/IEEE Std 26515:2012 - Systems and software engineering – Developing user documentation in an agile environment*—focuses on issues concerning creation of user documentation in agile environment (see [75])

Moreover, there is *IEEE Std 1063-2001 - IEEE Standard for Software User Documentation* (see [22]), which presents requirements for the content and format of user documentation. This standard is superseded by the ISO/IEC/IEEE 265*nm* series, but research still refers to it.

Presented guidelines and standards generally focus on two aspects:

- the process in which user documentation is created (including roles, management documents, etc.; this topic is outside the scope of this thesis) and
- the structure, content and format of user documentation (see Section 2.4).

Additionally, style guides for general writing can be taken advantage of, such as *The Chicago Manual of Style* and *The Elements of Style* (see [149] and [143] respectively). They are not as focused on user documentation as the literature presented above, but they can be a source of helpful advice.

2.4 General recommendations

The main goal is to create user documentation that can be easily understood by its readers, i.e. where language complexity (grammatical structures, vocabulary and technical terms), document organization, layout, and other elements do not create any difficulties for the target audience. Moreover, everything should follow the rules of a given language (grammar, punctuation, etc.) and be bias-free (i.e. without any sexual, political, or racial stereotypes) [34, 145].

Here are the most common recommendation [34, 37, 145]:

- Use present tense and active voice.
- Use second person (direct address).
- Write simple sentences with short and plain words.
- Use vocabulary consistently, especially technical terms.
- Use anthropomorphism only when necessary (e.g., when it is a convention in a given domain).
- Explain technical terms and abbreviations[2].
- Avoid ambiguous words.
- Avoid needless words (e.g., use *to* instead of *in order to*)
- Avoid foreign words (e.g., *vis-à-vis*, *ad hoc*).

[2] *Microsoft Manual of Style* recommends not to use Latin abbreviations [34].

- Avoid humor, jargon and slang.
- Avoid idioms and nonstandard colloquialism.

When it comes to technical terms, it is important to comply with the conventions of a given software platform (e.g. Windows, Linux) or environment (e.g. web-browser). For example, a widget in which users enter text can be called as a *Text Box* (using the class name `TextBox` from Google Web Toolkit) or *Input* (HTML). To avoid misunderstandings, it is important to find terminology that is the most appropriate for a given target audience.

Another problem concerns writing portable explanation (e.g., for a desktop and web application). In these cases, one can refer to a field by its label, not its type [34].

When it comes to usage of style guides it is important to use them consistently. It may happen that different styles propose to explain the same item using contradictory vocabulary. For example, Microsoft's style proposes to call `[[ ]]` as a *double brackets* [34], while Sun's style recommends phrase *double square brackets* and discourages the use of *double brackets* [145].

The structure of a user manual is discussed in Chapter 6.

2.5 Legal issues

User documentation should protect intellectual property of companies and other organizations [145]. The following elements should be taken into consideration when creating user documentation:

- copyright information (including third-party copyright information);
- trademark information (terms, usage, etc.);
- identification of confidential information.

2.6 Cost estimation

COST ESTIMATION

According to the style guide by Sun Technical Publications [145], 3-5 hours are required to write one page of a manual and 1-3 hours are needed to revise a previously created page. Editing takes 6-8 hours per page and indexing takes 20 minutes per page. Furthermore, 5% of the total time of all other activities is necessary for production preparations, and 10-15% is required for project management.

Jones states that for each function point estimated during software size estimation, 0.425 of a page in the user guide is required[3] [79]. This is an average value based

[3]Term *full users guides* is used in the book.

on four types of software: management information systems (0.2 page per function point), systems software[4] (0.4), military (0.8), and commercial (0.3).

2.7 Benefits of a good quality

The value of user documentation can be assessed within the following categories (among others) [145]:

- *increased benefits*—increasing vendor and user profit owing to user documentation, e.g., through higher sales or better productivity;
- *costs saved*—reducing vendor cost, e.g., through lower support and training costs.

For instance, Spencer presented a case in which the number of support calls was reduced from 641 to 59 over a 5-month period [137]. Taking into account that the cost of an average support call in 2008 was estimated at $32 [98], the time spent on improving user documentation quality is shown to be beneficial. Additionally, Redish demonstrated an estimation made by Cover, which shows that the cost of fixing a problem in user documentation is $123 during the *edit cycle*, but $3116 after a document has already been released (this estimation includes both customers' and vendor's cost; the paper was published in 1995) [124].

[4]Understood as *Software that controls a physical device* [79]

Chapter 3

Selected aspects of Natural Language Processing

In order to automatically generate user documentation one needs first to acquire information about the software system being documented. Unfortunately, most of the information processed and stored in software development projects is expressed using natural language. Therefore, in order to automatically process such loosely structured and often ambiguous information one needs to use natural language processing (NLP) tools—which include natural language understanding (NLU) and natural language generation (NLG) [81, 134].

In this thesis, the goal of NLU methods is to analyze input data so that it can be used to generate new content (Section 3.1), which can be further used to create descriptions using NLG techniques (Section 3.2).

3.1 Natural Language Understanding

To uncover the user's intentions, it is necessary to transform input text into an easy to analyze form. This section describes four stages in which it can be done. Each task focuses on one problem, discusses possible approaches and prepares data for the next stage.

Instead of creating a new tool, we decided to use existing NLU systems for different purposes (like Standford [2, 3], OpenNLP [1], and Natural Language Toolkit (NLTK) [121])

3.1.1 Segmentation

The goal of this stage is to divide a step into a list of words [121]. To do this, each step must be divided into sentences (it is recommended to write a step as one simple sentence [29], but, multiple sentences are also possible [12]); next each sentence needs to be divided into words. Both tasks can be done using segmentation (also known as tokenization) [49, 56, 121].

Segmentation into sentences can be done using a specified character as a boundary (e.g., a period). However, not all sentences end with a period, thus regular expressions (regexps) may support more options [6, 121]. Unfortunately, regexps can quickly become complicated and difficult to maintain [42]. A possible solution is using tokenizers which learn how to divide text on the basis of a training data set (i.e. trainable tokenizers). The training can be either supervised (e.g., TrTok [99]) or unsupervised (e.g., Punkt [87]).

For example, step *Selecting committee changes the status of the application to rejected. Use Case finishes.* (exemplary step from *UCDB: Quantitative Referential Specification Full version 2.0F* [12]) can be divided into the following sentences:

- *Selecting committee changes the status of the application to rejected.*
- *Use Case finishes.*

Next, each sentence is divided into words [4]. The simplest approach is to split a sentence at the point of a space character (unsegmented languages, like Chinese, are beyond the scope of this thesis). However, there is an issue with the punctuation within a sentence, i.e., whether a given punctuation mark should be assigned to the preceding or subsequent word, or perhaps left alone. It is important to divide sentences in a way supported by a tagger. For example, the sentence *No, he didn’t.* can be divided in the following ways (using word tokenizers available in NLTK [121]):

- *Treebank* tokenizer – *"No" "," "he" "did" "n’t" "."*
- *Punkt* tokenizer – *"No" "," "he" "didn" "’t."*
- *Punct* tokenizer – *"No" "," "he" "didn" "’" "t" "."*

Multi-part words and multi-word expressions are another concern. For instance, some words are constructed from other words by joining them with a hyphen, e.g. *event-driven.* To detect words of this kind, a variant of *maximum matching algorithm* can be used [36]. This method uses a list of words (for a given language) and, while analyzing text, it tries to find the longest word in that list.

There are two common outputs from this task: a list of words and the text itself, with a list of pairs of integers (each pair marks the beginning and the end of one word).

3.1.2 Part-of-speech tagging

This phase assigns parts-of-speech (POS) to words. Here is a small example (a period is not a POS, but can be tagged as well):

Sentence:	Actor	provides	data	.
POS:	noun	verb	noun	period

The name of POS may vary between taggers. For example, a noun can be called a *noun*, *NN* or any other way.

The main problem with this task is connected with the ambiguity of words. For example, words such as *book*, *break*, *cut*, *display*, *say* (and many more) can be used both as nouns and as verbs. DeRose states that over 40% of English words listed in *Brown University Standard Corpus of Present-Day American English*[1] have more than one tag [38, 46].

There are two common approaches to tagging: rule-based and stochastic [81, 121]. The first approach tags words according to certain rules; for instance, one can assume that all words which end with *ing* are verbs, all words which end with *ment* are nouns, etc. A more advanced rule-based approach uses affix tagging [121]. In this strategy, a tagger learns how to tag words on the basis of n-characters from words' affix (prefix or suffix) available in a training set.

Stochastic tagging, based on the Hidden Markov Model, computes the probability of a word with a certain tag on the basis of training corpus [81, 82]. For example, after the determiner *the* a noun will occur with a certain probability, adjective with another, etc.

An example of a supervised learning rule-based tagger is the Brill tagger [25] and an example of a stochastic tagger is Trigrams'n'Tags proposed by Brants [24].

3.1.3 Lemmatization

Lemmatization makes it possible to find the base form of a word, called a *lemma*. For example, the lemma of the word *provides* is *provide*.

[1]Corpus is a collection of selected documents with tagged parts of speech

This task is often done using *WordNet*, an online lexical database for the English language[2] [106]. Since some words are ambiguous, it is a good practice to use POS assigned by a tagger while searching for lemma.

There is also a Polish version of *WordNet* called *plWordNet* [39].

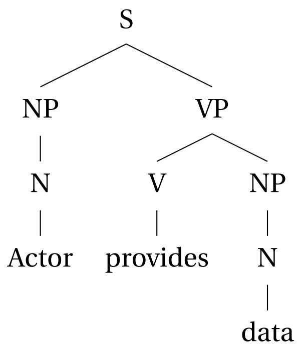
3.1.4 Parsing

This is the phase in which a syntactic structure is defined [81]. A syntactic structure is often represented by a parse tree, similarly to computer languages [8].

For example, using the following rules (where `N` stands for a noun, `V` for a verb, `S` stands for a sentence, `VP` for a verb phrase and `NP` for a noun phrase):

```
N -> actor | data
V -> provide
S -> NP VP
S -> VP
VP -> V NP
NP -> N
```

step *Actor provides data* can be represented by the following parse tree:



According to Jurafsky and Martin [81], the most popular strategies to construct a parse tree are top-down (also known as *goal-directed search*) and bottom-up (also known as *data-directed search*). Both strategies apply the grammatical rules of a language. The difference between them is in how the parse tree is constructed. In the top-down approach, a parse tree is build from node *S* down to the leaves. That is to say, a parser takes all starting points and adds leaves to it (on the basis of available grammar rules) until POS is reached. Trees that do not match all POS are rejected. Trees in the bottom-up approach are constructed from the bottom, i.e. staring from POS. The tree expands by adding grammar rules that match leaves. A tree to which no rule can be applied is rejected.

[2]The tool can be tested online at `http://wordnetweb.princeton.edu/perl/webwn`.

Due to the ambiguity of natural languages, it is possible to get more than one parse tree. In that case *Probabilistic Context-Free Grammar* can be used (also known as *Stochastic Context-Free Grammar*) [23, 131]. This approach adds the probability of occurrence to every rule. When two or more parse trees are created, the probability of each tree is computed, and the most probable is selected.

An example of a top-down parser is the Earley algorithm [41]. Its probabilistic version was presented by Stolcke [142]. An example of a bottom-up parser is the Cocke-Younger-Kasami algorithm [81, 85], of which the probabilistic version was presented by Ney [109].

3.2 Automatic Text Generation

There are two common approaches to converting input data to text: template-based and natural language generation (NLG) [81, 125, 126, 150]. Reiter proposed automatic text generation (ATG) as a term referring to both [125].

In this thesis we use the template-based approach.

3.2.1 Input data

According to Reiter and Dale, input data for ATG systems can be divided into the following categories [126]:

- *Knowledge source*—or simply a database. A source of raw information which can be used to generate content. For example, Chapter 5 presents a system which requires the name of a web form's entry box along with a regular expression (used for string validation) as input.

- *User model*—characteristics of the reader (knowledge, experience, education, etc.). This information can be used to adjust text to the audience. For example, the explanation of a regular expression provided to programmers may be different from the one given to IT-laymen (see Section 2.1).

- *Communicative goal*—defines what to do with available data, whether to explain, summarize, etc. For example, an explanation of metacharacters used in a regular expression may be expected in one case, but another may demand an in-depth explanation of how regexp works.

- *Discourse history*—elements that had been presented to a reader in the past. This can prevent a generator from describing items already known or previously explained.

When an ATG system is designed to create one type of explanation for a specified audience only, it is not necessary to provide *User model* and *Communicative goal.* If text optimization is not required, *Discourse history* can be omitted as well.

3.2.2 Template-based approach

Generally, in a template-based approach there is a data structure (a template) with gaps which need to be filled in. [104, 125, 126]. It is possible to insert either text (e.g., from input data) or another template (depending on how the template has been written) inside each gap. In the end, all gaps are filled in with available data and the text is ready to be presented.

For example, an online shopping application can show the number of items in a cart with the following code:

```
switch (n) {
    case 0: printf("There are no items in your cart"); break;
    case 1: printf("There is 1 item in your cart"); break;
    default: printf("There are %d items in your cart", n);
}
```

An appropriate template is selected on the basis of variable n. In the case of $n > 1$ a gap needs to be filled in, i.e. %d has to be replaced by a number of items in the cart.

Another example can be seen in mail-merge applications (e.g., mail-merge in OpenOffice or MS Word) [126]. An e-mail template (with gaps) is prepared by a human user. When messages are generated, all gaps are filled out with information from a database. More advanced tools can customize text on the basis of available data, e.g., there can be a version for women and men, adults and children, etc.

A more sophisticated strategy is used in YAG [103, 104]. This tool is designed to generate single sentences, and supports embedding templates. A template is made up of its name, default slot values (attributes) and rules for text creation. In order to generate a sentence, one needs to provide data using a feature structure[3] (which includes name of a template). The generation starts with setting additional attributes on the basis of information in templates. Next, rules for text creation are executed (rules for embedded templates are run first). Finally, top-level template rules generate the output.

YAG also supports knowledge representations described in the Semantic Network Processing System, which can be used to input data understood by YAG.

[3]A feature structure is a set of attribute-value pairs.

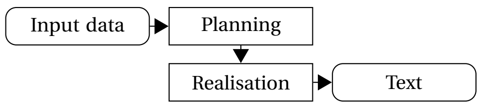


Figure 3.1: Simplified architecture of a text generation system

3.2.3 Natural Language Generation

Systems of this kind use linguistic knowledge and artificial intelligence [81, 125, 126, 150]. According to Reiter, this approach allows the creation of descriptions of higher quality than those generated by template-based systems [125]. However, not everyone agrees with this opinion (see [150]).

Reiter and Dale, as well as Jurafsky and Martin, provide an exemplary architecture of a NLG system [81, 126]. A simplified version is presented in Figure 3.1. Initially there is only raw input (which can be similar to the one used in the template-based approach). This input (in a data structure) is analyzed and the content of an expected description is planned. It may then be decided what kind of information to include (this is called *Content determination*) and how to represent it (e.g., what to put in each paragraph and what phrases to use). When everything is planned out, text is generated by a *realization* module.

Detailed planning of a long description can be problematic. To simplify this task, one can begin with making a *draft* with general decisions (e.g. what kind of data to include and where to put it) and postpone all detailed arrangement (e.g. how to represent particular data). This is called *Microplanning* and it can include the following tasks [126]:

- *lexicalization*—selection of words and phrases;
- *aggregation*—merge/division of elements into one/many groups (e.g. *John has a cat. Cat is always hungry* can be also presented as *John has a cat which is always hungry*);
- *generation of referring expressions*—selection of words indicating the same entity (e.g. *John, he, suppliant,* etc.)

It is up to the designer of a NLG system what phases to use and how to divide the system into modules.

3.2.4 Evaluation of language generation tools

According to Mellish and Dale, the evaluation of NLG systems can be carried out with the following goals in mind [105]:

- *Evaluation of a theory*—Text generators can use different approaches to natural text organization, called *theories* (e.g. Rhetorical Structure Theory). One may evaluate the degree to which a given theory is appropriate for a particular task.
- *Evaluation of a system*—Determine the performance of a system, e.g., measure the speed of generation or grammatical correctness of created text.
- *Evaluation of applicability*—Determine which solution is more suitable for a given problem, e.g., whether a more useful weather forecast is generated by a NLG system or a template-based approach.

The presented goals are dedicated to NLG systems, but due to their very general nature it should be possible to use them in assessing template-based systems as well.

The aforementioned goals can be evaluated in a number of ways. The most common strategies seem to be human-subject evaluation (with, e.g., prospective users, tool authors, experts, etc.) and comparison with existing texts (e.g., corpus) [81, 105].

In human-subject evaluation, participants are frequently asked to read a generated text and rate it, using various criteria [28, 95]. For example, an evaluation carried out by Coch used *correct spelling*, *comprehensiveness* and several other criteria [28]. Unfortunately, no standard or convention of criteria is known to us.

Domain experts are a unique type of participant [51, 95, 139]. The assessment can be carried out similarly to the evaluation by non-experts described above (i.e. read and rate), or by comparing texts created by a system and written by experts [138]. Another approach is to analyze modifications made by experts to a generated text [107].

Another variant of evaluation was proposed by Hardcastle and Scott [51]. They described a Turing-like test, in which human- and computer-created variants are shown to participants, who are then asked to determine whether the author of text is a human or a computer program; each decision needs to be justified.

Human-subject evaluation can also be used to measure the efficiency of achieving goals. It is, for example, possible to test the speed of reading or measure task performance [153, 155]. In the case of performance evaluation, there are frequently two or more groups, which perform the same tasks, but according to a different description [155]. Comparing completion time or the number of accomplished tasks makes it possible to determine which explanation is superior.

Experiments with human-subjects can be time consuming. Langkilde proposed an automatic, corpus-based evaluation [92]. Generator input is created on the basis of annotations in the corpus; text is then created, and compared with the original sentence. This approach is quite popular (see e.g. Bangalore et al. [15], and Marciniak and Strube [97]), however, it is questioned as well. Reiter and Sripada state that it is possible to generate text which is different from the original, but still meet the established quality requirements [127].

Another means of evaluating ATG systems is the use of metrics from the Machine Translation (MT) field. Bilingual Evaluation Understudy (BLEU) metric counts the proportion of n words placed next to each other (called n-grams) that are shared by the generated text and reference translation (a reference translation can be a text written by a professional translator; there can be more than one reference text) [117]. According to Belz and Reiter, 4-grams (4-words) is the common size for evaluation [18]. Unfortunately, BLEU have problems with n-grams that are more informative, i.e. those that are used less frequently [40, 130]. To solve this issue,the NIST metric[4] can be used [40].

[4]The name comes from the *National Institute of Standards and Technology.*

Chapter 4

The COCA quality model for user documentation

Preface

This chapter contains the paper: *Bartosz Alchimowicz and Jerzy Nawrocki:* The COCA quality model for user documentation, *Software Quality Journal, DOI: 10.1007/s11219-014-9252-4 (available on-line since 12 October 2014, but not assigned to an issue yet).*

My contribution to this paper included the following tasks: *1)* co-elaboration of COCA quality model and Documentation Evaluation Test; *2)* co-elaboration of evaluation methods; *3)* conducting experiments and case studies; *4)* creation of the COCA and the DET quality profiles.

Context: Evaluation criteria are needed to compare the quality of a generated user manual with the quality of a corresponding man-made documentation. There are some proposals in the literature (e.g. ISO Standards, Markel's measures of excellence, etc.), but they have many drawbacks, e.g., some of them are time consuming, other lack orthogonality, and still others are superficial.

Objective: The goal of this chapter is to propose a quality model for user manuals. The model has to be orthogonal, complete and equipped with evaluation methods.

Method: Creation of a quality profile started with a literature review, including analysis of existing quality models and standards. Proposed quality model has been compared with other models (which directly or indirectly concern user manual), and checked while evaluating 9 commercial user manuals.

Results: The COCA quality model is presented. It comprises of four orthogonal quality characteristics: Completeness, Operability, Correctness, and Appearance. Moreover, two acceptance methods are introduced: pure review (based on ISO Std. 1028:2008), and Documentation Evaluation Test (based on Browser Evaluation Test).

Two exemplary quality profiles have been proposed on the basis of collected data: for the COCA quality model and for the Documentation Evaluation Test.
Conclusion: It is possible to design a complete and an orthogonal quality model for user manuals which allows to evaluate a user manual from standpoint of end-users and can be used to compare quality of documents.

4.1 Introduction

A good quality user manual can be beneficial for both vendors and users. According to Fisher [44], a project can be called successful if its *software performs as intended and the users are satisfied.* From the point of view of end-users, the intended behaviour of a software system is described in the user manual. Thus, a defective user manual (e.g. lack of consistency with the software system) has an effect similar to defective software (off specification) – both will lead to user irritation, which will decrease user satisfaction. Pedraz-Delhaes et al. [120] also point out that users evaluate both the product and the vendor on the basis of provided documentation. According to the data presented by Spencer [137], a good quality user manual can reduce the number of calls from 641 to 59 over a 5-month period (in 2008 the average cost of support for one call was above $32 [98]).

Unfortunately, end-users are too frequently dissatisfied with the quality of their user manuals. They complain that the language is too hard to understand, the descriptions are boring, the included information is outdated and useless [110, 111]. Some users even feel frustrated while working with the software [55].

So, a good quality user manual is important. Thus, the question arises of what *good quality* means in this context, i.e. what quality characteristics should be considered when evaluating the quality of a user manual. A set of quality characteristics constitutes a quality model [65] and these should be *orthogonal* (i.e. there should be no overlap between any two characteristics) and *complete* (i.e. all the quality aspects important from a given point of view should be covered by those characteristics).

In this paper, an orthogonal and complete quality model for user documentation is presented. The model is called COCA and consists of four quality characteristics: Completeness, Operability, Correctness and Appearance. From the practical point of view, what matters is not only quality characteristics, but also the way they are used in the evaluation process. As indicated by the requirements of Level 4 of Documentation Maturity Model [59], quality characteristics should allow quantitative assessment. In this paper, two approaches are discussed, a review-based evaluation and an empirical one. Both of them provide quantitative data. For each of them, quality profiles for the educational domain are presented which can be used when interpreting evaluation data obtained for a particular user documentation.

The paper is organized as follows: In Section 4.2, a set of design assumptions for the proposed quality model is presented. Section 4.3 contains the COCA quality model. Section 4.4 shows how the proposed model can be used. Section 4.5 presents an empirical approach to operability assessment. Related work is discussed in Section 4.6. A summary of the findings and conclusions are contained in Section 4.7.

4.2 Design assumptions for the quality model

As defined by ISO Std. 25000:2005 [65], a quality model is a *set of characteristics, and of relationships between them, which provides a framework for specifying quality requirements and evaluating quality.*

The quality model described in this paper is oriented towards user documentation, understood as *documentation for users of a system, including a system description and procedures for using the system to obtain desired results* [70].

The design assumptions for the quality model are presented in the subsequent parts of this section.

4.2.1 Form of user documentation

User documentation can have different forms. It can be a PDF-like file ready to print, a printed book, on-screen information or standalone online help [71].

ASSUMPTION 1. *It is assumed that user documentation is presented in the form of a static PDF-like file.*

JUSTIFICATION. On-screen help is based on special software and to assess its quality one would have to take into account the quality characteristics appropriate for the software, such as those presented in one of the ISO standards [69]. That would complicate the quality model and the aspects which are really important for user documentation would be embedded into many other characteristics. Thus, for the sake of clarity, such forms of user documentation as on-screen help are out of the scope of the presented model. To be more precise, on-screen help can be evaluated on the basis of the proposed model, but to have a complete picture one should also evaluate it from the software point of view. □

4.2.2 Point of view

The quality of user documentation can be assessed from different points of view. Standards concerning user documentation presented by ISO describe a number of roles that are involved in the production and usage of user documentation (e.g. suppliers [71], testers and reviewers [68], designers and developers [67], and users for whom such documentation is created).

ASSUMPTION 2. *It is assumed that user documentation is assessed from the end-users' point of view.*

JUSTIFICATION. People may have different requirements for user documentation and thus they focus on different aspects, i.e. project managers may want to have documentation on time while designers may be interested in creating a pleasing layout. However, all work that is done aims to provide user documentation that is satisfactory for end-users. Thus, their perspective seems to be the most important. As a consequence, legal aspects, conformance with documentation design plans, etc. are neglected in the proposed model. □

4.2.3 External quality and quality-in-use

The software quality model presented in ISO/IEC Std. 9126:1991 was threefold: the internal quality model, the external quality model, the quality-in-use model. From the users' point of view, internal quality seems negligible and as such is omitted in this paper. We are also not taking into account the relationship between user documentation and other actors, such as the documentation writer. Considering the above, the following assumption seems justified:

ASSUMPTION 3. *A quality model for user documentation can be restricted to characteristics concerning external quality and quality-in-use.*

4.2.4 Context of use

There are many possible contexts of use for user documentation. One could expect that such documentation would explain scientific bases of given software or compare the software against its competitors. Although this information can be valuable in some contexts, it seems that text books or papers in professional journals would be more appropriate for this type of information. Thus, the following assumption has been made when working on the proposed quality model:

ASSUMPTION 4. *User documentation is intended to support users in performing business tasks.*

4.2.5 Orthogonality of a quality model

DEFINITION 1. *A quality model is orthogonal, if for each pair of characteristics C_1, C_2 belonging to it, there are objects O_1, O_2 which are subject to evaluation such that O_1 gets a highly positive score with C_1 and a highly negative score with C_2, and for O_2 it is the opposite.* □

ASSUMPTION 5. *A good quality model for user documentation should be orthogonal.*

JUSTIFICATION. If a quality model is not orthogonal, then it is quite possible that some of its characteristics are superfluous, as what they show (i.e. the information they bring) can be derived from the other characteristics. For instance, when considering the subcharacteristics of ISO Std. 9126 [64] one may doubt whether changeability and stability are orthogonal, as one strongly correlates with the other (see [80]). □

4.2.6 Completeness of a quality model

The completeness of a quality model should be considered in the context of the point of view of a stakeholder. This point of view can be characterized with the set of *quality aspects* one is interested in. A *quality aspect* is a type of detailed information about quality. Using terminology from ISO Std. 9126 and ISO Std. 25010 [69], a *quality aspect* could be a quality subcharacteristic, sub-subcharacteristic etc. An example of a *quality aspect* could be completeness of documentation from the legal point of view (that could be important from a company standpoint) or the presence of a table of contents. Many *quality aspects* can be found in standards such as ISO Std. 26513 and ISO Std. 26514 [67, 68].

DEFINITION 2. *A quality model is complete from a given point of view, if every quality aspect important from that point of view can be clearly assigned to one of the quality characteristics belonging to the quality model.* □

ASSUMPTION 6. *A good quality model for user documentation should be complete from the end-user point of view.*

The above assumption follows from Assumption 2.

4.3 The COCA quality model

The COCA quality model presents the end-users' point of view on the quality of user documentation. As its name suggests, it consists of four quality characteristics: **C**ompleteness, **O**perability, **C**orrectness, and **A**ppearance. Those characteristics are defined below.

COCA QUALITY MODEL

DEFINITION 3. *Completeness is the degree to which user documentation provides all the information needed by end-users to use the described software.* □

COMPLETENESS

DEFINITION 4. *Operability sensu stricto (Operability for short) is the degree to which user documentation has attributes that make it easy to use and helpful when acquiring information that is contained in the user documentation.* □

OPERABILITY

JUSTIFICATION. There are two possible definitions of Operability: *sensu stricto* and *sensu largo*. Operability *sensu largo* could be defined as follows:

> *Operability sensu largo is the degree to which user documentation has attributes that make it easy to use and helpful when operating the software documented by it.*

Operability *sensu largo* depends on two other criteria: Completeness and Correctness. If some information is missing from a given user manual or it is incorrect then the helpfulness of that user manual is diminished when operating the software. Operability *sensu largo* is not a characteristic of a user manual itself, but is also depends on (the version of) the software. For instance, Operability *sensu largo* of a user manual can be high for one version of software, and low for another, newer version, if that new version of software was substantially extended with new features. Thus, Operability *sensu largo* is not orthogonal with Completeness and Correctness. Operability *sensu stricto* is defined in such a way that it is independent of Completeness or Correctness of the user manual. It depends only on the way in which a user manual is made up and how it is organized. To preserve orthogonality of the proposed quality model, Operability *sensu stricto* has been chosen over Operability *sensu largo*. □

DEFINITION 5. *Correctness is the degree to which the descriptions provided by the user documentation are correct.* □

CORRECTNESS

DEFINITION 6. *Appearance is the degree to which information contained in user documentation is presented in an aesthetic way.* □

APPEARANCE

As mentioned earlier, it is expected that the COCA quality model is both orthogonal and complete. These issues are discussed below.

CLAIM 1. *The COCA quality model is considered orthogonal.*

JUSTIFICATION. Since the COCA quality model consists of four characteristics, one has to consider 6 pairs of them. All of the pairs are examined below, and, for each of them, two manuals which would lead to opposing evaluations are described.

Completeness vs Operability

When a user manual contains all the information a user needs to operate a given software, but the user manual is thick and ill-designed (no index, exceedingly brief table of contents, all text formatted with a single font type without underlining etc.), then such a user manual would be highly complete, but its operability would be low. And vice versa: a user manual can be highly operable (i.e. its Operability *sensu stricto* can be high) but still be missing a lot of important information, causing its completeness to be low. That shows that Completeness and Operability are orthogonal.

Completeness vs Correctness

It is possible that a user manual covers all the aspects concerning usage of a given software, but the screen shots still refer to the old version of the software. Similarly, business logic described in the user manual may be based on outdated law regulations etc., which meanwhile have been changed in both the real world and in the software, but not in the user manual. And the contrary is also possible: all the descriptions provided by a user manual can be correct, but some important information can be missing (e.g. about new features added to the software recently). Thus, Completeness and Correctness are orthogonal.

Completeness vs Appearance

It is pretty obvious that a document can be highly complete, as far as information is concerned, but far from giving an impression of beauty, a good taste, etc.; and vice versa. Therefore, Completeness and Appearance are orthogonal.

Operability vs Correctness

According to Definition 4, Operability is the degree of ease of finding information contained in the user manual. It does not take into account whether or not that information is correct. Because of this, Operability and Correctness are orthogonal.

Operability vs Appearance

According to Definition 6, Appearance is about aesthetics. According to the Free Dictionary[1], aesthetics is about *beauty or good taste.* Here are several examples of factors that can impact the aesthetics of a user manual:

- the chosen set of font types (many different font types can increase Operability, but decrease aesthetics; small font types can increase aesthetics but decrease Operability);

[1]http://www.thefreedictionary.com/aesthetic

- the set of colors used in the document (red and green can increase Operability but, if used improperly, can decrease the aesthetic value of a user manual);
- screenshots (they can be very valuable from the Operability point of view, but – if not properly placed – can decrease the aesthetics of a user document);
- decorative background (though favoured by some, it can decrease the readability of a document, thus it can decrease its Operability).

These factors can create a trade-off between the aesthetics and Operability of a user manual, thus Operability and Appearance can be regarded as orthogonal.
Correctness vs Appearance

It seems pretty clear that those two characteristics are orthogonal; a document can be highly correct but its Appearance can be low, and vice versa. □

CLAIM 2. *The COCA quality model is considered complete.*

JUSTIFICATION. To check completeness of the COCA model, the model will be examined from the point of view of the following sets of quality characteristics: ISO Std. 26513 and ISO Std. 26514 [67, 68], Markel's measures of excellence [98], Allwood's characteristics [14], Ortega's systemic model [116], and Steidl's quality characteristics for comments in code [141].

If talking about completeness, it is important to distinguish between two notions:

- documentation-wide *quality aspects*: all of them should be covered by a quality model if that model is to be considered complete;
- documentation *themes*: all of them should be covered by a user manual if that manual is to be considered complete.

Here are the documentation *themes* identified on the basis of ISO Std. 26513 and ISO Std. 26514:

- description of warnings and cautions,
- information about the product from the point of view of appropriateness recognizability,
- information on how to use the documentation,
- description of functionality,
- information about installation (or getting started).

If one of those *themes* is missing, the documentation can be incomplete in the eye of an end user. Thus, documentation *themes* influence *Completeness* of a user manual, but do not directly contribute to a quality model.

The *quality aspects* that can be found in ISO Std. 26513 and ISO Std. 26514 are listed in Table 4.1. They can be mapped into the three COCA characteristics: *Operability* (covers ease of understanding and consistency of terminology), *Correctness* (it corresponds to consistency with the product), and *Appearance* (it is influenced by consistency with style guidelines, editorial consistency, and cultural requirements). Thus, from the point of ISO Std. 26513 and ISO Std. 26514 the COCA model seems complete.

Completeness of the COCA quality model can be also examined against Markel's model of quality of technical communication [98]. Merkel's model is based on eight measures of excellence. Seven of them are presented in Table 4.2 and they are covered by the COCA characteristics. The eighth measure of excellence is *honesty*. It does not fit any of the COCA characteristics. However, it is not an external quality nor a quality-in-use characteristic, so – according to Assumption 3 – it is out of scope of the defined interest. Thus, the COCA model, when compared against Markel's measures of excellence, is considered complete.

Another set of quality characteristics has been presented by Allwood [14]. Two of them, i.e. *comprehensibility* and *readability*, are covered by COCA's *Operability* (if a document lacks *comprehensiveness* or *readability* then acquiring information from it is difficult, so COCA's *Operability* will be low). The third Allwood's characteristic is *usability*. It is a very general characteristic, which is influenced by both *comprehensibility* and *readability*. When comparing it with the COCA characteristics, one can find that *usability* encompasses COCA's *Completeness*, *Operability*, and *Correctness*, i.e. Allwood's *usability* can be regarded as a triplet of COCA's characteristics. Allwood also mentioned two other quality characteristics: *interesting* and *stimulating*. As we are interested in user documentation as support in performing business tasks (see

Table 4.1: Documentation-wide *quality aspects* vs COCA characteristics

Quality aspect (ISO Std. 26513 and ISO Std. 26514)	**COCA characteristics**
ease of understanding consistency of terminology	Operability
consistency with the product	Correctness
consistency with style guidelines editorial consistency cultural requirements	Appearance

Table 4.2: Markel's measures of excellence[98] vs COCA characteristics

Markel's measures of excellence	**COCA characteristics**
Comprehensiveness *A good technical document provides all the information readers need.*	Completeness
Clarity *Your goal is to produce a document that conveys a single meaning the reader can understand easily.*	Operability
Accessibility *readers should not be forced to flip through the pages . . . to find the appropriate section*	
Conciseness *A document must be concise enough to be useful to a busy reader.*	
Accuracy *a major inaccuracy can be dangerous and expensive*	Correctness
Professional appearance *document looks neat and professional.*	Appearance
Correctness *A correct document is one that adheres to the conventions of grammar, punctuation, spelling, mechanics, and usage.*	

Assumption 6), those characteristics can be neglected. Thus, one can assume that the COCA model is complete in its context of use.

Other quality characteristics the COCA model can be examined against are Ortega's characteristics [116]. Although those characteristics are oriented towards software products, they can be translated into the needs of user documentation, see Table 4.3. For instance, *learnability*, in the context of user documentation, can be understood as the degree to which it is easy to learn how to use a given user documentation. So, *learnability* is part of COCA's *Operability*. Similar meaning can be given to *self-descriptiveness* in the context of user documentation. Ortega's *understandability* also fits COCA's *Operability*, as it supports acquiring information from documentation. *Consistency* of software can be translated into consistency of user documentation with its software, so it is COCA's *Correctness*. *Attractiveness* of user documentation and its appearance are synonyms. Thus, all those characteristics are covered by COCA's characteristics. What is left outside is *effectiveness* (i.e. the capacity of producing a desired result), and a requirement for software to be *specified* and *documented*. All those three characteristics have no meaning when translated into quality of user documentation perceived from the point of view of the end-user.

The last set of quality characteristics is Steidl's quality model for comments in code [141]. Steidl's *coherence* (*how comment and code relate to each other*) maps onto COCA's *Correctness* (how user documentation and code relate to each other). Steidl's *completeness* and COCA's *Completeness* are also very similar as they refer to the completeness of information they convey. The remaining two Steidl's characteristics are *usefulness* (the degree of *contributing to system understanding*) and *consistency* (is the language of the comments the same, are the file headers structured the same way etc.). When translating them into the needs of user documentation readers, they map onto COCA's *Operability* (if user documentation did not contribute to understanding how to use the software, or the language of each chapter was different, *Operability* of such documentation would be low). Thus, the COCA model is also complete from the point of view of Steidl's characteristics. □

4.4 Review-based evaluation of user documentation

One of the aspects concerning software development is to decide whether a product is ready for delivery or not. A typical activity performed here is acceptance testing. However, this issue concerns not only software, but also user documentation. A counterpart of acceptance testing, when talking about user documentation, is quality evaluation of documentation for the purpose of acceptance. That assessment can be performed taking into account the COCA characteristics and is described below. Another application of the COCA quality model is selection. This kind of evaluation is used to compare two user manuals concerning the same system. The comparison can be performed for a number of purposes, e.g. to decide which method of creation is better (manual writing vs. computer aided) or to select a writer who provides a more understandable description for an audience.

Table 4.3: Ortega's quality characteristics[116] vs COCA characteristics.

Ortega's characteristics	**COCA characteristics**
Completeness	Completeness
Learnability Self-descriptiveness Understandability	Operability
Consistency	Correctness
Attractiveness	Appearance

4.4.1 Goal-Question-Metric approach to evaluation of user documentation

Quality evaluation is a kind of measurement. A widely accepted approach to defining a measurement is Goal-Question-Metric [151] (GQM for short). It will be used here to describe quality evaluation when using the COCA quality model.

Goal

The measurement goal of quality evaluation of user documentation can be defined in the following way:

> *Analyze the user documentation for the purpose of its acceptance with respect to Completeness, Operability, Correctness, and Appearance, from the point of view of the end-user in the context of a given software system.*

Questions

Each of the COCA characteristics can be assigned a number of questions which refine the measurement goal. Those questions should cover the quality aspects and documentation themes one is interested in (see justification to Claim 2). Table 4.4 presents the questions that, from our point of view, are the most important. We hope that they will also prove important in many other settings. Obviously, one can adapt those questions to one's needs.

At first glance it may appear that the question assigned to Operability is too wide when compared with the definition of Operability (Definition 4), as the definition excludes the completeness and correctness problems. That exclusion is not necessary when the evaluation procedure first checks Completeness and Correctness, and initiates Operability evaluation only when those checks are successful (see Figure 4.1).

Metrics

When evaluating user documentation, two types of quality indicators, also called metrics, can be used: *subjective* and *objective*.

Subjective quality indicators provide information on what people think or feel about the quality of a given documentation. Usually, they are formed as a question with a 5-grade Likert scale. Taking into account the questions in Table 4.4 (*To what extent...*), the scale could be as follows: *Not at all* (*N* for short), *Weak* (*w*), *Hard to say* (*?*), *Good enough* (*g*), *Very good* (*VG*). The results of polling can be presented as a vector of 5 integers [$\#N$, $\#w$, $\#?$, $\#g$, $\#VG$], where $\#x$ denotes the number of

Table 4.4: Questions assigned to the COCA characteristics

Question
Completeness
• To what extent does the user documentation cover all the functionality provided by the system with the needed level of detail? • To what extent does the user documentation provide information which is helpful in deciding whether the system is appropriate for the needs of prospective users? • To what extent does the user documentation contain information about how to use it with effectiveness and efficiency?
Operability
• To what extent is the user documentation easy to use and helpful when operating the system documented by it?
Correctness
• To what extent does the user documentation provide correct descriptions with the needed degree of precision?
Appearance
• To what extent is the information contained in the user documentation presented in an aesthetic way?

responses with answer x. For example, vector [0, 1, 2, 3, 4,] means that no one gave the answer *Not at all*, 1 participant gave the answer *Weak*, etc. (this resembles the quality spectrum mentioned by Kaiya et. al. [84]). These kinds of vectors can be normalized to the relative form, which presents the results as a percentage of the total number of votes. For example, the mentioned vector can be transformed to the following relative form [0%, 10%, 20%, 30%, 40%]. This form of representation should be accompanied by the total number of votes that would allow one to return to the original vector.

Objective quality indicators are usually the result of an evaluation experiment and they strongly depend on the design of the experiment. For instance, one could evaluate the Operability of user documentation by preparing a test for subjects participating in the evaluation, asking the subjects to take an open-book examination (i.e. having access to the documentation), and measuring the percentage of correct answers or time used by the subjects.

Interpretation

The fourth element of GQM is interpretation of measurement results. Interpretation requires reference data, against which the obtained measurement data can be compared. Reference data represent a population of similar objects (in our case, user manuals), and they are called a quality profile. In the case of *subjective quality indicators* both the profile and measurement data should be represented in the relative form – this allows one to compare user manuals evaluated by different numbers of people. An example of a quality profile for user manuals is presented in Table 4.6.

4.4.2 Evaluation procedure

The proposed evaluation procedure is based on *Management Reviews* of IEEE Std. 1028:2008. This type of review was selected on the grounds that it is very general and can be easily adapted to any particular context.

Moreover, the proposed procedure applies very well to quality management activities undertaken within the framework of PRINCE2 [148]. PRINCE2 is a project management methodology developed under the auspices of UK's Office of Government Commerce (OCG). Quality management is the central theme of PRINCE2. It is based on two pillars: *Product Description* and *Quality Register*. *Product Description* (one for each product being a part of project output) specifies not only the product's purpose and its composition, but also the quality criteria (with their tolerances), quality methods to be used, and the roles to be played when using the quality methods. In PRINCE2, quality methods are split into two categories:

- in-process methods: they are the means by which quality can be *built into* the products – these are out of scope of this paper,
- appraisal methods: using them allows the quality of the finished products to be assessed – these are what the proposed evaluation procedure is concerned with.

Quality Register is a place (database) where the records concerning planned or performed quality activities are stored.

Roles

The following roles participate in user documentation evaluation:

- **Decision Maker** uses results from the evaluation to decide whether user documentation is appropriate for its purpose or not.

- **Prospective User** is going to use the system documented by the user documentation. For evaluation purposes, it is important that a *Prospective user* does not yet know the system. This lack of knowledge about the system is, from the evaluation point of view, an important attribute of a person in this role.
- **Expert** knows the system very well, or at least its requirements if the system is not ready yet.
- **Review Leader** is responsible for organizing the evaluation and preparing a report for the *Decision Maker*.

Input

The following items should be provided before examining the user documentation:

1. Evaluation mandate for *Review Leader* (see below)
2. Evaluation forms for *Prospective Users*, *Experts* and *Review Leader* (Appendix A.2 contains an example of such a form)
3. User documentation under examination
4. Template for an evaluation report (see Appendix A.3)

EVALUATION MANDATE

Evaluation Mandate is composed of five parts (an example is given in Appendix A.1):

- **Header** – besides auxiliary data such as id, software name, file name, etc., it includes the purpose, scope and the evaluation approach:
  - **Purpose of examination** – There are two variants: *Acceptance* and *Selection.*
  - **Scope of evaluation** – The evaluation can be based on *exhaustive reading* (one is asked to read the whole document) or *sample reading* (reading is limited to a selected subset of chapters). *Sample reading* allows saving effort but makes evaluation less accurate.
  - **Evaluation approach –** Depending on available time and resources, different approaches to evaluation can be employed. One can decide to organize a physical meeting or use electronic communication only. Furthermore, the examination can be carried out individually or in groups (e.g. Wideband Delphi[100]). Each meeting can be supported by a number of forms (e.g. *evaluation forms*) and guidelines which should be available before the examination.

- **Evaluation grades** – These grades depend on the purpose of the examination. In the case of *Acceptance* evaluation, typical grades are the following: *accept*, *accept with minor revision* (necessary modifications are very easy to introduce and no other evaluation meeting is necessary), *accept with major revision* (identified defects are not easy to fix and a new version should go through another evaluation), *reject* (quality of the submitted documentation is unacceptable and other corrective actions concerning the staff or process of writing must be taken). These grades can be given on the basis of evaluation data presented together with the population profile. In the case of *Selection* between variants *A* and *B* of the documentation, the grades can be based on the 5-grade scale: *variant A when compared to variant B is definitely better/ rather better/ hard to say/ rather worse/ definitely worse.*

- **Selection of quality questions** – One should choose quality questions (see Table 4.4) to be used during evaluation. Each question should be assigned to roles taking into account the knowledge, experience and motivation of people assigned to each role. For example, it is hard to expect from people who do not know the system (or requirements) that they decide whether *user documentation* describes all the functionality supported by the system, thus evaluation of *Completeness* in such conditions may provide insignificant results.

Evaluation Mandate can be derived from information available in project documentation. For example, a project in which PRINCE2 [148] is used should contain a *Product Description* for user documentation. An *Evaluation Mandate* can be derived from that description. In PRINCE2 *Product Description* contains, among others, *Quality Criteria* and *Quality Method* (see Appendix A.17 in [148]). The *Scope of evaluation* and *Evaluation approach* can be derived from *Quality Method*, and *Selection of quality questions* follows from *Quality Criteria*. *Purpose of examination* usually will be set to *Acceptance* (*Selection* will be used only in research-like projects when one wants to compare different methods or tools).

Evaluation

Activities required to evaluate user documentation are presented in Figure 4.1 in the form of a use case [29]. Use cases seem to be a good option as they can be easily understood, even by IT-laymen.

UC: **Evaluation of user documentation**
Main scenario:

1. Review Leader creates, on behalf of Decision Maker, an *Evaluation Mandate*. He also prepares *Evaluation Forms*.
2. Experts assess the *user documentation* from the point of view of the quality characteristics assigned to them (e.g. Completeness and Correctness) and fill in the *Evaluation Forms*.
3. Review leader gets the *Evaluation Forms*.
4. Prospective Users assess the *user documentation* from the point of view of the quality characteristics assigned to them (e.g. Operability and Appearance) and fill in the *Evaluation Forms*.
5. Review Leader collects the *Evaluation Forms*, determines the final grade and writes her/his *Evaluation Report*.

Exceptions:

3.A. Experts' evaluation is negative.

3.A.1. Go to step 5.

Figure 4.1: Procedure for evaluation of user documentation

Quality evaluation procedure vs. *management reviews*

The proposed procedure differs from the classical *Management review* [62] in the following aspects:

- The proposed procedure has a clear interface to PRINCE2's *Product Description* through *Evaluation Mandate* (see Section 4.4.2).
- Experts (their counterparts in *Management Review* are called Technical staff) and Prospective Users (in *Management Review* they are called User representatives) have clearly defined responsibilities (see Figure 4.1).
- Decision making is based on clearly described multiple criteria accompanied by a quality profile describing previously evaluated documents (see Interpretation of Section 4.4.1 and Appendix A.3).

4.4.3 Quality profile for user documentation

In the case of *Acceptance* it is proposed that a given user documentation is compared with other user manuals created by a given organization (e.g. company) or available on the market. Instead of comparing user documentation at hand with *n* other documents, one by one, it is proposed that those *n* documents are evaluated, a quality profile describing an average user documentation is created and the given user documentation is compared with the quality profile (see Table 4.6).

To give an example, a small research has been conducted, the goal of which can be described as follows:

> *Analyze a set of user manuals for the purpose of creating a quality profile from the point of view of end-users and in the context in which the role of end-users is played by students and the role of Experts is played by researchers and Ph.D. students.*

The evaluation experiment was designed in the following way:

- For each considered user manual, one of the authors played the role of Review Leader, three Experts were assigned from Ph.D. students and staff members, and 16-17 students were engaged to play the role of Prospective Users.
- The evaluation was performed as a controlled experiment based on the procedure described in Figure 4.1.
- The evaluation time available to Prospective Users was limited to 90 min. None of the subjects exceeded the allotted time.
- The evaluated user manuals were selected to describe commercial systems and concerned a domain which was not difficult to understand for the subjects playing the role of Prospective Users. The user manuals were connected with the products available on the Polish market which are presented in Table 4.5. For *Plagiarism.pl*, *nSzkoła*, and *Hermes* the whole user manual was evaluated; in all the other cases, only selected chapters describing a consistent subset of functionality went through review.

The resulting quality profile is presented in Table 4.6 and the data collected during evaluation are available in Appendix A.4. As the role of experts was played by Ph.D. students and staff members, who knew only some of the systems used in the experiment, the percentage of *g* (good) and *VG* (very good) grades shown in Table 4.6 (questions Q1 and Q5) should be regarded rather as upper limits (real experts could identify some functionality provided by the system which was not covered in the evaluated users manuals, or some additional incorrect descriptions).

How to use the data of a quality profile such as the one presented in Table 4.6 is another question. When making a final decision (to accept or reject a user

Table 4.5: List of evaluated user manuals (pages are counted without cover page and table of contents; last column presents number of Experts and Users participating in an evaluation)

User manual	**Description**	**Pages**	**Experts/ Users**
Plagiarism.pl – Manual for individual user (in Polish *Plagiat.pl – Instrukcja użytkownika indywidualnego*)	The system allows detection of plagiarism in different types of documents, e.g. M.Sc. thesis.	13	3/16
Getting Started with Deanery.XP 9.65.5.0 (in Polish *Podstawy obsługi Dziekanatu.XP*)	Supports staff of a dean office in management of students at a university.	19	3/17
Optivum Secretariat – User manual (in Polish *Sekretariat Optivum – Podręcznik użytkownika programu*)	Supports management of a primary and secondary school.	25	3/17
User manual for nSzkoła platform – Student's panel (in Polish *Instrukcja obsługi Platformy nSzkoła – Panel Ucznia*)	Allows students to read records in an electronic log.	16	3/16
Secretariat DDJ 6.8 (in Polish *Sekretariat DDJ 6.8*)	Supports management of a school.	21	3/16
LangSystem 4.2.5 – User documentation (in Polish *LangSystem 4.2.5 – Dokumentacja użytkownika*)	Supports management of a school of foreign languages.	22	3/17
SchoolManager – User manual (in Polish *School Manager – Podręcznik użytkownika*)	Supports management of a school of foreign languages.	27	3/17
User manual for Hermes 2012 (in Polish *Instrukcja obsługi aplikacji HERMES 2012*)	Collecting data about examinations concerning professional qualifications.	21	3/16
E-grades: Electronic log (in Polish *Dziennik elektroniczny e-oceny*)	Allows students to read records in an electronic log.	23	3/16

manual) one can use one of many multi-attribute decision making methods and tools (there are many of them – see e.g. [43, 156]). For instance one could use the notion of dominance and require that a given user manual gets a score, for every criterion (characteristic), not worse than a given threshold. Such a threshold could be calculated, for instance, as a percentage of *g* and *VG* answers to each question. It is also possible to infer thresholds from a historical database, providing that the database contains both evaluation answers and final decisions (or customer opinions).

When using the profile presented in Table 4.6 one should be aware that all the evaluated documents are connected with educational software (see Table 4.5). So, one must be careful when using the presented profile in other contexts. We believe that a profile, such as of Table 4.6 can be useful especially when a company or a project does not have its own quality profile. To support this we established a web page with results from ongoing evaluations[2].

4.5 Empirical evaluation of operability

To evaluate a user manual experimentally, one can use a form of Browser Evaluation Test (BET) [152]. The BET method was developed to evaluate the quality of meeting browsers based on a video recording of a meeting. In such an evaluation each subject is given a list of complementary assertions (one is true and the other is false), and must identify which of the two is true (e.g. one is *Susan says the footstool is not expensive* and the other is *Susan says the footstool is expensive* [152]). Obviously, by making lucky guesses one can get a score of about 50%. From our point of view this is unacceptable. To help this, a variant of BET was developed (see below) which is oriented towards evaluation of user documentation (it is called Documentation Evaluation Test - DET) and by guessing one can get a score of about 25%. The DET procedure is presented in Figure 4.2.

DOCUMENTATION EVALUATION TEST

4.5.1 DET questions

*Question*s are very important for the effectiveness of the DET procedure. An exemplary question is presented in Table 4.7. A DET *question* consists of a *theme* (e.g. *The following items are included into a similarity report*) and four proposed answers of which one is correct and the other three are false. Every question is accompanied by an auxiliary statement (*I could not find the answer*) which is to be evaluated by the subject (true/false). That statement allows subjects to say that for some reasons

[2]http://coca.cs.put.poznan.pl/

Table 4.6: An exemplary quality profile (9 user manuals, 3 experts, 16-17 prospective users per manual; abbreviations: *N* - Not at all, *w* - Weak, *?* - Hard to say, *g* - Good enough *VG* - Very good)

Id	**Questions**	**N**	**w**	**?**	**g**	**VG**
Completeness						*responsible: Expert*
Q1	To what extent does the user documentation cover all the functionality provided by the system with the needed level of detail?	3.7%	18.5%	29.6%	44.4%	3.7%
Q2	To what extent does the user documentation provide information which is helpful in deciding whether the system is appropriate for the needs of prospective users?	0.0%	3.7%	11.1%	55.6%	29.6%
						responsible: Prospective User
Q3	To what extent does the user documentation contain information about how to use it with effectiveness and efficiency?	6.1%	9.5%	7.4%	50.0%	27.0%
Operability						*responsible: Prospective User*
Q4	To what extent is the user documentation easy to use and helpful when operating the system documented by it?	1.4%	6.8%	14.9%	48.0%	29.1%
Correctness						*responsible: Expert*
Q5	To what extent does the user documentation provide correct descriptions with the needed degree of precision?	0.0%	18.5%	25.9%	44.4%	11.1%
Appearance						*responsible: Prospective User*
Q6	To what extent is the information contained in the user documentation presented in an aesthetic way?	1.4%	12.2%	12.2%	49.3%	25.0%

they failed when trying to find the answer. Questions with answers and additional statements are used to create a *Knowledge Test* which is presented to subjects during an evaluation.

When analyzing questions provided by Experts at early stages of this research, we identified a number of weaknesses, which are unacceptable:

W1. Some choices were synonyms, e.g. *month* and *1/12 of year.*

W2. Some choices were answers to other questions.

W3. Some questions were suggesting a number of choices (e.g. *The following values* `are` *correct ISBN numbers*).

W4. Some references to the user interface were imprecise, especially when elements with the same name occur multiple times in a different context.

Table 4.7: Exemplary question

Question no 2	
The following items are included into a similarity report:	
Choose one of the proposed answers:	*Correct?*
A) Info about whether a given document is plagiarised	
B) Similarity coefficients and a list of similar documents	
C) Similarity coefficients, a list of similar documents and whether a given document is plagiarised	
D) Similarity coefficients, a list of similar documents and fragments of the document which have been found in another document	
The answer is in the user documentation on page:	
I could not find the answer:	

W5. Some choices did not require the user manual to make a selection – it was enough to use general knowledge.

To cope with these weaknesses, a set of guidelines was formulated. Here they are:

- the choices of questions should not contain a synonym of any other choice (addresses weakness W1).
- the choices of questions should not contain an answer to any other question (addresses weakness W2).

UC: **Documentation Evaluation Test**
Main scenario:

1. Experts individually read *user documentation*, create *Question*s and pass them to Review Leader.
2. Review Leader cleans the *Question*s submitted by the Experts (i.e. removes duplicates, corrects spelling, etc.).
3. Review Leader prepares a *Knowledge Test* by random selection of *Question*s.
4. Prospective Users, to assess *Operability*, take an open-book *Knowledge Test* (the book is the *user documentation*).
5. Review Leader writes a *Review Report* concerning the *user documentation.*

Extensions:

3.A. Review Leader realizes that the number of *Question*s is too small.

3.A.1. Review Leader asks one more Expert to perform step 1.

Figure 4.2: The DET procedure

- questions should not suggest a number of choices (addresses weakness W3).
- references to the user interface must be unambiguous (addresses weakness W4).
- selecting a choice must require information contained in the user documentation (addresses weakness W5).

4.5.2 Case studies

To characterize the DET method, we have analyzed five user manuals with the aim of presenting an example of how such an evaluation could be conducted. Each user manual was assessed with the following purpose in mind:

> *Analyze the user manual for the purpose of quality evaluation with respect to Operability, from the point of view of end-users in the context of Ph.D. students playing the role of Experts and students as Prospective Users.*

The evaluation experiment was designed in similarly to the one presented in Section 4.4.3. The evaluation procedure used in the experiment is described in Figure 4.2 and the manuals are listed in Table 4.8. All of them had been checked earlier for Completeness and Correctness by Experts (that role was played by three researchers and Ph.D. students) and it was executed as a one-person review (see Appendix A.4 for results of the Completeness and Correctness checks).

The data collected during the evaluation are summarized in Table 4.8. The average speed of reading a manual by a Prospective User was about 4 pages per 10 min and the average percentage of correct answers was about 81%. Table 4.9 contains data concerning preparation of questions. There are two numbers referring to questions: total number of questions and final number of questions. The first one describes total number of questions proposed by the experts. Some of those questions overlapped, so the final number of questions included in the *Knowledge test* was a bit smaller (e.g. for *Plagiarism.pl* 31 questions have been proposed and 29 of them have been included into the *Knowledge test*). The average speed of writing questions is about 6 questions per hour. One can use those data as reference values when organizing one's own DET evaluation.

4.6 Related work

One could consider the 265nm series of ISO/IEC standards [67, 68, 71, 74, 75] as a quality model for user documentation as those standards present a number of aspects concerning the quality of user documentation. Unfortunately, those aspects do not constitute an orthogonal quality model. For example, *completeness of information* contains *error messages* as its subcharacteristic. On the other hand, *safety* is described as containing *warnings and cautions*. Thus, the scope of *completeness of information* overlaps the scope of *safety*. Another example is *Technical accuracy*, which is described as *consistency with the product*, and *Navigation and display* which requires that *all images or icons [...] are correctly mapped to the application* – those two characteristics overlap. A similar relation exists between *Technical accuracy* and *Accuracy of information*, which – according to its description – should *accurately reflect the functions* of the software. Thus, the intention of the authors of the standards was not to present an orthogonal quality model, but rather the way in which user documentation should be assessed.

Markel [98] presented *eight measures of excellence* which are important in technical communication: *honesty*, *clarity*, *accuracy*, *comprehensiveness*, *accessibility*, *conciseness*, *professional appearance* and *correctness*. Each item on the list is described, and why it is important from the quality perspective is explained. Unfortunately, there is no information on how to evaluate the presented *measures*. Moreover, some of these *measures* overlap, i.e. both *honesty* and *accuracy* emphasize the importance

Table 4.8: Results of DET evaluation

User documentation	No. of participants	No. of pages	Average answer time [min]	No. of questions	Average percentage of correct answers
Plagiarism.pl	16	13	39	29	82.97%
Deanery.XP	17	19	40	28	86.97%
Optivum Secretariat	17	25	61	30	76.47%
LangSystem	17	22	52	30	81.76%
Hermes	16	21	52	28	77.01%
Total	83	100	244	145	

Table 4.9: Preparation of questions for DET evaluation

User documentation	No. of experts	Final / total no. of questions	Total time of writing questions [min]	Average time for one final question [min]
Plagiarism.pl	3	29/31	329	11.3
Deanery.XP	3	28/31	365	13.0
Optivum Secretariat	3	30/32	264	8.8
LangSystem	3	30/31	350	11.7
Hermes	3	28/30	224	7.7
Average		29/31	306.4	10.6

of not misleading the readers. Moreover, *honesty* is not a characteristic of a user manual but rather a relation between a writer and his/her work (a reviewer can only observe inconsistency between a user manual and the corresponding software but is not able to say if those defects follow from bad will or whether they occurred by chance).

Allwood et al. [14] described the process of assessing the usability of a user manual by reading it and noting difficulties. During the evaluation, participants are asked to rate, for each page of a user manual, its *usability*, *comprehensibility*, *readability*, and how *interesting* and *stimulating* it is. Again, the orthogonality of the proposed model is questionable as *usability* strongly depends on the *comprehensibility* of user documentation. Moreover, if the proposed model is to be complete, *usability* should cover operability. As operability depends on *readability* (if a user document is not readable then it will take longer to get information from it, and thus its operability will suffer), *usability* and *readability* overlap.

Other quality models considered in this paper are Ortega's systemic quality model and Steidl's characteristics for code comments. They do not directly relate to user documentation but contain quality characteristics that can be "translated" to the context of user documentation. We used them to examine completeness of the COCA model (see Section 4.3, justification for Claim 2).

4.7 Conclusions

This article presents the COCA quality model, which can be used to assess the quality of user documentation. It consists of only four characteristics: Completeness, Operability, Correctness, and Appearance. The model is claimed to be orthogonal and complete, and justification for the claims are presented in Section 4.3. As quality evaluation resembles measurement, the GQM approach [151] was used to define the goal of evaluation, the questions about quality one should be interested in, and the quality indicators which, when compared with the quality profile for a given area of application, help to answer those questions. The empirical data (quality profile) have been obtained by evaluating 9 user manuals available on the Polish market which concern education-oriented software (see Table 4.6). The collected data are pretty interesting. Although the evaluated user manuals concern commercial software, their quality is not very high. For instance, only in 48.1% of the cases the Experts evaluated the manuals as *good* or *very good* with respect to functional completeness of the examined user documentation (question Q1 in Table 4.6), in 22.2% of the cases the answer was *weak* or *not-at-all*.

Quality of user documentation can be evaluated with the COCA model using two approaches: *pure review* based on *Management Review* of IEEE Std. 1028:2008 (see Section 4.4.2), or *mixed evaluation* where Completeness, Correctness and Appearance are evaluated using *Management Review* and Operability is evaluated experimentally using the DET method proposed in Section 4.5. That method is based on questions prepared by experts. The operability indicator is defined as the percentage of correct answers given by a sample of prospective users. Empirical data concerning DET-based evaluation show that, on average, there is about 1.5 questions per page of user documentation (see Table 4.8) and, on average, it takes an expert about 10 minutes to prepare one question. In the DET-based evaluation prospective users read a user manual at the average speed of about 25 pages per hour and for documentation concerning commercially available software the average percentage of correct answers is between 77% and 87%.

Future work should mainly focus on further development of the quality profile, of which an initial version is presented in Section 4.4.3 (Table 4.6) and Section 4.5.2 (the rightmost column of Table 4.8). It would also be interesting to investigate Operability indicators based on readability formulae such as SMOG [102] or the Fog Index [50] (the Fog Index was used by Khamis to assess the quality of source code comments [86]; a similar approach could be applied to user manuals).

Acknowledgements

This work has been supported by the Polish National Science Centre based on the decisions DEC-2011/03/N/ST6/03016.

Chapter 5

Automatic explanation of field syntax in web applications

Preface

This chapter contains the report: *Bartosz Alchimowicz, Jerzy Nawrocki, Mirosław Ochodek: Towards automatic explanation of field syntax in web applications, Politechnika Poznańska, RA-10/2014.*

My contribution to this work included the following tasks: *1)* co-design and implementation of generation methods; *2)* conducting the experimental evaluation.

Context: Users of web applications are often asked to enter texts into form fields with special syntax (e.g. ISBN number) described by a regular expression. For instance, in HTML5 there is the "input" tag, used to declare a form field, and it can have the attribute "pattern" which contains a regular expression. Sometimes such fields can pose a substantial challenge for an end-user who is an IT-layman. Thus, an explanation of the syntax would be needed (in HTML5 it can be provided via the "title" attribute). Unfortunately, writing good explanations (especially in multiple languages) can be quite time consuming.

Objective: The aim of this work is to check whether it is possible to automatically generate an explanation of form field syntax that would be no worse than a man-made one.

Method: It has been assumed that a good explanation of form field syntax should consist of three parts: narrative explanation (variants in multiple natural languages should be available in interest of localizability), syntax diagram, and a set of correct and erroneous examples of input text. For the sake of modifiability, a rule-based domain specific language has been designed which resembles a compiler's syntax-directed definition. Using that language, one can specify the rules of generating narrative descriptions in various natural languages (taking into account the inflection characteristic of Slavonic languages), rules of splitting a complex regular expression into a set of easy-to-comprehend subexpressions, and also rules of assigning meaningful names to automatically extracted subexpressions.

As regards the generation of erroneous examples, a heuristic algorithm has been proposed that produces easy-to-explain incorrect texts. The algorithm is based on two mutations, removal and contamination, which destroy a regular expression in a "controlled" way.

To check the quality of the proposed method of generating form field syntax explanations, a controlled experiment has been conducted. A set of regular expressions was selected and a group of 15 students of Software Engineering were asked to prepare explanations for them. The same regular expressions were also used to generate explanations using the proposed method. Next, 207 subjects were given a set of exemplary input texts and syntax explanations. Their task was to mark input texts as correct or incorrect. The score was the average percentage of correct marks.
Results: The average percentage of correct marks for the group using automatically generated explanations was 84%, while for the subjects who were given man-made explanations it was less than 79%.
Conclusion: It seems possible to automatically generate an explanation of form field syntax that would be no worse than a man-made one. The proposed methods of generating multiple-language narrative explanations and easy-to-explain erroneous examples could be used to make a practical tool (possibly a web service) that would generate explanations for HTML5/JavaScript regular expressions.

5.1 Introduction

The syntax of text fields is often not very well explained in applications. This is not a problem for typical strings, like the user's name or age. However, there are strings that may by challenging for end-users, like ISBN, ISSN and others. To help users, one can include an explanation of a field's syntax in an application or place it in a user manual. However, complex fields do not occur frequently which is why their description is often omitted and users are left without any help. A solution to this problem could be the automatic generation of an explanation of a field on the basis of information available in the source code.

Such an explanation could prevent problems arising from a lack of information needed by users. According to data presented by Markel, the average cost of one support call in the year 2008 was above $32 [98]. Moreover, a user who cannot find a solution to a problem searches for it among other users [111], which means that in a company she/he disturbs other co-workers. Another argument for explaining field syntax is the recommendation of ISO/IEC Std. 26514:2008 [67], according to which an explanation of a field should appear in the user documentation and in the application.

In our research we assumed that in order to check the syntactical correctness of a field's input, a programmer should describe the field's syntax with a regular expression, and on the basis of the regular expression an explanation of the field's syntax is generated in an automatic way.

A regular expression is the most prevalent notation for the validation of input strings and is widely used in computer languages (e.g. HTML5, Java, Perl, Python, etc. [48]), lexical analyzers (e.g. Lex [93]) and other tools. Moreover, a number of extensions were created (e.g. for the Perl language). There is also a standard for describing regular expressions (see the POSIX documentation [147]).

If the input is syntactically incorrect, the application should support the end-user and explain the structure of acceptable input. The simplest solution is just to show the regular expression entered by the programmer. However, that could be unreadable to the end users (as shown by Erwig and Gopinath [42], even programmers have difficulty in understanding complex regular expressions). The need to explain regular expressions was recognized by many authors, including Ranta [123], Erwing and Gopinath [42], and Blackwell [20, 21]. Moreover, a number of tools were created for that purpose, like YAPE [144], and RegExpert [26]. Unfortunately, none of the proposed methods was aimed at explaining regular expressions to an IT-layman.

In this paper we describe the Field Explanation System which generates a field explanation that takes a regular expression describing the syntax of the field as the input. The generated explanation consists of three parts: a narrative explanation, examples (positive and negative), and a visual representation. This paper extends the previous research carried out by Alchimowicz and Nawrocki [9] (see Appendix C) that focused on the visual representation of regular expressions using syntax diagrams. It focuses on the generation of narrative explanations and also briefly discusses explanation by examples.

The paper is organized as follows: In Section 5.2, the problem of automatic generation of a field explanation is defined. Section 5.3 presents templates used for text generation. Section 5.4 introduces grammatical attributes. Section 5.5 describes conditional templates. Section 5.6 shows how to cope with EBNF extensions which result in repetitions of nonterminals in a syntax tree. Section 5.7 presents how to identify a regular expression that can lead to an empty. Section 5.8 introduces patterns which allow to produce explanations better than those that can be obtained using the standard rules. Section 5.9 focuses on the generation of referring expressions. Section 5.10 presents how to assign easy-to-memorize names for parts of complex regular expressions. Section 5.11 describes the generation of examples and the focus is on generating erroneous ones. This is not a trivial task, if one expects to obtain erroneous examples for which it is easy-to-explain what is wrong with them. Section 5.12 presents an empirical evaluation and Section 5.13 discusses related work. Finally, a summary of the findings and conclusions can be found in Section 5.14.

5.2 Problem

As was stated in Section 5.1, we are interested in the automatic generation of explanations of field syntax for fields that appear in web applications. To accomplish this goal one must solve the following problem.

PROBLEM 1. Given a field description composed of a field name and its syntax described by a regular expression, generate a field explanation explaining its syntax. The generated explanation should be at least as helpful as explanations written by humans.

REQUIREMENT 1. (Multilingualism) A generator of field explanations should be multilingual, i.e. it should support the generation of field descriptions in various natural languages.

JUSTIFICATION. English is a very popular language, but many web applications are still used by people who would prefer to speak another language in their daily activities. Therefore, it would be useful if the generator supported many languages. The current version of the system supports English and Polish. □

ASSUMPTION 1. An explanation of a regular expression should consist of a diagrammatic representation (based on the concept of syntax diagrams [9]), a narrative explanation, and a set of examples with correct and incorrect input. In the remaining part of the paper this type of explanation is called a 3-fold explanation.

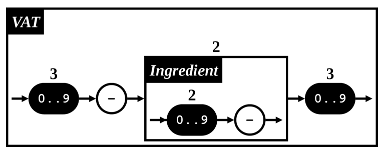
JUSTIFICATION. The pertinence of this assumption was somehow confirmed by empirical evaluation of the presented method (see Sec. 5.12). Nevertheless, further study would be necessary to investigate the importance of each part of the 3-fold explanation. □

For example, for the VAT field whose syntax is described by the following regular expression:

```
VAT = [0-9]{3}-([0-9]{2}-){2}[0-9]{3}
```

one would get for the English language the 3-fold explanation presented in Fig. 5.1.

VAT is described by the following diagram:



VAT consists of 3 digits, a hyphen (-), two Ingredients and three digits.
An **Ingredient** consists of two digits and a hyphen (-).

Example	Correct?
`948-93-00-158`	Yes
`195-19-75-984`	Yes
`793-28-87-441`	Yes
`563-328`	No (absence of `Header`)
`23987-58-87-441`	No (too long)

Figure 5.1: An example of a 3-fold explanation

5.3 Syntax-Directed Flexible Templates

As the input is an expression of a formal language, a generator of field explanations is a kind of translator and generation rules can be described as syntax-directed definitions [8]. The main difference between a compiler and an explanation generator is the output: for a compiler it is another formal language while for an explanation generator it is a natural language. Nevertheless, translation rules in both cases can be described in a similar way, i.e. they can be based on production rules describing the input. For the purpose of generation of field explanations, the use of a special set of translation rules is proposed, called here flex-templates, where flex stands for 'flexible and extendible'. Explanation rules based on flex templates have syntax as presented below. In the paper an EBNF-like notation is used [63], i.e. the right-hand side of a production rule can contain a regular expression built over terminal and nonterminal symbols of grammar rather than just a finite sequence of those symbols. However, since none of the names of the nonterminals that appear in this paper has a space inside, we decided to skip colons that in EBNF denote concatenation.

```
      Rule = Production
             ( Lang ':' Template )+
             ( AttributEval )*
             ';' ;
Production = Parent '=' RHS ;
```

The nonterminal `Rule` represents a whole explanation rule. Each explanation rule consists of a `Production` (where the production is a part of the grammar describing the syntax of regular expressions), a `Template` for each natural language, `Lang`, served by those rules, and a possibly empty sequence of `AttributEvals` defining the evaluation of general-purpose attributes (those attributes are discussed in

Sec. 5.7 and—as they are optional—for the time being they will be neglected). Each `Production` is a pair consisting of a parent nonterminal symbol, a `Parent`, and a right-hand side, `RHS`, that is a nonempty sequence of simple right-hand sides or repeatable phrases (`SimpleRHS` and `RepPhrase` respectively).

```
Parent =   Nonterm  ;
   RHS = ( SimpleRHS | RepPhrase )+ ;
```

Repeatable phrases (`RepPhrase`) will be discussed in the next section. A simple right-hand side is either a terminal symbol or a nonterminal one (as in classical context-free grammars):

```
SimpleRHS = ( Term | Nonterm ) ;
```

A nonterminal symbol is a sequence of letters (lower or upper case) and—for the sake of readability—it is underlined. Here are a few examples of nonterminal symbols: `Regex`, `Component`, `Factor`. Sometimes there is a need to make a distinction between two or more occurrences of the nonterminal symbol in a production rule, to allow referral to a particular occurrence in a template (this is discussed in detail in Sec. 5.6). Then one can augment a nonterminal symbol with a 1-digit suffix, e.g. `Factor`1, `Factor`2 denote different occurrences of the nonterminal `Factor`.

Each natural language `Lang` is specified using ISO Standard 639-1 (e.g. English is denoted as "EN", Polish as "PL" and so on). The advantage of this standard is the fixed length of all the symbols (they are 2-character long). We call those symbols 'language symbols' and denote them as `LangSymbol`. They are composed of upper case letters. The language symbol is followed by a possibly empty list of synthesized attributes, `Syn`—for the time being let us assume the list is empty (those attributes are discussed later in this section):

```
Lang = LangSymbol '(' Syn? ')' ;
```

`Template` describes a template to be used for a given natural language. It represents a sequence of boilerplates (denoted here as `Boilerplate`), nonterminal gaps (`NontermGap`), conditional fragments (`CondFragment`), and repeatable fragments (`RepFragment`):

```
Template = ( Boilerplate | NontermGap | CondFragment | RepFragment)+ ;
```

With the exception of boilerplates, all will be described later (Nonterminal gaps in Sec. 5.4, conditional fragments in Sec. 5.5, and repeatable fragments in Sec. 5.6). As regards boilerplates, roughly speaking they are a sequence of words of a given

```
Zero= "0"
EN(): "null "
    ;
```

Figure 5.2: An extremely simple explanation rule.

natural language (a word is defined as a nonempty sequence of letters that is subject to grammatical adjustment—this issue is discussed in Sec. 5.4) surrounded by strings of any characters including letters, digits, colons, dots, space characters etc.

An extremely simple example of an explanation rule is given in Fig. 5.2. In the `Production` part, the parent is nonterminal `Zero`, and `SimpleRHS` is the single-character string "`0`" (it is derived from the terminal symbol `Term`). In Fig. 5.2 the only natural language is British English (denoted by "`EN()`"), and the template consists of one `Boilerplate` which is the string "`null `". When "`0`" is encountered in the input, the string "`null `" will be generated as (part of) the output.

5.4 Grammatical attributes

It is assumed that the final explanation will be based on a superposition of the provided templates. Here 'superposition' means that one template can be 'inserted' into another using gaps marked with nonterminal symbols (the symbol `NontermGap` represents those gaps in a template). Superposition is attractive, but it requires some adjustment of the template that is to be inserted in a given place. That adjustment must take into account the context where a given template is to be inserted. The information about the context is passed into a given template through grammatical attributes.

In general, there are two kinds of attributes: general-purpose attributes and grammatical ones. Both of them must be declared in an appropriate way:

```
AtriDec = ( GrAtriDec | GenAtriDec )* ;
```

General-purpose attributes are discussed in Sec. 5.7. In this section the focus is on grammatical attributes.

Declaration of a grammatical attribute provides information about the possible values of that attribute. As the set of those values is finite, declaration of a grammatical attribute, `GrAtriDec`, resembles the declaration of an enumeration type in programming languages:

```
GrAtriDec = GrAtriName ':' GrAtriVal (',' GrAtriVal)* ';' ;
```

GrAtriName is the name of a given grammatical attribute (it is a sequence of lower case letters), and GrAtriVal is one of the possible values of that attribute (only upper case letters and digits are allowed). The second part of a grammatical attributes declaration is a flow description:

```
GrAtriFlow = LangSymbol ':' 'Nonterm' '(' Syn ')' '<' Inh ';' ;
```

Inh and Syn are lists of names of inherited and synthesized grammatical attributes, respectively:

```
Inh = GrAtriName (',' GrAtriName)* ;
Syn = GrAtriName (',' GrAtriName)* ;
```

Inherited attributes are passed from a parent to its children and they are used to transfer information about the context in which a given piece of text will be inserted (that piece of text is to be adjusted accordingly). Synthesized attributes are passed from a child to its parent and they convey information that allows adjustment of the context. Examples of both types of adjustments will be presented later. Before that, an example of attribute declaration will be presented. That example is based on the following assumption:

ASSUMPTION 2. For the purpose of generating an explanation of a regular expression, it is enough to take into account the grammatical number (singular or plural), the grammatical gender (feminine, masculine or neuter), and six grammatical cases: Nominative, Genitive, Dative, Accusative, Instrumental, and Locative [146].

JUSTIFICATION. As the purpose is to describe objects (i.e. strings of characters), one can assume, without loss of generality, that the tense will always be present tense and the grammatical person will be the third one. Putting the templates together can require adjustment of the grammatical number of the nouns or noun phrases that appear in the templates. Moreover, in some languages (e.g. in Slavonic languages like Polish or Russian), the grammatical case of a noun or noun phrase influences the ending of the noun (nouns are subject to inflection). For instance, in Polish, the Nominative of "colon" is "dwukropek", and the Instrumental is "dwukropkiem". Another linguistic phenomenon that must be taken into account is the grammatical gender of the adjective inherited from the grammatical gender of the noun (or noun phrase) described by that adjective. Like the inflection of nouns, there is inflection of adjectives, i.e. the ending of an adjective depends on its grammatical gender (this phenomenon appears, for instance, in Slavonic languages). □

Taking into account Assumption 2, one could declare the set of grammatical attributes as presented in Fig. 5.3 (the text enclosed between "/*" and "*/" is a

```
        /* Cases:      */
  c: N  /* Nominativus */, G  /* Genetivus       */,  D  /* Dativus   */,
     A  /* Accusativus */, I  /* Instrumentalis */,  L  /* Locativus */;
        /* Gender:     */
  g: F  /* Feminine    */, M  /* Masculine */,        T  /* neuTer */;
        /* Number:     */
  n: 1  /* Singular    */, 2  /* Plural    */;

 PL: Nonterm(g) < n,c;
 EN: Nonterm( ) < n;
```

Figure 5.3: An exemplary declaration of attributes and their flow.

comment). It seems good practice to have the names of attributes and their values 1-character long: they will be used as annotations and as such they should not dominate the main text that is to be produced in the output.

The last two lines of Fig. 5.3 declare which of the declared attributes are synthesized (gender for Polish and none for English) and which are inherited (number and case for Polish, and only number for English). Using `Nonterm` in those declarations means that every nonterminal symbol in a given language has those attributes.

Knowing how to declare grammatical attributes, let us return to boilerplates, as attributes are used there. As has been already mentioned, a boilerplate contains words of a given language and those words need to be adjusted using information passed via grammatical attributes. The attributes used to adjust a particular word are assigned to it by placing them next to the word in the superscript. They are preceded by the redirection symbol "<". An example boilerplate using grammatical attributes is presented below:

$$\texttt{"nice child}^{<\ n}\ \texttt{"}$$

The final text depends on the value of the attribute n (grammatical number). If n = '2' (i.e. the number is plural), the word "*child*" would be replaced by "*children*" and the final text would be "*nice children* " (there is a dictionary which, given a particular word and its grammatical attributes, returns the word in the right form).

Generally speaking, boilerplate syntax is defined as follows:

```
Boilerplate = '"' ( Word ^{'<' GrAttribute (',' GrAttribute)*}
                  | AnyCharacter )+
              '"' ;
 GrAttribute = ( GrAtriName | GrAtriVal );
```

Although `GrAtribute` is defined as any name or value of a grammatical attribute, annotating a word with a fixed value of a grammatical attribute makes no sense – one can use a given word directly in the appropriate form (then such a word is treated as a sequence of AnyCharacters). As grammatical attributes are simply annotations to a word and the word itself is more important than its annotation, the annotations are specified in superscripts.

The second component of a template is a nonterminal gap (NontermGap). It is a nonterminal augmented with information about inherited and synthesized attributes:

$$\texttt{NontermGap = Nonterm}^{\texttt{'<' GrAttribute (',' GrAttribute)*}}_{\texttt{'>' GrAtriName (',' GrAtriName)*}}$$

Information about grammatical attributes inherited by a Nonterm is presented in superscript and is preceded by '<'. The list of inherited attributes can contain an attribute value or name of a grammatical attribute, inherited from the parent of a Nonterm or synthesized by a sibling of a Nonterm. Inherited attributes resemble input procedure parameters in programming languages. Synthesized attributes are written in subscript and are preceded by '>' ('<' and '>' should be read as simplified arrows). A synthesized attribute is similar to an output procedure parameter. Its value is conveyed from the inside of a Nonterm to its environment via a name resembling the name of an actual parameter.

In Fig. 5.4 there are three explanation rules for Polish (PL) and English (EN). The first one contains templates with nonterminal gaps. The second rule contains the keyword Text which represents the fragment of the input text that matches the right-hand side of the production – in this case it will be a digit found in the input (Text resembles the array yytext in YACC [78]). The Polish variant of the first explanation rule contains the word '*opcjonalny*' (in English: '*optional*') which is followed by the nonterminal gap Primary. When Factor is given n='1' (i.e. the grammatical number is singular) and c='G' (i.e. the grammatical case is Genitive), those attributes are assigned to the word '*opcjonalny*' and via the nonterminal gap Primary they are also transferred to the word '*cyfra*' (in English '*digit*'). As the Genitive of singular number for '*cyfra*' is '*cyfry*', and the Genitive of singular number and feminine gender for '*opcjonalny*' is '*opcjonalnej*', one will get the following explanation in Polish for the input '0?':

```
opcjonalnej cyfry 0
```

For English the explanation would be

```
optional digit 0
```

5.5 Conditional fragments

A conditional fragment (`CondFragment`) is either a single boilerplate or nonterminal gap preceded by a condition. It can also be a nonempty sequence of boilerplates and nonterminal gaps enclosed in brackets and preceded by a condition:

```
CondFragment = Condition '?'     ( Boilerplate | NontermGap )
             | Condition '?' '(' ( Boilerplate | NonTermGap )+ ')'
             ;
```

A condition can describe a constraint imposed on the value of an attribute. Here is an example of two alternative conditional fragments:

```
n=1 ? ("a sequence of " Factor< 2)
n=2 ? ("sequences, each composed of " Factor< 2)
```

If the grammatical number is singular (n equals 1), and assuming the explanation system responds to the input `0?`, according to the rules of Fig. 4 the generated fragment of the explanation would be "*a sequence of optional digits 0*"; otherwise (i.e. for the plural number) the generated text would be "*sequences, each composed of optional digits 0*" (in both cases the underlined text marks the strings generated from `Factor`$^{<\ 2}$).

5.6 Extensible templates

To make explanation rules compact and more readable, we have decided to allow repetitions within the right-hand side of a production (this concept is also present in the EBNF notation [63]). As stated in Sec. 5.3, the right-hand side of a production can

```
Factor= Primary "?"
  PL(g): "opcjonalny< n,c,g " Primary< n,c
                                     > g
  EN( ): "optional< n "    Primary< n
       ;
Primary= [0-9]
  PL(F): "cyfra< n,c " Text
  EN( ): "digit< n " Text
       ;
Primary= "?"
  PL(M): "znak< n,c zapytania "
  EN( ): "question mark< n "
        ;
```

Figure 5.4: Two explanation rules for Polish and English that uses grammatical attributes.

be a simple right-hand side (`SimpleRHS`) or a repeatable phrase (`RepPhrase`). Simple right-hand sides have already been discussed. A repeatable phrase, as the name suggests, is a phrase that can be repeated several times – the possible number of repetitions is specified as a range. Without loss of generality, one can reduce the number of possible forms of range expressions to two: *{ Min, }* and *{ Min, Max }*. The former represents a set of natural numbers from *Min* to infinity, and the latter – from *Min* to *Max*. The syntax of a repeatable phrase can be presented as follows:

```
RepPhrase = Phrase '{' ( Min ','  |  Min ',' Max ) '}'
          ;
   Phrase =           Nonterm
          | '(' Term* Nonterm Term* ')'
          ;
```

An example production written in the format presented above is given below:

```
Regex = Component ( "|" Component ){0,}
```

Regex is an instance of a `Parent`. The right-hand side of the production consists of one simple right-hand side (`Component`) and one repeatable phrase consisting of the vertical bar character (`"|"`) and the nonterminal `Component`. The phrase is repeatable 0 or more times (the range expression is '`{0,}`'). Taken together, `Regex` describes a nonempty sequence of `Component`s that are separated with '`|`'.

Two popular shortcuts are allowed for a phrase *p* (those shortcuts are developed into the full version during preprocessing):

```
p* = p {0, }
```

and

```
p+ = p {1, }
```

In symmetry with repeatable phrases in grammar productions, there are also repeatable fragments in a template (they have been mentioned in Sec. 5.3). Repeatable fragments of a template are written using Kleene's star ('*') which means zero or more repetitions:

```
RepFragment= '(' ( Boilerplate | NontermGap | CondFragment )+ ')' '*' ;
```

If a nonterminal symbol occurs more than once in the right-hand side of a production, then it should be appended with a digit (this has already been mentioned in Sec. 5.3). Thanks to this, one can refer to a given occurrence of the nonterminal in an unambiguous way. Let us consider, as an example, the following explanation rule:

```
Component= Factor1 Factor2* Factor3
    EN( ): n=1 ? "a sequence composed of "
           n=2 ? "sequences, each composed of "
           Factor1< 1 ", " (Factor2< 1 ", ")* "and " Factor3< 1
         ;
```

Assume the number (n) is singular (i.e. '1'), and from Component a sequence of two factors is derived, the first one matches '1' on the input, and the second one matches '3' (Factor is defined as in Fig. 4). Then the repeatable fragment would be reduced to an empty string and the above template would produce the following fragment of explanation:

```
a sequence composed of digit 1 , and digit 3
```

(the space character before the colon would be removed later, during the so called polishing phase, which is the last phase of generating a narrative explanation). If the input was '1', '2', '3', the corresponding output could be

```
a sequence composed of digit 1 , digit 2 , and digit 3
```

5.7 General purpose attributes

For some regular expressions, the languages described by them can contain an empty string. Using the already presented mechanisms, one would simply paraphrase a regular expression in a natural language. As a result, the final explanation would consist of sentences of the form "a possibly empty sequence of ..." or "a nonempty sequence of optional...". All such explanations have two characteristics: *1)* they allow empty strings, and *2)* they can be perfectly correct but not necessarily easy to comprehend. For the sake of understandability, it would be better to write directly that a given field can be left empty. For instance, one could generate an explanation of the form

"You can leave the field empty or enter...".

It is not difficult to decide if a regular expression defines language containing an empty string. An exemplary set of rules is presented in Table 5.1.

To allow implementation of rules similar to those presented in Table 5.1, one can use the general purpose attributes mentioned in Sec. 5.4, i.e. synthesized attributes which are well-known in the compiler domain [8, 78]. Declaration of such attributes has the following form:

Table 5.1: Exemplary rules of inference for checking if an empty string, ϵ, belongs to language L(r) defined by a regular expression r.

Regular expression r	ϵ **in** **L**(r)
r_1*	true
$r_1?$	true
r_1+	iff ϵ in L(r_1)
$r_1\{a,b\}$	iff $(a=0)$ or $(\epsilon$ in L$(r_1))$
$r_1\|..\|r_n$	iff there exists $j:\ 1 \le j \le n \bullet \epsilon$ in L(r_j)
$r_1..r_n$	iff for every $j:\ 1 \le j \le n\ \bullet\ \epsilon$ in L(r_j)

```
GenAtriDec = '@' GenAtriName ':' TypeDec ';'
           ;
   TypeDec = 'Boolean'
           | 'Integer'
           | GenAtriVal ( ',' GenAtriVal )*
           ;
```

The name of a general purpose attribute, `GenAtriName`, is a sequence of letters (lower or upper case) and digits. `Boolean` denotes the Boolean type which consists of two values: `true`, and `false`. Such attributes can be manipulated with the `Boolean` operators (they are encoded using the notation of the C language, i.e. '`&&`', '`||`', and '`!`'). `GenAtriVal` stands for a general-purpose attribute value and its lexical structure is the same as `GenAtriName` (obviously, each `GenAtriVal` must be unique). An exemplary declaration of a general-purpose attribute is given below:

```
@Empty: Boolean ;
```

Evaluation of general-purpose attributes, `AttributEval`, mentioned in Sec. 5.3, has the following syntax:

```
AttributEval = GenAtriName '=' Expression
             ;
  Expression = ClasExp '(' Oper ClasExp ')' '*' ( Oper ClasExp )?
             | ClasExp
             ;
```

`ClasExp` denotes a classical expression returning a value that fits the declared attribute type, and `Oper` is a classical operator appropriate for the attribute type (e.g. '+' for the `Integer` type or '`&&`' for the `Boolean` type). Every expression containing the extension expression '(..)*' is evaluated from the leftmost son to the rightmost one (i.e. the first one to the last one). Here is an example of a rule containing evaluation of a general-purpose attribute:

```
 Component= Factor1 Factor2* Factor3
     EN( ): n=1 ? "a sequence composed of "
            n=2 ? "sequences, each composed of "
            Factor1< 1 ", " (Factor2< 1 ", ")* "and " Factor3< 1
     Empty= Factor1.Empty ( && Factor2.Empty )* && Factor3.Empty
           ;
```

It makes the Empty attribute of Component true if and only if the Empty attributes of all the Factors have the value true (to be more precise, only the Factors that are sons of that Component are checked here). When the value of the Empty attribute is available, one can use it to generate the beginning of the sentence in two different forms, depending on the value of Empty:

```
Syntax= Regex
 EN( ):  Regex.Empty ?
         ( "You can leave the field empty or enter " Regex< 1 ". " )
        !Regex.Empty ?
         ( "Enter " Regex< 1 ". " )
       ;
```

As the reader perhaps has already noticed, the main differences between general-purpose attributes and grammatical ones are the following:

- General-purpose attributes are independent of a natural language, while grammatical attributes are strongly connected with a given language.
- The aim of grammatical attributes is to adjust some words into an appropriate form. The general-purpose attributes work at a higher level – they are used to select a sentence structure that would be the best from the understandability point of view.

5.8 Idiomatic patterns

As mentioned in the previous section, the simple paraphrasing of a regular expression in a natural language is in many cases not enough to produce an effective explanation. Some problems concern a special characteristic of a regular expression, such as the possibility of accepting an empty string. As shown in the previous section, those cases can be treated with general-purpose attributes. Another sort of problem is connected with the special structure of a regular expression, which is not appropriate for direct paraphrasing in a natural language. Let us consider the following example:

```
Series = [0-9] ( "," [0-9] )+
```

When directly paraphrased using rules like those presented earlier, it could produce the following explanation:

Series is a sequence consisting of a decimal digit followed by a nonempty sequence consisting of a sequence of comma and a decimal digit.

It is correct but not easy to understand. The explanation given below seems much better:

Series is a sequence of at least two decimal digits separated with commas.

To implement that type of explanation, one can use *idiomatic patterns*, i.e. patterns describing a special structure of a regular expression (or its fragment) that can be better explained with an idiomatic phrase, i.e. a phrase different from the one that can be obtained using the superposition of standard templates. Idiomatic patterns have the following syntax:

```
IdiomPattern= CaseDesc
             ( Lang ':' Template )+
                   ';'
             ;
```

From the presented syntax it follows that an idiomatic pattern, `IdiomPattern`, consists of a case description, `CaseDesc` (which is a counterpart of `Production` in the explanation rule discussed in Sec. 5.3), and a nonempty sequence of templates, one for each natural language, `Lang`, covered by the explanation rules. When a case described by `CaseDesc` is discovered (at the implementation level the parse tree is examined) a `Template` appropriate for a given language is used to replace the standard explanation.

A case description consists of a primary condition and a secondary one:

```
CaseDesc= PrimaryCond '&&' SecondaryCond ;
```

Roughly speaking, the primary condition describes a context in which the secondary condition applies. More precisely, the primary condition is a nonempty sequence of conditions imposed on what can be derived from a given nonterminal symbol. The syntax of a primary condition is given below:

```
PrimaryCond=      Nonterm '=>' ReMex
            ('&&' Nonterm '=>' ReMex )*
            ;
```

`ReMex` is a regular meta-expression describing a sentential form (i.e. a sequence of terminal and nonterminal symbols) that can be directly derived from `Nonterm` (in other words, `ReMex` describes the list of sons of `Nonterm` on the syntax tree):

```
     ReMex= Component ( "|" Component )* ;
 Component= Factor Factor* ;
    Factor= Primary ( "+" | "*" | "?" )? ;
   Primary= Term | Nonterm | "(" MetaRex ")" | "." ;
```

A dot (".") stands for any (single) terminal or nonterminal symbol. The secondary condition has the following syntax:

```
SecondaryCond= Nonterm '==' Nonterm ;
```

It means that a string of terminal symbols derived from the `Nonterm` on the left of the symbol '==' must be equal to the string derived from the `Nonterm` on the right.

Assuming that S denotes the starting symbol of the grammar, r is the analyzed regular expression, α, β, γ, δ, π, σ denote a possibly empty sequence of terminal and nonterminal symbols (π stands for 'prefix', and σ for 'suffix'). Moreover, assume that:

- $=>^*$ denotes reflexive and transitive closure of the derivation relation '=>',
- $=>^+$ denotes transitive closure of the derivation relation '=>',
- α `~ /r/` means that a sequence α matches (from its beginning to the end) a regular expression r.

Then, the semantics of `CaseDesc` of the form

$$
\begin{array}{l}
\quad N_0 \ => \ ReMex_0 \\
\&\& \ N_1 \ => \ ReMex_1 \ \ldots \ \&\& \ N_{i+1} \ => \ ReMex_{i+1} \ \ldots \ \&\& \ N_k \ => \ ReMex_k \\
\&\& \ M_1 \ == \ M_2
\end{array}
$$

can be presented as in Table 5.2, rows 1-4.

Let us consider the following example. Assume that (a fragment of) a regular expression's language is described with the following productions:

```
    Regex= Component ;
Component= Factor1 Factor2* Factor3 ;
Component= Factor ;
   Factor= Primary "+" ;
   Factor= Primary ;
  Primary= "(" Regex ")" ;
```

To cope with cases like the one exemplified by `Series`, one could write the following idiomatic pattern:

```
Component1=> Factor1 Factor2
        && Factor2    => Primary "+"
        && Primary    => "(" Regex ")"
        && Regex      => Component2
        && Component2=> Factor3 Factor4
        && Factor1    == Factor4
     EN( ): "a sequence of at least two " Factor1< 2
            "separated with " Factor3< 2
          ;
```

The above pattern looks for Components (i.e. fragments of a regular expression) that have a special structure, namely $r_1(r_2r_1)+$ (r_1 corresponds to Factor1, and r_2 to Factor3). Then the standard explanation is replaced with a new one of the form:

a sequence of at least two r_1 separated with r_2

(r_1 and r_2 will be adjusted to the plural number).

5.9 Auxiliary diagrams and referring expressions

Some regular expressions are too complex to explain with one syntax diagram and one sentence. Consider the following regular expression:

```
Index= [0-9]+(\ +[0-9]+)+
```

The corresponding diagram is presented in Fig. 5.5a and a 1-sentence explanation would be something like the sentence given below (the idiomatic pattern presented in the previous section has been applied):

Table 5.2: The formal meaning of various forms of case-description expressions. δ is a nonempty sequence of terminal symbols, η is possibly an empty sequence of terminal and nonterminal symbols. *FISH* and *fish* are sets of pairs $[\rho, \nu]$ where ρ is a regular expression and ν is a name. L(r) denotes a language defined by a regular expression r.

No.	Case description	Meaning
1	`N`$_0$ `=> ReMex`$_0$	S $=>^+ \pi$ N$_0$ σ => $\pi\ \delta_0\ \sigma$ $=>^*$ r ^ δ_0 ~ / ReMex$_0$ /
2	`&& N`$_1$ `=> ReMex`$_1$	$\delta_0 = \alpha_1$ N$_1$ β_1 ^ $\pi\ \alpha_1$ N$_1$ $\beta_1\ \sigma$ => $\pi\ \alpha_1\ \delta_1\ \beta_1\ \sigma$ $=>^*$ r ^ δ_1 ~ / ReMex$_1$ /
3	`&& N`$_{i+1}$ `=> ReMex`$_{i+1}$	$\alpha_i\ \delta_i\ \beta_i = \alpha_{i+1}$ N$_{i+1}$ β_{i+1} ^ $\pi\ \alpha_{i+1}$ N$_{i+1}$ $\beta_{i+1}\ \sigma$ => $\pi\ \alpha_{i+1}\delta_{i+1}\beta_{i+1}\ \sigma$ $=>^*$ r ^ δ_{i+1} ~ / ReMex$_{i+1}$ /
4	`&& M`$_1$ `== M`$_2$	$\alpha_k\ \gamma_k\ \beta_k = \gamma_1$ M$_1$ γ_2 M$_2$ γ_3 ^ $\pi\ \gamma_1$ M$_1$ γ_2 M$_2$ $\gamma_3\ \sigma$ $=>^+$ $\pi\ \gamma_1\ \Delta\ \gamma_2\ \Delta\ \gamma_3\ \sigma$ $=>^*$ r
5	`&& M => %FISH .* %fish`	$\alpha_k\ \delta_k\ \beta_k = \gamma_1$ M γ_2 ^ $\pi\ \gamma_1$ M $\gamma_2\ \sigma$ $=>^+$ $\pi\ \gamma_1\ \eta_1\ \varphi_1\ \eta_2\ \varphi_2\ \eta_3\ \gamma_2\ \sigma$ $=>^*$ r ^ $[\rho_1, \nu_1]$ $\underline{\text{in}}$ FISH ^ L(φ_1) = L(ρ_1) ^ $[\rho_2, \nu_2]$ $\underline{\text{in}}$ fish ^ L(φ_2) = L(ρ_2)
6	`&& M < %FISH`	$\alpha_k\ \delta_k\ \beta_k = \gamma_1$ M γ_2 ^ $\pi\ \gamma_1$ M $\gamma_2\ \sigma$ $=>^+$ $\pi\ \gamma_1\ \varphi\ \gamma_2\ \sigma$ $=>^*$ r ^ $[\rho, \nu]$ $\underline{\text{in}}$ FISH ^ L(φ) subset L(ρ)

Index is a sequence of at least two nonempty sequences of decimal digits separated with nonempty sequences of space characters.

It is not very easy to understand, mainly due to nested sequences (the word 'sequence' appears three times). It would be much easier if the syntax description was split into three regular expressions:

```
Number= [0-9]+
   Gap= \ +
 Index= Number ( Gap Number )+
```

The corresponding syntax diagrams are shown in Fig. 5.5b, c, d, and the narrative explanation could be like the one given below:

Assume Number is a nonempty sequence of decimal digits, and Gap is a nonempty sequence of space characters. Then Index is a sequence of at least two Numbers separated with Gaps.

AUXILIARY DIAGRAM

The diagrams of Fig. 5.5b and 5.5c are auxiliary diagrams – they illustrate a part of the regular expression and their aim is to make the whole explanation easier to understand. Names assigned to auxiliary diagrams (Number and Gap) are so-called *referring expressions.* In general, *referring expression* is "*a description of an entity that enables the hearer to identify that entity in a given context*" [126]. One of the possible forms of referring expressions is a proper name [90, 126]. As noticed by Krahmer and Deemter, "*proper names have limited applicability because many domain objects do not have a name that is in common usage*". In the context of syntax explanations there are two options: *a)* a name can be given by a programmer, or *b)* a name can be automatically created by a computer.

REFERRING EXPRESSION

Some languages allow the naming of parts of a regular expression by a programmer (e.g. Lex [93]), and some others do not (e.g. HTML5). Even if a language allows the use of names in regular expressions, it might be useful if a computer could support a programmer and the naming of some parts of a regular expression. The main reason is the difference in the points of view: from the programmer's point of view a given expression can appear quite simple and she/he may not see a need to introduce an auxiliary term, whereas for an inexperienced user it could be difficult to comprehend. Thus, the following problem arises:

PROBLEM 2. How can we identify auxiliary diagrams and name them in an automatic way?

One solution is to create a glossary of popular regular expressions such as *Number* (i.e. a nonempty sequence of digits) or *Name* (a nonempty sequence of letters starting with a capital letter). Whenever a subexpression of the regular expression matches the definition of a name, the regular expression is split into a part defining a given name and the remaining part in which the name appears instead of the subexpression. The definition of a glossary item resembles idiomatic patterns:

```
GlossaryItem= Nonterm '=>+' Term +
             ( Lang    '!'   Boilerplate )+
                       ';'
             ;
```

Here is an example of a glossary item definition:

```
Factor =>+  "[0-9]" "+"
    PL(F)!  "Liczba< n,c "
    EN( )!  "Number< n "
         ;
```

In a similar way one can define *Word*, *Name*, *Gap* etc. It is important to correctly split the right-hand side of the derivation symbol ("=>$^+$") into a sequence of lexemes. In the above example, presenting a right-hand side as "`[0-9]+`" is incorrect because the same sequence with a white space character before '+' would not be recognized (i.e. the string "`[0-9] +`"). Similarly, presenting it as a sequence of "`[`", "`0-9`", "`]`", and "+" would also be incorrect as it could match a single number as a sequence of digits with the space characters inside (i.e. the regular expression "`[0-9 ]+`" would match a set of strings different from "`[0-9]+`").

Unfortunately, in some cases a glossary is not enough. Assume a field *X* is described with the following regular expression:

```
[0-9.]+ \ + [0-9.]+
```

The subexpression "`\ +`" can easily be recognized as Gap. Then the explanation could be presented in the following way:

> *Assume Gap is a nonempty sequence of space characters. X is a sequence of a nonempty sequence of digits or dots, a Gap, and a nonempty sequence of digits or dots.*

The fragment "*a sequence of a nonempty sequence*" can be a little bit confusing for an inexperienced user. An alternative explanation could be the one given below:

> *Assume Extended Number is a nonempty sequence of digits or dots, and Gap is a nonempty sequence of space characters. X is a sequence of an Extended Number, a Gap, and an Extended Number.*

The approach applied here is based on the following assumptions:

- The user knows the *Number* category very well (if there is a doubt, one could start the explanation with an additional statement such as "*Usually number is just a nonempty sequence of digits.*").
- The expression "*extended Z*" is understood by the user as Z plus some additions.

The above description is a good illustration of our approach to identifying auxiliary diagrams (i.e. auxiliary subexpressions) and naming them. If the glossary does not contain an exact match, a 'fuzzy' match will be used instead, which identifies a 'handle'. Handle is a glossary item which fuzzily matches a given subexpression (in the above example the handle was *Number*). Then a new name (i.e. a referring expression) is created with the help of some modifiers such as "extended", "pseudo", or "reduced", depending on the type of fuzzy match ("extended" is used when a given subexpression represents a superset of the language defined by the handle, and "reduced" is used when it is the opposite). Another approach is based on identifying a regular subexpression containing a handle and a character that taken together can form a referring expression (then the name has the form "*Z with Y*"). To allow this, the glossary is split into two parts: big fishes (denoted as `%FISH`) and small fishes (denoted as `%fish`):

- `%FISH` – contains popular regular expressions that (usually) contain a repetition operator such as '+' or '*' (e.g. *Gap, Name, Number, Word* etc.);
- `%fish` – is a set of popular regular expressions that (usually) contain just one character (e.g. *Colon, Dot, Hyphen, Semicolon* etc.).
- Whether a given regular expression belongs to `%FISH` or `%fish` depends on the programmer. The rules of extracting subexpressions and naming them have the following syntax:

```
 Auxiliary= PrimaryCond ( '&&' FinalCond )?
          ( Lang '!' NameAssign )+
            ';'
          ;
 FinalCond= Nonterm '=>' '%FISH' '.*' '%fish'
          | Nonterm '=>' '%fish' '.*' '%FISH'
          | Nonterm ('<' | '>') '%FISH'
          ;
NameAssign= Nonterm '=' ( '%FISH' | '%fish' | Boilerplate ) {1,5}
          ;
```

`PrimaryCond` is described in the previous section. The meaning of a sequence of a `PrimaryCond` and a `FinalCond` is given in Table 5.2, row 1-3, 5-6 (the case

"`Nonterm => %fish .* %FISH`" is very similar to row 5 and "`Nonterm > %FISH`" to row 6).

In `NameAssign` the name is assigned to the same `Nonterm` as in `FinalCond`. The "longest" name consists of two fishes (one big and one small) and three boilerplates: one in the middle and two as the prefix and suffix. Here is an example:

```
Factor =>  Primary ( "+" | "*" )
       &&  Primary   => "(" Regex ")"
       &&  Regex     => Component
       &&  Component => %FISH   .*    %fish
   EN( )!  Component =  %FISH "with " %fish
        ;
```

According to the presented example, whenever a `Component` is part of a repetition ('+' or '*') and contains a big fish (say *Word*) followed by a small fish (say *Ampersand*), then such a `Component` will be classified as auxiliary and given an appropriate name (in our case "*Word with Ampersand*").

5.10 Gordian knots of explanation

In some cases, pure regular expressions lead to unreadable descriptions. Assume, for instance, that an Internet application contains a field in which the end-user is to

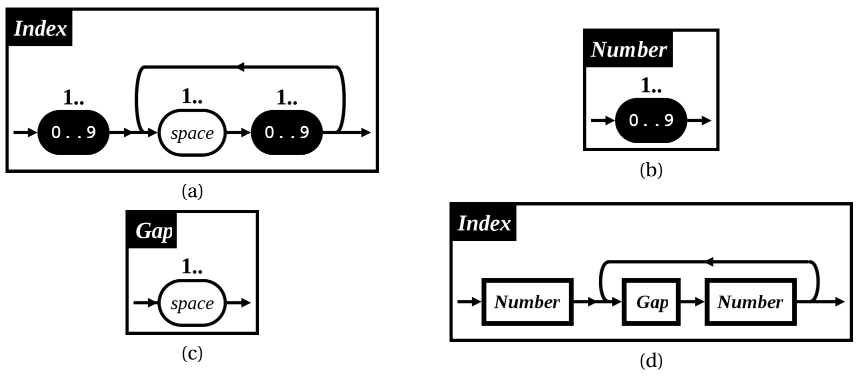


Figure 5.5: Diagrammatic explanation of *Index*

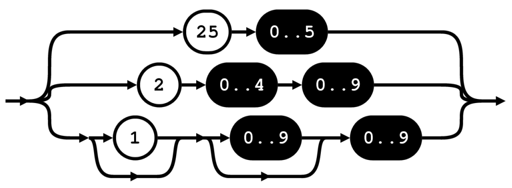


Figure 5.6: Diagrammatic explanation of numbers between 0 and 255

enter an IP address. As we know, IP address consists of four integers, each from the range 0..255. One can describe such a number with the following regular expression

```
25[0-5] | 2[0-4][0-9] | 1?[0-9]?[0-9]
```

and that expression can be visualized as in Figure 5.6. The problem is that the diagrammatic explanation of Figure 5.6 is too complex to be helpful. It would be much easier to figure out what is the acceptable input, if instead of Figure 5.6 the following description was used:

Unfortunately, 0..255 is not a regular expression.

Another possible approach would be based on the idea of approximation: IP address can be described just as a sequence of four integers instead of four bytes (i.e. bytes are approximated to natural numbers) and it would be the application logic (instead of the user interface) that is responsible for checking if all the four integers are within the range 0..255. The advantage of this approach is a very simple regular expression (i.e. [0-9]+) which results in a diagrammatic representation that is easy to understand. Unfortunately, this approach also has an important disadvantage. As was already mentioned, our explanation is three-fold and consists of a diagrammatic representation, a narrative description, and a set of examples (see Figure 5.1). All of them are generated in an automatic way. Thus, it is quite possible that for the regular expression [0-9]+ the system would generate a number greater than 255 (e.g., 347), as an example of a correct input. Obviously, if the end-user entered such a number into a given field, the application logic would reject it and that would be an awkward situation for the end-user (the system rejects an input that is shown in the user manual as a correct one).

Cases like the one described above are difficult and resemble the Gordian knot. We tried some different solutions but none of them were satisfying. So, we decided to apply a solution similar to Alexander's sword, i.e. to create and maintain a 'black list' of regular expressions that are too difficult to explain with the earlier presented approach – we call them Gordian Knots:

```
GordianKnot= Nonterm '=' Regex
           ( Lang    ':' Template
                     '!' Boilerplate
                     '@' '"' AnyCharacter+ '"'
           )+ ;
```

Regex is a regular expression which is difficult to explain, Template provides the text to be used in the narrative explanation, Boilerplate provides the name that will be used for this expression, and a sequence of AnyCharacters defines the text that will be shown in the syntax diagrams. Here is an example:

```
Component= "25"[0-5] | "2"[0-4][0-9] | "1"?[0-9]?[0-9]
    EN( ): n==1 "a number from the range 0..255 "
           n==2 "numbers from the range 0..255 "
         ! "Byte<n "
         @ "0..255"
         ;
```

5.11 Generation of examples

As declared in Section 5.2, Assumption 1, each field explanation should be enriched with a set of exemplary input strings. Such a set should contain correct and incorrect inputs. Additionally, each incorrect string should be complete with a short description explaining why it is considered erroneous (see Fig. 5.1). Thus, the following problem arises:

PROBLEM 3. How to generate examples of incorrect input strings to be able to explain what is wrong in the given incorrect input?

One possible solution is to find a regular expression r_c that is a complement of the original regular expression r, i.e. $L(r_c) = s : s \notin L(r)$. A standard approach for converting r into r_c is to transform r into a nondeterministic finite automaton (NFA), then convert the NFA into a deterministic one (DFA), and once provided with the DFA create r_c. Knowing r_c one can generate "erroneous" strings in the same way as in the case of the original regular expression r. The disadvantage of that approach is the difficulty in explaining why a given "erroneous" string is wrong. For example, assume that the original expression is (aa | bb)*. Then its complement can be expressed as [140]:

```
a(aa)*  |  b(bb)*  |  a(aa)*b(a|b)*  |  b(bb)*a(a|b)*
```

and one of the erroneous strings is aaabaa. It is easy to check that the string does not belong to $L(r)$ [1], but it is difficult to explain what is wrong in this string. Notice that, in fact, a full complement of the original regular expression is not necessary—a subset would do just as well. What we must provide is the ability to clearly say what is wrong in a given erroneous string.

[1]One can use, for instance, Hovland's algorithm [58] or a web browser complying with HTML5.

Our idea is to use mutation operators that “destroy” the original regular expression in a “controlled” way. The resulting regular expression describes only a subset of erroneous strings, but what we get is the ease of explaining what is wrong. Two types of mutations are proposed:

- **Removal**, i.e. omitting one of the factors of concatenation. An example is replacing $r_1 r_2 r_3$ with $r_1 r_3$.
- **Contamination**, i.e. inserting a contamination into a sequence of concatenated regular expressions, e.g. replacing $r_1 r_2$ with $e r_1 r_2$ or $r_1 e r_2$, where e is an erroneous character playing the role of contamination.

For the sake of explainability, only regular expressions with one defect are taken under consideration (i.e. either one contaminant is injected or one factor of concatenation is removed). Nevertheless, one original regular expression can result in a set of “damaged” regular expressions. Each of them is used to generate a few erroneous strings (they should not be long, so that they remain understandable). Each erroneous string is checked as to whether it satisfies the following conditions:

- *Is it really outside of the language described by the original regular expression?* It can happen that the damage caused by contamination or removal is ineffective. Let us consider the following regular expression:

  ```
  [0-9]+\.@+   |   [0-1]@[a-z]*
  ```

  The component `[0-9]+\.@+` can be damaged by removing the period. This results in the component `[0-9]+@+` and one possible erroneous string is `1@`. But this string belongs to the language described by the second component, i.e. `[0-1]@[a-z]*`. Thus, the string `1@` is in fact correct and the damage (in this case, removal) was ineffective.

- *Is it a new erroneous string that has not been previously discovered?* To illustrate the problem, let us assume that the original regular expression has the following form:

  ```
  [0-9]+\.@+   |   [0-1]@[a-z]+
  ```

  If each component is used to generate erroneous strings separately, then after removing the period from the first component, the result is `[0-9]+@+` and the string `1@` can be generated as an erroneous one. But the same string could be generated if the factor `[a-z]+` was removed in the second component.

Thus, duplicates are possible and they must be eliminated from the final set of examples (the same is true for positive examples).

Checking the above conditions is trivial, but designing the mutation operators is not. Each of them is described in more detail below.

5.11.1 Removal

As mentioned earlier, only one factor of concatenation is removed at a time, for the sake of explainability. Assume that each factor of a concatenation is either plain or red (to make the paper readable for colorblind people, red factors are also underlined). A regular expression with red factors will be called "painted". Painted regular expressions are used in the following way:

When an erroneous string is generated, red factors are omitted but they are used to explain what is missing.

Let us consider the following painted regular expression:

```
[0-9]+ \. [0-9]+
```

Using that expression, one could generate the string "`.01`" and the following comment:

A nonempty sequence of digits is missing in the beginning.

Templates for comments such as above can be described using the notation introduced earlier:

```
Component= Factor1 Factor2* Factor3
    UK( ): Factor1.Red ? Factor1<1 "is missing in the beginning. "
           Factor3.Red ? Factor3<1 "is missing in the end. "
          ;
Component= Factor1* Factor2 Factor3 Factor4+
    UK( ): Factor3.Red ? "After " Factor2<1 ", " Factor3<1 "is missing. "
          ;
```

Generating all painted regular expressions for a given component is trivial. For instance, the component

```
[0-9]+\.[0-9]+
```

consists of three factors: `[0-9]+`, `\.`, and `[0-9]+`. Thus, the regular expression above would generate the following set of painted regular expressions:

```
[0-9]+ \. [0-9]+
[0-9]+ \. [0-9]+
[0-9]+ \. [0-9]+
```

If a given factor can lead to an empty string (see the rules presented in Sec. 5.7), it makes no sense to paint it, because omitting that factor will not result in an erroneous string.

5.11.2 Contamination

For the sake of simplicity, assume regular expressions consist only of the five operations: alternative ($r_1|..|r_n$), concatenation ($r_1..r_n$), non-zero repetition ($r+$), any repetition, also called Kleene's star ($r*$), and option ($r?$). Moreover, we allow brackets and sets of characters ($[c_1c_2..c_n]$).

Not every character can be an effective contamination. Let us consider once again the regular expression `[0-9]+\.[0-9]+`. If a period is inserted at its end, the resulting regular expression (i.e. `[0-9]+\.[0-9]+\.`) will produce strings outside of the original language. But if in the same place a digit is inserted instead of a period, the resulting regular expression (i.e. `[0-9]+\.[0-9]+[0-9]`) will describe the same language ($r+r = r+$ for any r). This is because some regular expression like $r+$ or $r*$ can "swallow" some characters that are placed just before or after them. A set of characters that can be swallowed by a regular expression or subexpression will be called its *menu*. Some regular expressions swallow one set of contaminating characters inserted in front of them, and another set of characters inserted after them. Consider the following regular expression:

```
[A] ( [0-9]+ [a-z]+ )+ [B]
```

The subexpression `([0-9]+[a-z]+)+` will swallow any digit placed after `[A]`, but not a letter, and it will swallow any letter placed before `[B]`, but not a digit. Thus, each swallower has its *left* and *right menu*. Obviously, it can happen that both of them are the same. We are using the rules of computing left and right menus presented in Table 5.3. These rules make use of the attribute Σ, which describes the alphabet of a given subexpression (the rules of computing the alphabet of a subexpression are given in Table 5.4).

Left and right menus are not enough. Let us consider another exemplary regular expression:

```
[a-f]+ ( [0-9]+ [a-z]+ )+
```

What contaminating character can be placed just after the opening bracket? Taking into account only menus (in this example only the left menu of `[0-9]+[a-z]+` matters), one could answer that it can be any character other than a digit. Obviously it is not true: if a letter `[a-f]` was placed after the opening bracket, it would be swallowed by `[a-f]+` and the contamination would be ineffective. Therefore, aside from left and right menus, one should also consider the left and right *context* of a given subexpression. To compute left and right contexts, we propose using the rules presented in Table 5.5. It is assumed that for the root regular expression, its left and right contexts are empty sets of characters. After taking into account menus and contexts, it will be clear that any character from the set `[g-z]` is a good candidate for contaminating the above regular expression just after the opening bracket.

For a component of a regular expression $c = f_1..f_j..f_n$ there are n+1 positions where a contaminating character σ can be placed:

$$c = \sigma_0 f_1 \sigma_1..\sigma_{j-1} f_j \sigma_j..\sigma_{n-1} f_n \sigma_n$$

We have assumed that each contaminating character σ_j must fulfill the following constraints:

- $\sigma_0 \not\subset$ c.LC $\wedge$ $\sigma_0 \not\subset$ f$_1$.LM

Table 5.3: Computing left menu (LM) and right menu (RM).

Regular expression	**Left and right menu**
$r = c_1\|..\|c_j\|..\|c_n$	r.LM = $\bigcup_j$ c$_j$.LM; r.RM = $\bigcup_j$ c$_j$.RM;
$c = f_1..f_j..f_n$	c.LM = f$_1$.LM $\wedge$ c.RM = f$_n$.RM
$f = p+$	f.LM = f.RM = p.Σ
$f = p*$	f.LM = f.RM = p.Σ
$f = p?$	f.LM = p.LM $\wedge$ f.RM = p.RM
$p = (r)$	p.LM = r.LM $\wedge$ p.RM = r.RM
$p = [Chars]$	p.LM = p.RM = $\emptyset$

Table 5.4: Computing the alphabet Σ of a regular subexpression.

Regular expression r	**Summation**
$r = c_1\|..\|c_j\|..\|c_n$	r.Σ = $\bigcup_{1 \le j \le n}$ c$_j$.Σ
$c = f_1..f_j..f_n$	c.Σ = $\bigcup_{1 \le j \le n}$ f$_j$.Σ
$f = p+$	f.Σ = p.Σ
$f = p*$	f.Σ = p.Σ
$f = p?$	f.Σ = p.Σ
$p = (r)$	p.Σ = r.Σ
$p =$ [Chars]	p.Σ = {Chars}

- $\underset{1 \le j < n}{\forall} \sigma \not\subset \mathrm{f}_j.\mathrm{RM} \land \sigma_j \not\subset \mathrm{f}_{j+1}.\mathrm{LM}$
- $\sigma_n \not\subset \mathrm{c.RC} \land \sigma_n \not\subset \mathrm{f}_n.\mathrm{RM}$

To make generation of erroneous examples easier, we use the set of transformation rules presented in Table 5.6. Using them provides a set of simple components for a given regular expression (a simple component is a concatenation of sets of characters). For instance, for the regular expression

```
[a-f]+ ( [0-9]+ [a-z]+ )+
```

one would get a set of simple components with contaminating characters containing the following (the set of contaminating characters is given in red and it is underlined):

```
[a-f][a-f][g-z][0-9][0-9][a-z][a-z][0-9][0-9][a-z][a-z]
```

To generate an erroneous example, one can randomly choose any character from each set being part of a simple component (for the above example it could be `aag00aa00aa`). Knowing the contaminating character and its position, one can generate the following comment to an erroneous example of this sort:

> *This string is wrong because the character 'g' is superfluous. If it is removed, the string will be correct.*

5.12 Experimental evaluation

It is important to investigate whether the generated explanations are easy to understand by end-users of IT systems.

Table 5.5: Computing left context (LC) and right one (RC).

Regular expression r	**Left and right context**
$r = c_1\vert..\vert c_j\vert..\vert c_n$	$\underset{1 \le j \le n}{\forall} \mathrm{c}_j.\mathrm{LC} = \mathrm{r.LC};$
	$\underset{1 \le j \le n}{\forall} \mathrm{c}_j.\mathrm{RC} = \mathrm{r.RC};$
$c = f_1..f_j..f_n$	$\mathrm{f}_1.\mathrm{LC} = \mathrm{c.LC}; \mathrm{f}_n.\mathrm{RC} = \mathrm{c.RC};$
	$\underset{1 \le j \le n}{\forall} \mathrm{f}_j.\mathrm{LC} = \mathrm{f}_{j-1}.\mathrm{RM};$
	$\underset{1 \le j \le n}{\forall} \mathrm{f}_j.\mathrm{RC} = \mathrm{f}_{j+1}.\mathrm{LM};$
$f = p+$	$\mathrm{p.LC} = \mathrm{f.LC} \land \mathrm{p.RC} = \mathrm{f.RC}$
$f = p*$	$\mathrm{p.LC} = \mathrm{f.LC} \land \mathrm{p.RC} = \mathrm{f.RC}$
$f = p?$	$\mathrm{p.LC} = \mathrm{f.LC} \land \mathrm{p.RC} = \mathrm{f.RC}$
$p = (r)$	$\mathrm{r.LC} = \mathrm{p.LC} \land \mathrm{r.RC} = \mathrm{p.RC}$
$p = [Chars]$	–

In order to explore this issue, we decided to conduct a controlled experiment, the goal of which was to investigate whether generated fields explanations are at least as easy to understand as explanations prepared by people.

5.12.1 Experiment design

The *independent variable* considered in the experiment was the approach to preparing explanations of fields, with two considered treatments: generating explanations with the use of the prototype tool or preparing them by people. The *dependent variable* was the *understandability* of the explanations. We assumed that the user understands the explanation, if after reading the explanation, she/he is able to verify the correctness of field inputs.

Objects of the experiment

In order to increase the realism of the study, we decided to select some real-life examples of field definitions as objects of the experiment:

- Bank identifier — ISO/IEC 7812 Identification cards – Identification of issuers;
- Internet identifier — RFC 5322: Internet Message Format;
- Transmission identifier — IPv4 identifier RFC 791;
- HTTP identifier — RFC 3986: Uniform Resource Identifier (URI);
- Publication identifier — International Standard Book Number (ISBN).

At this stage we identified an important confounding factor, which was the quality of the explanations provided by people (in terms of completeness and linguistic correctness). In order to control this factor and increase the realism of the study, we decided to ask fifteen 4th year students of Software Engineering at Poznan University

Table 5.6: Transforming regular expressions into simple components.

Regular expression r	**Transformation T(r)**
$r = c_1 \vert .. \vert c_j \vert .. \vert c_n$	T(r) = T(c_j) (choose any)
$c = f_1..f_j..f_n$	T(c) = T(f_1) .. T(f_j) .. T(f_n)
$f = p+$	T(f) = T(p) T(p)
$f = p*$	T(f) = T(p)
$f = p?$	T(f) = T(p)
$p = (r)$	T(p) = T(r)
$p = [Chars]$	T(p) = [Chars]

of Technology to provide the field explanations. As a next step, we reviewed the explanations and combined the most representative ones into two sets that were used in the experiment: the best explanations (S1) and the average-quality explanations (S2). The third set of field explanations used in the experiment contained the ones generated by the prototype tool (S3).

Participants

In total, there were 207 participants in the experiment: P1 — 81 1st-year students of Logistics, P2 — 67 1st-year students of Security Engineering, P3 — 49 3rd-year students of Security Engineering, P4 — 10 2nd-year students of Electrocardiology.

Among each homogenous group of participants P1 to P4, participants were randomly assigned to three groups: group *G1*, which was asked to assess the validity of the exemplary input data based on the explanations from the set S1; group *G2*, which was asked to use the set S2; and group *G3*, which was asked to use the set S3.

5.12.2 Operation of the experiment

The experiment was executed at the Poznan University of Technology in four independent sessions.

Prepared instrumentation

Each participant in the experiment was provided with the handouts that contained the description of the field-explanation notation; three examples of exemplary input data as a warm-up task; and the description of the experiment tasks.

The description of a task consisted of the name of a field (e.g., bank identifier), its explanation and the set of seven pieces of exemplary input data to be validated by the particpants.

Execution

Each experimental session began with a 10-minute presentation explaining the goal of the experiment to the participants.

During the execution of the experiment participants were asked to verify the correctness of the provided exemplary input data (to classify them either as correct or wrong). The whole process was supervised by a researcher who was not allowed to provide any hints related to the solutions to the tasks. The participants had 80 minutes to acquaint themselves with the provided materials and solve the tasks.

Data validation

After collecting the answer forms, we checked their completeness to find out whether all of them had been filled in correctly.

5.12.3 Analysis and interpretation

Descriptive statistic

In the following step, the experiment data was analyzed in order to find and handle any potentially outlying observations. Figure 5.7 presents the distributions of the total number of correctly classified exemplary data by the participants belonging to each of the groups. In addition, the descriptive statistics are presented in Table 5.7. We identified two potentially outlying observations, which were investigated separately. However, in the course of the analysis we did not find any evidence that would allow us to reject them from the analysis.

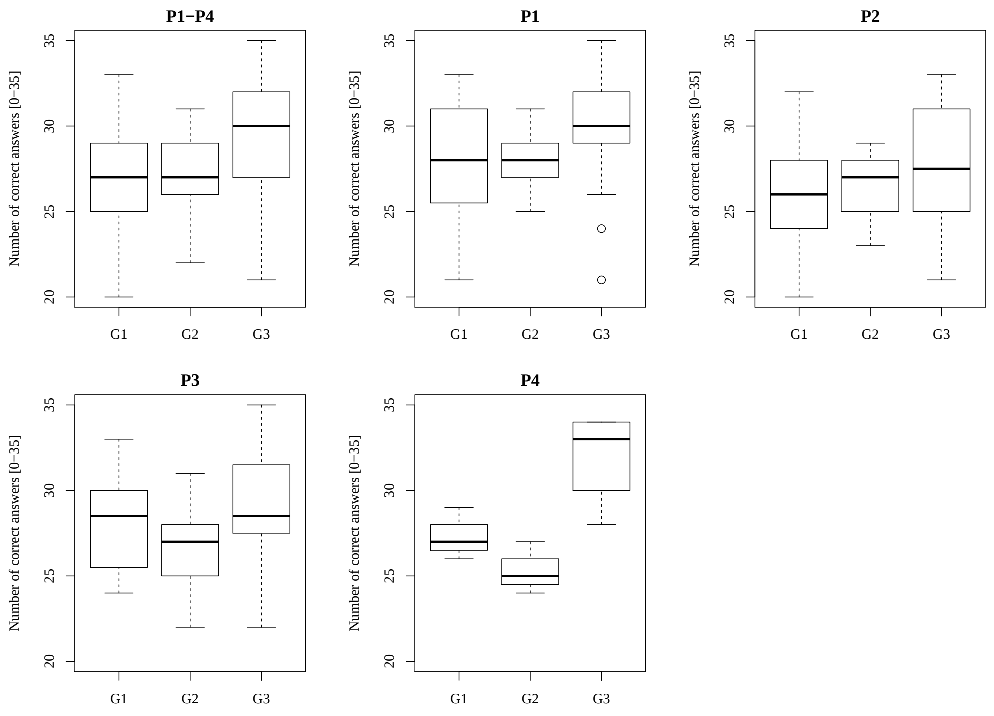


Figure 5.7: Distributions of the number of correctly classified examples (*P1–P4: all participants; P1: Logistics; P2 and P3: Security Engineering; P4: Electrocardiology*).

In order to select an appropriate statistical test, we investigated whether the samples come from normally distributed populations. After analyzing the Q-Q plots we suspected that the assumption about samples normality might be violated. However

Table 5.7: Descriptive statistics (*P1–P4: all participants; P1: Logistics; P2 and P3: Security Engineering; P4: Electrocardiology*).

	P1–P4			*P1*			*P2*			*P3*			*P4*		
	G1	G2	G3	G1	G2	G3	G1	G2	G3	G1	G2	G3	G1	G2	G3
N	69.0	71.0	67.0	28.0	28.0	25.0	22.0	23.0	22.0	16.0	17.0	16.0	3.0	3.0	4.0
Min	20.0	22.0	21.0	21.0	25.0	21.0	20.0	23.0	21.0	24.0	22.0	22.0	26.0	24.0	28.0
1st Qu.	25.0	26.0	27.0	25.8	27.0	29.0	24.0	25.0	25.3	25.8	25.0	27.8	26.5	24.5	31.0
Median	27.0	27.0	30.0	28.0	28.0	30.0	26.0	27.0	27.5	28.5	27.0	28.5	27.0	25.0	33.0
Mean	27.4	27.1	29.4	28.0	28.2	30.2	25.8	26.5	27.9	28.1	26.7	29.4	27.3	25.3	32.0
3rd Qu.	29.0	29.0	32.0	31.0	29.0	32.0	28.0	28.0	31.0	30.0	28.0	31.3	28.0	26.0	34.0
Max	33.0	31.0	35.0	33.0	31.0	35.0	32.0	29.0	33.0	33.0	31.0	35.0	29.0	27.0	34.0

this suspicion was not confirmed by the Shapiro-Wilk test [136] with the assumed significance level 0.05[2]. Nevertheless, we decided to use non-parametric statistical tests.

Hypotheses testing

In order to compare the levels of understandability of generated field explanations with the best and average field explanations prepared by people, the following hypotheses were formulated. The null hypothesis stated that there was no difference with respect to the total number of correctly classified examples of fields input data (all the participants were considered, i.e., P1–P4):

$$H_0 : \theta_{G1} = \theta_{G2} = \theta_{G3} \tag{5.1}$$

The alternative hypothesis was that the median number of correctly classified exemplary input data was not equal in the case of at least two groups:

$$H_1 : \text{Not } H_0 \tag{5.2}$$

The observed normalized effect size[3], expressed as Cohen's d coefficient [30], was between "medium and "large"[4] for groups G1–G3, G2–G3 (the observed values of d were 0.60 and 0.78). The observed effect size between groups G1 and G2 was very "small" ($d = 0.08$).

To test the hypotheses we used the Kruskal-Wallis test [91], which is a non-parametric version of the well-known ANOVA test. The significance level α was

[2]G1: W = 0.9707, p-value = 0.1046; G2: W = 0.966, p-value = 0.05143; W = 0.9653, p-value = 0.05847.

[3]Please note that the retrospectively calculated effect size only approximates the to real effect size in the populations from which the samples were drawn.

[4]According to Cohen [30] effect size is perceived as "small" if the value of d is equal to 0.2, as "medium" if the value of d is equal to 0.5, and as "large" if d is equal to 0.8.

set to 0.05. The obtained value of the χ^2 statistics was equal to 20.495 (df=2). This allowed us to reject the null hypothesis with a significance level less than the assumed 0.05 (p-value = 3.545×10^{-5}). Therefore, we concluded that there is a significant difference between the median numbers of correctly classified example data for at least two groups.

As the next step of the analysis, we performed the post-hoc pairwise comparison of subgroups according to the procedure proposed by Conover [32]. For each pair of samples, two values were calculated: the difference between mean ranks, and the critical difference of the mean ranks. If the difference between the mean ranks is greater than the calculated critical value, the difference is indicated as significant. According to the results of the analysis, presented in Table 5.8, the differences between the median number of correctly classified input data examples seemed to be significant for the comparisons between groups G1–G3 and G2–G3.

Table 5.8: The post-hoc analysis [32] of Kruskal-Wallis test (in each cell, row vs. column: difference of the mean ranks; the critical difference of the mean ranks — in brackets; * denotes a statistically significant difference).

	G1	G2	G3
G1			
G2	-5.75 (19.04)		
G3	36.85* (19.32)	42.60* (19.18)	

Interpretation

The goal of the experiment was to investigate whether the prototype tool was able to generate explanations of fields that are at least as easy to understand as those prepared by people. The results of the experiment indicated that the generated explanations helped participants (people with limited IT knowledge) to more accurately assess the validity of exemplary data. In fact, this is the result one may expect; taking into account the fact that field explanations generated by the prototype tool contain additional information, such as syntax diagrams and examples. However, it shows that the three-part explanation of fields is usable and does not overwhelm the reader with information.

5.12.4 Threats to validity

In order to correctly draw conclusions from the results of the experiment, some threats to validity need to be discussed. We classified them into four groups: construct validity, internal validity, external validity and conclusion validity.

Construct validity

Construct validity discusses, for instance, whether the selected metrics really measure the phenomena being investigated in the study.

In the experiment, the main threat to construct validity relates to the metric that was used to measure the understandability of field explanations. We believe that a user has to understand the explanation of the field in order to assess its correctness. However, we are not able to prove that the metric measures all of the aspects of understandability.

Internal validity

Internal validity refers to any factors (other than the independent variable) that could affect the dependent variable and were not controlled by researchers. One of such factors could be the experience of the participants. In the experiment we had different groups of participants (students of different majors). We assumed that the level of knowledge related to IT systems could differ between the groups, therefore we decided to perform a stratified assignment—to randomly assign participants to treatments within each of the groups individually (P1 to P4), instead of doing it once for the whole group of participants.

The second threat relates to the types of fields used in the experiment. In order to make the results of the experiment more realistic, we decided to include real-life examples of fields as objects of the study. However, we cannot be sure that some of the participants did not rely on their experience rather than on the provided explanations while performing the tasks. In order to mitigate this problem we decided to name the fields in such a way that participants were not sure if the patterns actually represented what they may suspect. For instance, a field named *bank identifier* was in fact representing a *credit card number*. The impact of the potential problem was also reduced by the random assignment of the participants to the treatments.

The third problem relates to the quality of field explanations prepared by people. We wanted to avoid preparing the explanation ourselves, because it could constitute a serious threat to the validity of the experiment. To mitigate the problem we asked M.Sc. Software Engineering students to prepare the descriptions.

External validity

External validity concerns all the issues that could affect the possibility of generalizing the results of the study to the wider population.

In the conducted experiment, the main threat to external validity relates to the selection of participants. Although the participants were potential users of IT systems, they did not represent all the possible classes of end-users. However, we believe that the results could be generalized to the population of typical users of IT systems with further or higher education.

The second threat relates to how well the fields used in the experiment correspond to real-life cases. If we had prepared artificial examples of fields we would have potentially limited the possibility of generalizing the results. Therefore, we decided to only include real-life patterns that were formally defined by third parties (e.g., ISO/IEC standards, RFCs).

Conclusion validity

The threats to conclusion validity comprise all problems related to the proper usage of methods and the tools used to draw conclusions from the data.

In the experiment we decided to use a non-parametric test because we suspected that the assumption about the normally distributed population could be violated.

The second threat relates to the power of the tests. The sample size of 207 participants (67 to 71 per treatment) was sufficient to obtain valid results of statistical tests.

5.13 Related work

The correctness of data entered into web forms can be checked in a number of ways. A programmer can write a function which takes a string as a parameter and analyze its content. However, writing a function for each data type can be a time consuming task and it is not cost-effective—this is why web frameworks which support data validation are often used (e.g. in Django, one can create a form which generates an HTML code and also supports a data check [54]). However, frameworks are often general purpose and do not support all types of fields. In such cases, one can use *topes* [133] or regular expressions [48]. A *tope* is a user-defined abstraction of a string that allows one to determine whether a text is valid, invalid or questionable (it is possibly valid, but double-checking is recommended). This approach can be useful to end-user programmers but may not match up to professional programmers' expectations, e.g. acceptance of a questionable string that is incorrect may cause inconsistency in data. Regular expressions may be harder to create and maintain, but they are free from this disadvantage. There are also a number of tools that support work with regular expressions (like GraphRex [35] and RegViz [17]], including tools that use visual programming language to facilitate work [47]. Regular expressions are also more widespread than topes. We decided to focus on regular expressions.

Different computer languages use various versions of regular expressions. It was decided to use a version from Lex [93], since it is supported by most languages (like Java, Python and Perl).

To explain a regular expression to end-users one can use YAPE [144]. For example, `[A-Z]{2}([0-9]X)?` is explained in the following way:

```
NODE       EXPLANATION
----------------------------------------------------------------------
(?-imsx:   group, but do not capture (case-sensitive)
           (with ^ and $ matching normally) (with . not
           matching \n) (matching whitespace and #
           normally):
----------------------------------------------------------------------
  [A-Z]{2}   any character of: 'A' to 'Z' (2 times)
----------------------------------------------------------------------
  (          group and capture to \1 (optional
             (matching the most amount possible)):
----------------------------------------------------------------------
    [0-9]      any character of: '0' to '9'
----------------------------------------------------------------------
    X          'X'
----------------------------------------------------------------------
  )?         end of \1 (NOTE: because you are using a
             quantifier on this capture, only the LAST
             repetition of the captured pattern will be
             stored in \1)
----------------------------------------------------------------------
)          end of grouping
----------------------------------------------------------------------
```

Such an explanation is precise, but uses technical vocabulary (e.g. *group*, *quantifier*) and (e.g. `?-imsx:`, `\1`)—this may be helpful to programmers, but hard to understand for non-specialists.

A more user-friendly approach was proposed by Ranta [123]. In the paper, one can read the following explanation of a vowel: *A vowel is a symbol from the list 'a', 'e', 'i', 'o', 'u', 'y'.* [123] (regular expression: `[aeiouy]`), which looks more reader-friendly than the output from YAPE. Unfortunately, as the author stated, a complex input can result in a hard to read explanation and in order to improve understandability one needs to manually divide the expression into smaller parts and manually assign names to them—automatic division and assignment of names is not supported. Another disadvantage concerns the form of presentation, it is limited to text only.

One could try to combine Ranta's approach with a graphical representation of regular expressions, which can be obtained (for example) from tools like GraphRex [35] or RegViz [17]. Unfortunately, both of them are dedicated to professionals and showing their output to end-users is questionable. RegViz[5] is dedicated to

[5]http://regviz.org/

debugging regular expressions, thus their presentation is enriched with information which may be confusing for end-users. GraphRex is a plugin for the Eclipse integrated development environment. It uses a limited number of elements to visualize a regular expression and, as a consequence, different parts of a regular expression can be presented in a similar way (e.g. allowed characters and their regularity of occurrence are drawn a in similar way)—this may be problematic, not only for end-users, but also for inexperienced programmers.

A different approach is used in RegExpert [26]. This tool takes a regular expression, transforms it into a nondeterministic finite automata with epsilon transitions (NFA-epsilon), and generates a state diagram. On the diagram, characters that can be typed are drawn on arrows which connect states (each state has a label, a letter *q* with a number), while repetition and optionality are obtained with additional arrows. According to Budiselic et al., these tools *make the learning process entertaining and simple for students*; unfortunately, in the case of people who are IT-laymen, it may be too complicated. There is one additional problem, generated diagrams are space consuming and using them may significantly increase the size of user documentation.

A different approach to visualisation was proposed by Blackwell [20, 21]. He compared four notations that describe regular expressions: a regular expression itself, a text, *a declarative graphic* (graphical notation without arrows that connects elements, items are drawn from left to right) and *an ordered evaluation graphic* (graphical notation with arrows that connect elements, items are drawn from top to bottom). His studies show that the last form results in the lowest number of misunderstandings. However, his proposal has two disadvantages. Firstly, the proposed notation is space consuming and a complicated regular expression may be hard to fit in a user manual (this fact was also noticed by the author). Secondly, the proposed notation can be misleading. During our preliminary studies we noticed that redundant information on diagrams should be used with caution. For example, Blackwell presented "*any one of* {3,5,6}" as an explanation of `[356]` regular expressions; since only part *any one of* is in a different font, some users may treat brackets and a comma as valid characters.

Erging and Gopinath proposed an explanation which consists of three elements: a regular expression, a structure (identification of commonalities among subexpressions, e.g. the regular expression `[oO][nN]` describes the word *on* written with letters of any case) and a format (detection of separators, e.g. dashes in a date) [42]. These elements form an explanation for programmers who would like to understand and reuse regular expressions in an easy and fast way, though non-professionals may not find the required explanation. This paper also mentions the decomposition of regular expressions, and the division into subexpressions and assignment of names is done here manually.

5.14 Conclusions

The goal of this work was to investigate whether it is possible to automatically generate a 3-fold explanation of form field syntax that would be no worse than a man-made one.

It has been assumed that the 3-fold explanation should consist of three parts: narrative explanation, syntax diagram, and a set of correct and erroneous examples of input text. The generation of the 3-fold explanation is driven by rules written in a specially designed domain-specific language (DSL), which resembles a compiler's syntax-directed definition. The proposed DSL allows extracting subexpressions from a complex regular expression. An easy-to-memorize name can be automatically assigned to each subexpression. Moreover, conditional fragments, detection of patterns and repetition in regular expressions (like allowance of empty string), and approximation of a regular expression are also supported. The resultant description can be multilingual and can be customized, e.g. a linguist can add new descriptions or add rules for a new language.

Each field explanation is enriched with a set of correct and erroneous strings. To generate erroneous strings, a part of a regular expression can be removed or a regular expression can be contaminated in a "controlled" way—using a heuristic algorithm.

To investigate whether generated explanations are no worse then explanations prepared by people, a prototype tool was created and a controlled experiment was conducted. Five regular expressions were selected and a group of 15 Software Engineering students were asked to prepare explanations for them. The results indicate that the 3-fold explanation helps users with limited IT knowledge assess the validity of strings that can be entered into a field. The empirical data have been obtained from the evaluation of fields by 207 participants. In the experiment, 84% of correct answers have been obtained for generated field explanations and less than 79% for man-made descriptions.

Acknowledgements

This work has been supported by the Polish National Science Centre based on the decisions DEC-2011/03/N/ST6/03016.

Chapter 6

Compiling software artifacts to generate user manuals

Preface

This chapter contains the report: *Bartosz Alchimowicz, Jerzy Nawrocki, Sylwia Kopczyńska, Reusing software artifacts for the automatic generation of user manuals, Politechnika Poznańska, RA-13/2014.*

My contribution to this work included the following tasks: *1)* co-design and implementation of generation methods; *2)* conducting the exploratory studies and the experimental evaluation; *3)* analysis and interpretation of the results of the experiments.

Context: According to Novick and Ward, end-users are often dissatisfied with the quality of user manuals. This can lead to financial loss. For example, a user who tries to solve a problem by asking the vendor for help is absorbed by issues with no business value, instead of performing assigned tasks. Situations like this increase support costs, reducing the vendor's income as well. A high quality user manual would be beneficial here. Unfortunately, creating such a document is an expensive and time-consuming task.

Objective: The aim of this work is to check whether it is possible to automatically generate a user manual that would be no worse than a corresponding handmade manual.

Method: To generate a user manual automatically, one needs input data. An analysis of project documentation showed that a business case, a software requirement specification (SRS), and acceptance tests are often used in a software development project. It was assumed that SRS includes functional requirements, non-functional requirements, and technical constraints, and that functional requirements are defined using use cases.

A literature review and an analysis of manuals was carried out to determine the content of a generated user manual. Two variants of user manuals were proposed: naive and complete. The former can be created using only existing information, while the latter requires generating new content.

Two methods of generating additional content were proposed, the first being to add requirements concerning the operating environment and the second to generate examples of interaction between a user and a system, exemplary usages for short. The generation of exemplary usages is based on acceptance tests. Two metrics were proposed to select the most representative test cases: “widget coverage” to select a test which covers the highest number of web elements, and “event coverage” to select tests which cover the highest number of events in use cases. Each exemplary usage is enriched with an explanation and screenshots of a working application (or a GUI mock-up).

To discover potential flaws in generated materials, two exploratory studies were conducted. Next, a manual for a commercial system *Plagiat.pl* was generated and then evaluated with the COCA quality model and the Documentation Evaluation Test. For that purpose, the business case and the software requirements specification was re-engineered based on the application and the original user manual. The generated user manual was in the English language, but the participants were fluent in Polish only—thus, the generated manual was translated into Polish. Both exploratory studies included brainstorming sessions with 11 participants, while the controlled experiment included 3 experts and 16 students.

Results: The user manual generated for the *Plagiat.pl* system is no worse than the corresponding commercial manual written by humans, with respect to all the 6 COCA quality criteria. In the case of the DET method, the number of correct answers was 85% for the generated manual and 83% for the original one.

Conclusion: It seems that the automatic generation of a user manual of no worse quality than a corresponding manual written by humans is possible. The methods proposed in this work can be used to create a tool which can automatically generate a user manual for web applications on the basis of a business case, a software requirements specification (which includes use cases), acceptance tests, and working software (or GUI mock-ups).

6.1 Introduction

Writing a high quality user manual is an expensive and time-consuming task [128, 145]. According to Sun Technical Publications, 3-5 hours are needed to write one page of a manual [145]. However, due to the common tendency to minimize costs, it may be hard to devote so much time solely to creating a manual, and this may affect the quality of documents. Furthermore, this task distracts from other activities [128]. For example, in some companies, user manuals are written by a development team [67], i.e. people who often would prefer to be writing code or performing tests, rather than writing texts which describe software. Thus, it is not surprising that such a task is often considered as undesirable and given little attention. Problems with writing user manuals are noticeable even in projects which use agile software development. Since such approaches often assume changes in the already created code, a development team may also have to modify documentation multiple times (to reflect changes). Unfortunately, frequent changes to an unreleased document can give the impression that this work is unnecessary, even if a client is paying for it.

User manuals can be created by technical writers. Unfortunately, hiring new employees does not mean that they can do this task themselves. Technical writers still need to be in contact with a development team and occupy their time, since they are the best source of information (e.g. without support from a development team it may happen that technical writers are not aware of how a given piece of software works, whether a new feature has been added, etc.)

The presented issues can lead to a low quality user manual. For example, a document can lack an explanation of available functionality, screenshots can be inconsistent with an application, vocabulary can be hard to understand, etc.—this can lead to dissatisfaction among readers [110, 111]. Moreover, a user may treat software as defective, not the manual. Consequently, savings made on a user manual may be illusory, a vendor can gain a bad reputation and/or a company can be exposed to additional expenses (e.g. higher costs of technical support [137]).

To reduce costs and improve the quality of a user manual, one can consider automatic generation. The benefits of generated documentation were noticed by Reiter et al. [128]. They claim that an automatic approach can reduce the costs of creation and maintenance of documentation, provide consistency between the product and its description, ensure compliance with standards, create explanations in many languages, adapt the complexity of a description to a target audience, and present information in many forms (e.g., text and graphics). However, one needs to create and maintain a project database with all the information required to generate new content.

Regarding the possibilities and limitations in the area of user manual generation, the following questions arise:

QUESTION 1. To what extent is it possible to generate a user manual on the basis of the artifacts available in a software project?

QUESTION 2. To what extent does a user manual, automatically generated on the basis of the artifacts available in a software project, meet the quality requirements established by the COCA quality model and the Documentation Evaluation Test?

JUSTIFICATION. Whether a user manual is written by a human or generated by software, it needs to fulfil quality criteria. To compare the quality of a generated user manual with a corresponding manual written by a human, we decided to use the COCA quality model and the DET method [10]. The COCA quality model allows one to evaluate a manual from the viewpoint of four orthogonal characteristics: completeness, operability, correctness and appearance. The DET is an additional approach which focuses on operability. □

In this paper, we propose a method for automatic generation of a user manual on the basis of the artifacts available in a software project. We describe how a project database is created and use it to generate a user manual (or parts of another document). We limit the research focus to manuals for web applications which are dedicated to people whose IT knowledge and experience is limited.

The goal of this paper can be summarized in the following statement:

GOAL. *Design a set of methods which allow one to generate a user manual for a web application on the basis of existing artifacts dedicated to people who are IT-laymen.*

This paper presents the initial work and is organized as follows. Section 6.2 discusses the content of a generated user manual. Section 6.3 presents documents which are commonly created in software projects. Section 6.4 briefly explains the generation process. Section 6.5 explains how to generate a user manual using software documentation only. Section 6.6 shows how requirements concerning the operating environment can be described. Section 6.7 focuses on the generation of exemplary usages and Section 6.8 shows how to cope with terms in a user manual. Section 6.9 presents the results of an early evaluation. Section 6.10 discusses related work. Finally, Section 6.11 concludes the paper.

6.2 Content of a generated user manual

In the context of the research goal presented in Section 6.1, the following issue can be addressed:

ISSUE 1. What kind of information should be included in a user manual and how should this information be organized?

To refer to a chapter in a user manual, the term *component* will be used, as proposed in ISO/IEC Std 26514:2008 [67].

6.2.1 Components of a user manual

To determine the content and structure of a user manual, an analysis of relevant literature and manuals was carried out. The literature review included research papers (including: [110, 111, 129]), guidelines (including: [34, 37, 145]), and standards (including: [22, 67, 68, 71, 74, 75]). The analysis of user manuals covered 9 documents used to create a COCA quality profile [10].

It was decided that a generated user manual should contain the following components:

- **Cover**—allows a reader to identify a user manual. The front cover can contain the name of an application and its version, the version of the user manual, the issuing organization, etc. The back cover can present the International Standard Book Number (ISBN) or other useful information.
- **Table of contents**—lists items with a corresponding page number. There can be many types of lists (e.g. a list of chapters, figures, tables, keywords, etc.).
- **Warning and Notices**—presents legal information, warnings, cautions, etc.
- **Conventions**—describes the conventions used in a user manual.
- **Introduction**—presents the concept and the idea behind the application, especially what kind of problem exists and how the software solves it.
- **Requirements concerning the operating environment**—lists the requirements that a user needs to fulfil in order to use the application (e.g. type and version of web browser).
- **Information objects**—presents data that a user creates, retrieves, updates or deletes using the described application.
- **Tasks**—lists available goals and describes how to achieve them using the application. Each goal is described by a procedure which needs to be followed and a description of events which can occur (i.e. procedures which describe special cases, e.g. how to deal with errors). If it is possible, an exemplary usage is presented. Goals are grouped by an actor who plays the main role (the one who performs a given task). Additionally, there is a description of the main actor.
- **Glossary**—lists and explains terms used in an application.

Component	Variant	
	Naive user manual	Complete user manual
Cover	✓	✓
Table of contents	✓	✓
Warning and Notices	✓	✓
Conventions	✓	✓
Introduction	✓	✓
Requirements concerning operating environment	—	✓
Information objects	✓	✓
Tasks:		
Actor	✓	✓
Scenarios	✓	✓
Examples	—	✓
Glossary	—	✓

Table 6.1: Proposed variants of a user manual

6.2.2 Variants of a user manual

Using the components in Section 6.2.1, one can create many variants of a user manual (e.g. to adjust content to a target audience). We propose two versions: *complete user manual* (which uses all the components in Section 6.2.1) and *naive user manual* (which seems to be the smallest usable set of components)—see Table 6.1.

COMPLETE USER MANUAL

NAIVE USER MANUAL

6.2.3 Completeness of selected components

To ensure that all information required by readers is within the scope of the generated user manual, a comparison with the recommendation of ISO/IEC Std 26514:2008 (also known as IEEE Std 26514-2010)[1] [67] and guidelines presented in the book *Read Me First!* by Sun Technical Publishing [145] was carried out. ISO Std 26514 describes various types of manuals (e.g. paper and electronic versions, instructional and reference modes)—we used a paper-based instructional-mode user manual for comparison. Sun's guidelines present variants with one and many chapters—we used a version with many chapters. We selected these variants, since they are often used in user manuals dedicated to IT-laymen.

The results of the comparison are presented in Table 6.2. There are differences in the scope of the listed components and their names, however, all required and recommended components are supported by *complete user manual.* When it comes to optional components, there are 3 items which are unsupported (i.e. *Appendixes*, *Bibliography*, and *Revision history*). However, if such information exists (in the data source) it can be reused.

[1]This standard supersedes a well known IEEE Std 1063-2001 [22]

ISO/IEC Std 26514:2008 [67]	**Sun Technical Publishing [145]**	**Component of Section 6.2.1**
Required and recommended		
Identification data	Title page	Cover
Identification data	Legal notice	Warning and Notices
Table of contents	Table of contents	Table of contents
Introduction	Preface	Introduction
Information for use of the documentation	Preface	Conventions
Concept of operations	Preface	Introduction
Procedures	(content of a) Chapter	Tasks (Scenarios)
Error messages and problem resolution		Tasks (Examples)
Glossary		Glossary
Index	Index	Table of contents
Optional		
List of illustrations	List of figures	Table of contents
	List of tables	Table of contents
	List of examples	Table of contents
	Chapter table of contents	Table of contents
	Appendixes	—
	Glossary	Glossary
Related information sources	Bibliography	—
	Revision history	—

Table 6.2: Comparison of components listed by ISO/IEC Std 26514:2008 (paper based, instructional mode) [67] and Sun Technical Publishing (version with multiple chapters) [145], with components selected for a *complete user manual.*

6.3 Universal artifacts of software projects

Instead of requiring specified data (to generate a user manual), we propose reusing existing artifacts; thus we consider the following issue:

ISSUE 2. What kind of artifacts are commonly available in a software project?

There is a certain body of information which seems to be used quite often, regardless of the methodology used in a project [45, 66, 135, 148]:

- Business Case,
- Software Requirements Specification, and
- Acceptance Tests.

A *Business Case* presents the justification for initiating a project. For example, a project which uses PRINCE2 methodology has a document which covers: an executive summary, reasons, business options, expected benefits and drawbacks, timescale, costs, investment appraisal, major risks [148].

Software Requirements Specification (SRS) describes the software to be developed. It can be created according to IEEE standards (i.e. IEEE Std 830:1998 and its successor ISO/IEC/IEEE Std 29148:2011 [61, 72]), a product backlog, or any other form. SRS often includes *Functional Requirements*, *Non-functional Requirements*, and *Technical*

Constraints—each of them can be organized in many ways, but natural language seems the most popular mode (e.g. use cases or user stories).

Functional Requirements describe the functions an application should support, while *Non-Functional Requirements* describe "how" the system should provide these functions and what its constraints are (e.g. taking into account the minimal system resource utilization requirements, one can decide whether the system is useful for them or not). *Technical Constraints* are requirements which affect software architecture [73].

The goal of *Acceptance Tests* is to ensure that the software meets the acceptance criteria [70, 114]. The creation of automatic *acceptance tests* for web applications can be problematic at the early stages of a project, thus, one could create tests using a GUI mock-up (a layout of a web application) [114, 115].

Additionally, some documents may contain a *glossary*, which can be reused as well.

6.4 Generation of a user manual

In addition to the Issue 2, one also needs to consider the following issue:

ISSUE 3. What kind of data are required to generate a user manual and where can these data be found?

6.4.1 Design assumptions

The following assumptions have been made concerning artifacts:

ASSUMPTION 1. The following artifacts are available: *1)* Business Case, *2)* Software Requirements Specification, *3)* Acceptance Tests, and *4)* an interactive mock-up or a running application.

Artifacts can vary between vendors and methodologies (e.g. due to tailoring), thus, it is important to clarify their content:

ASSUMPTION 2. Functional requirements are defined in use cases [29] using Formal USE cases notation (FUSE) [108].

JUSTIFICATION. Functional requirements can be represented in many forms, especially as use cases or user stories [7, 31, 76]. Both representations are written using natural language (which allow functionality to be described in an easy to understand way), however, use cases seem to be more suitable for automatic analysis [113]. FUSE notation allows us to organize the structure of use cases. □

ASSUMPTION 3. Acceptance tests are defined using Test Description Language (TDL) [114] and GUI mock-ups are defined using ScreenSpec [115].

JUSTIFICATION. ScreenSpec notation allows one to define GUI mock-ups, which can be used as screenshots in a manual (such images can be replaced when a working application is available). To capture screens one needs to run an application—this can be done thanks to acceptance tests. TDL allows one to define acceptance tests and connect them to use cases, which allows us to see what a test is responsible for (coverage of tests can be checked as well). Moreover, GUI elements defined using ScreenSpec can be referenced by TDL and FUSE [114]. □

ASSUMPTION 4. Non-functional requirements (NFR) are defined using Non-functional Requirement Templates (NoRTs) [89] and Technical Constraints (TC) are defined using Technical Constraint Templates (TeCTs).

JUSTIFICATION. Use of the catalogue of NoRTs to define NFRs ensures the completeness of such requirements. TeCTs use the same approach. □

To focus on the goal of Section 6.1, we also assume:

ASSUMPTION 5. All artifacts are up-to-date, well written and follow guidelines.

JUSTIFICATION. The goal is to generate a user manual, not to analyze the quality of information stored in artifacts. For example, we assume that all acceptance tests are passed, business objects and actors are defined, etc. □

6.4.2 Project database

Information from artifacts is used to construct a project database (which is further used to generate a user manual). The following types of data are stored:

- *fixed*—pieces of information created once (by a human) and reused each time a manual is generated (stored in configuration files); FIXED TYPE
- *imported*—existing data from artifacts, reused without any modification; IMPORTED TYPE
- *generated*—new content generated on the basis of existing data (see sections 6.6, 6.7, and 6.8). GENERATED TYPE

By using data from artifacts and configuration files, one can fill a project database. Table 6.3 summarizes the variables available in a project database and their origin. A full description of the project database is presented in Appendix B.1.

Variable	Description	Artifact	Type
`name`	Name of an application	SRS	Imported
`version`	Version of an application	SRS	Imported
`problem`	Problem description	Business case	Imported
`scope`	Description of how a given problem is solved	SRS	Imported
`ucs`	Use cases	SRS	Imported
`nfrs`	Non-Functional Requirements	SRS	Imported
`tcs`	Technical Constraints	SRS	Imported
`bos`	Business Objects	SRS	Imported
`actors`	Actors	SRS	Imported
`tests`	Acceptance Tests	SRS	Imported
`glossary`	Glossary	SRS or other document	Imported
`wans`	Warnings and notices	Configuration file	Fixed
`convs`	Conventions used in a manual	Configuration file	Fixed

Table 6.3: Project database

6.4.3 Templates

A special type of *fixed* element is a *template* (a *template* can be a file or value of a variable). It organizes the structure of a user manual by defining where to put a given component and how to create it (using *fixed*, *imported* and *generated* elements). The content of a user manual is defined in *document template*; it includes other templates which can generate the contents of components (included templates can be understood as boilerplates presented in Chapter 5). TEMPLATE

Each template is defined using notation supported by the *jinja2* engine [132] (its syntax is similar to Django and other popular frameworks used in web development [57]). A template can contain text (which is copied without any changes), enter data from a project data base (variables are accessible by using object `project` and a dot notation; references needs to be entered in double braces, e.g. to access the name of an application one needs to type `{{ project.name }}`), execute an expression (in curly-percent syntax, e.g., `{% for uc in project.ucs %}`), include a template (e.g., `{% include "toc.txt" %}`), etc. A detailed description is available at *jinja2*'s web page [132].

The output from *jinja2* is provided in Latex format [94]. We decided to generate content for Latex, since it simplifies typesetting and allows us to automatically create a table of contents and other lists of elements.

A part of an exemplary *document template* is presented in Figure 6.1. Each *template* can be modified to provide more suitable explanations.

6.5 Naive user manual

Fixed and *imported* data allow one to generate a *naive user manual* presented in Section 6.2 (see Table 6.4). In this approach, one can copy scenarios from use cases

```
\documentclass[final,a4paper,11pt,oneside]{memoir}
\begin{document}
{% include "frontcover.txt" %}
{% include "toc.txt" %}
{% include "introduction.txt" %}
...
\end{document}
```

Figure 6.1: Part of an exemplary document template

Component	**Variable**	**Artifact**		
		Configuration file	*Business case*	*Software Requirements Specification*
Cover	`name`			✓
	`version`			✓
Table of contents	N/A			
Warning and Notices	`wans`	✓		
Conventions	`convs`	✓		
Introduction	`problem`		✓	
	`scope`			✓
	`actors`			✓
	`ucs`			✓
Information objects	`bos`			✓
Tasks:				
Actor	`actors`			✓
Scenarios	`ucs`			✓

Table 6.4: *Naive user manual* using a project database of Section 6.3 (some variables are used to create many components).

to describe tasks. However, it seems naive to base an explanation of an application only on scenarios from use cases, especially as a well written use case has neither references to GUI elements nor exemplary data. This concern was confirmed in exploratory studies (an unstructured brainstorming meeting with 3 subjects), i.e. participants expressed their dissatisfaction concerning the presented user manual and addressed many issues, e.g.:

- *Presentation of a procedure gives only an overview of a task, a set of examples would be more beneficial.*
- *There are neither screenshots of an application nor exemplary data in an explanation of the procedure.*
- *The order in which tasks are presented varies from the order in which users perform tasks.*

However, the biggest concern was about web page address: the user manual did not provide this information, it was not even present in the project database.

6.6 Requirements concerning the operating environment

In the context of the weaknesses of *naive user manual* (of Section 6.5) we considered the following issue:

ISSUE 4. Where can one find a web page address?

Moreover, a *complete user manual* needs to present requirements concerning the operating environment, thus, the following issue can be considered as well:

ISSUE 5. Where can one obtain requirements regarding users' computers and users' education (knowledge, skills, etc)?

To handle these issues, non-functional requirements and technical constraints are of value. These requirements are often defined using natural language, however, one can use a catalogue of Non-Functional Requirements Templates (NoRTs) dedicated to the elicitation and specification of NFRs, proposed by Kopczyńska and Nawrocki [89]. One can browse the catalogue, choose an appropriate NoRT, fill in the NoRT and, as a result, an NFR is created. We decided to employ the method, but in the reverse order. Since NoRTs are well-structured, it is possible to automatically recognize which template is used and then extract information. The same approach can be used with technical constraints (TC). According to our knowledge, there is no catalogue of technical constraint templates (TeCTs), as was proposed for NoRTs.

The whole process is presented in Fig. 6.2. We analyze the given NFRs and TCs in the following way:

1. *Identify*—identify which NoRT or which TeCT is used,
2. *Extract data*—extract values of parameters from NFRs or TCs,
3. *Process data*—pre-process parameters (a set value of additional parameters, if required).

An exemplary analysis of an NFR is presented in Fig. 6.3.

The collected data can be used to generate content for a user manual. If some data are missing (e.g. there is no NFR with a web page address), the generation process can be aborted with an error message.

Appendix B.2 lists NoRTs and TeCTs considered as valuable for a user manual.

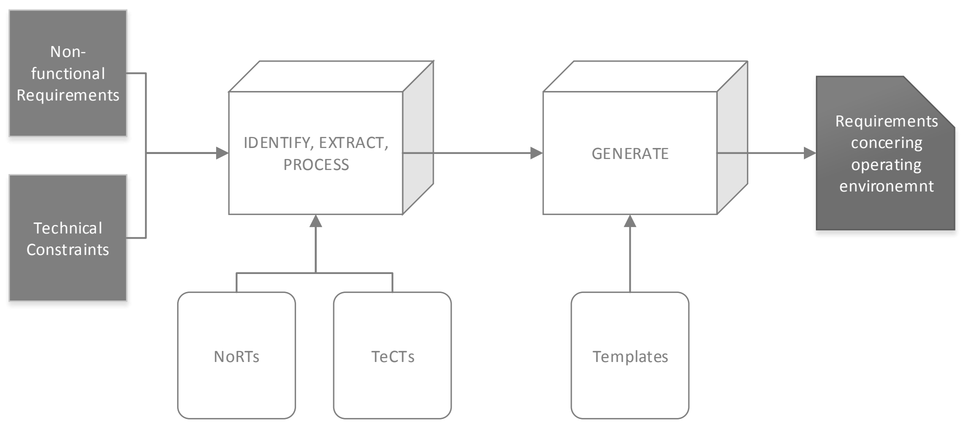

Non-functional Requirements
Technical Constraints
IDENTIFY, EXTRACT, PROCESS
GENERATE
Requirements concering operating environemnt
NoRTs
TeCTs
Templates

Figure 6.2: Data analysis and generation of the *Requirements concerning the operating environment* component.

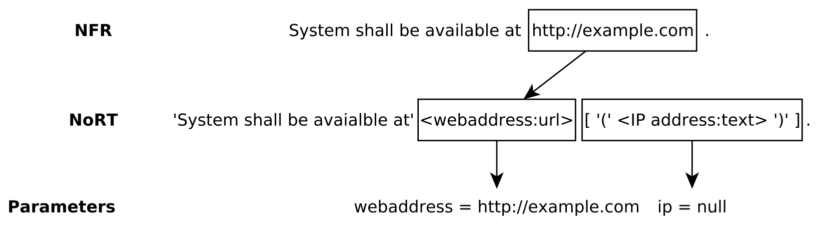

NFR
System shall be available at http://example.com .
NoRT
'System shall be avaialble at' <webaddress:url> ['(' <IP address:text> ')'] .
Parameters
webaddress = http://example.com
ip = null

Figure 6.3: Exemplary analysis of an NFR with a web page address.

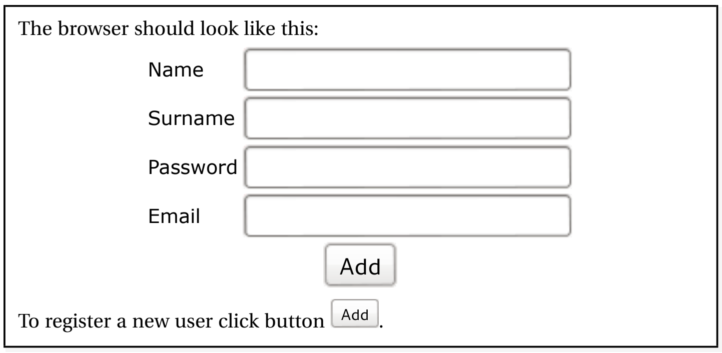
6.7 Exemplary usages

According to the participants in the exploratory studies (see Section 6.5), the description of tasks should be enriched with examples. Such examples should present how to accomplish a task by presenting an interaction between a user and a system using screenshots and real data. We will use the term *exemplary usage* to refer to such an explanation. *Exemplary usage*s are often written by humans, however, it is tempting to check whether it is possible to generate them. This section discusses the following issue:

ISSUE 6. How can we generate an exemplary usage of a web application, which presents the interaction between a user and a system, using screenshots, exemplary data and a narrative explanation?

Acceptance tests contain exemplary data (which can be used to run an application to collect screenshots and provide exemplary data) and use cases describe the intentions behind a user's actions. By using these data one should be able to generate an *exemplary usage*.

We decided that the main ingredient of *exemplary usages* are screenshots. Each captured image is preceded by a brief description and followed by the activities required to carry on a task. A part of an example is presented in Figure 6.4. The appearance of screenshots depends on whether a working application or GUI mock-ups are available.


The browser should look like this:

To register a new user click button Add.

Figure 6.4: A simple example of a user's interaction with an enriched system.

6.7.1 Find relationships between data

There can be many test cases for one use case, thus one needs to select the most appropriate acceptance test. Unfortunately, at this stage there are no relationships between data in a project database. This concerns not only references to use cases in test cases, but also actors, mock-ups, etc. Hence, the following issues need to be solved first:

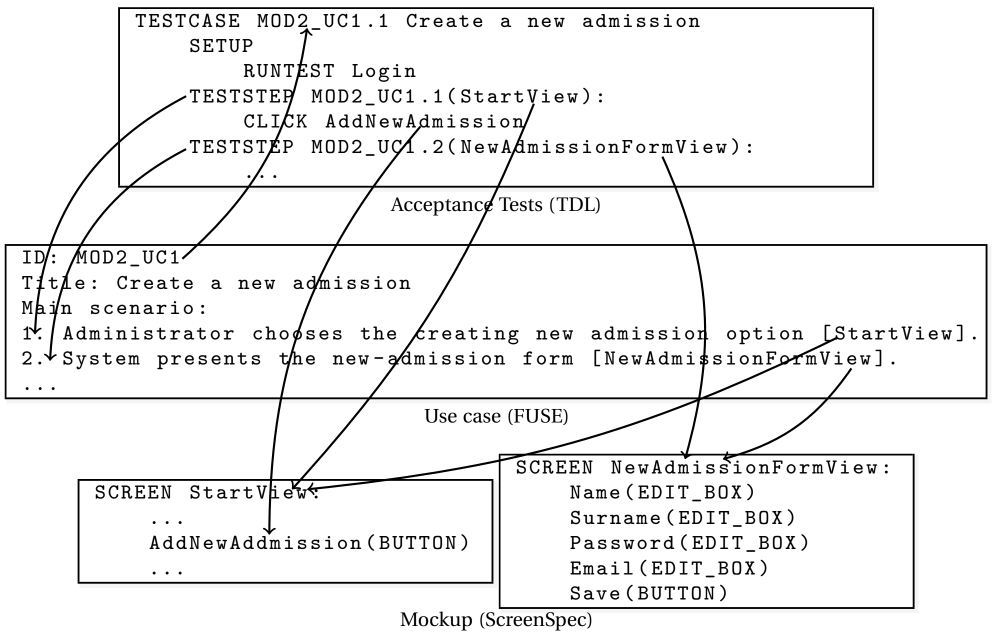
ISSUE 7. Find relationships between use cases, test cases and mock-ups.

ISSUE 8. Find references to actors and business cases in use cases.

ISSUE 9. Find test cases suitable for exemplary usages.

Notation used to store functional requirements, acceptance tests and GUI mock-ups (i.e. FUSE, TDL and ScreenSpec) assumes that identifiers are consistent between artifacts [114], thus, to find relationships one needs to browse labels and search for matches. A simple example is given in Figure 6.5. (this example is based on *Admission System version 2.0F (quantitative version)* from UCDB [12]).

A more complicated task is to find references to actors and business objects in use case steps. Simple text comparison is not enough here, since words in sentences may be written in a number of linguistic forms (e.g. singular or plural form) and



```
TESTCASE MOD2_UC1.1 Create a new admission
    SETUP
        RUNTEST Login
    TESTSTEP MOD2_UC1.1(StartView):
        CLICK AddNewAdmission
    TESTSTEP MOD2_UC1.2(NewAdmissionFormView):
        ...
```

```
ID: MOD2_UC1
Title: Create a new admission
Main scenario:
1. Administrator chooses the creating new admission option [StartView].
2. System presents the new-admission form [NewAdmissionFormView].
...
```

```
SCREEN StartView:
    ...
    AddNewAddmission(BUTTON)
    ...
```

```
SCREEN NewAdmissionFormView:
    Name(EDIT_BOX)
    Surname(EDIT_BOX)
    Password(EDIT_BOX)
    Email(EDIT_BOX)
    Save(BUTTON)
```

Figure 6.5: Exemplary relationship between different data sources (selected fragments, the presented use case is from UCDB [12].

multiple words may be used to refer to an item (e.g. text *New admission form* may be used as a name for a business object). To find matches, we decided to use natural language understanding tools. First, using the Standford toolkit [96], we carry out a linguistic analysis of a step (i.e. segmentation, tokenization, part-of-speech tagging, lemmatization, and parsing). Next, using a step constructed from *lemmas* (words in their base form) and *maximum matching algorithm* [36], we search for references (this algorithm allows us to find multiword expressions). When a connection is found, the structure of a step is modified to reflect a reference.

When only a working application is available, it is necessary to reconstruct GUI mock-ups, as they are used to store screenshots and the widget's type. An initial structure of mock-ups is created on the basis of TDL and clarified while screenshots are captured (see Appendix B.1 for more details).

6.7.2 Selection of acceptance tests

To select a proper acceptance test, one needs to solve the following issue:

ISSUE 10. How to decide which acceptance test is the most suitable as the basis for an *exemplary usage*?

We decided to design metrics *widget coverage* and *event coverage* which will indicate the most suitable variant.

Widget coverage is computed for test cases which test the main scenario. Here, the number of web elements used in a test case is counted (one can count the percentage of used widgets, but that needs the total number of the web element—value which is not present when GUI mock-ups are unavailable). Thanks to preliminary studies, we have concluded that an example with the highest number of tested widgets is the most suitable for readers (as one of the participants stated, this allows one to *check everything, not only the less problematic items*).

Metric *Event coverage* allows to select the minimum number of test cases which cover the highest number of events (i.e. the *minimal covering suites* [53]). First, the total number of events in a use case is counted, and for each test case a vector with the exact number of zeros is assigned. Next, the content of each test case is analyzed: if an event is checked then a value `1` is assigned to an index which represents a given event in a vector. Finally, test cases which allow one to present all events are chosen. For example, if there are two events, a vector `[0, 0]` is assigned to each test case. For three test cases one can have the following vectors: `[1, 0]`, `[0, 1]`, `[1, 1]`. The last variant covers all events, thus, it is used in a user manual. If there is no variant that covers all events, a number of tests are used.

6.7.3 Planning generation of an exemplary usage

The next challenge is connected with this issue:

ISSUE 11. How can we generate an *exemplary usage* for selected acceptance tests?

To describe the content of acceptance tests, one needs to know the intentions behind them and these can be found in use cases [29]. First, we search for activities in use case steps, next we plan an explanation.

Activities in use case steps

An activity allows us to identify the process triggered by an actor in a use case step. An example of a process may be data provision and the corresponding activity is `ENTER`. A list of supported activities (with exemplary steps) is presented in Table 6.5 (this list is based on research presented by Ochodek et al. [112, 113] and Jurkiewicz [83]). There is one additional activity *WAIT* which is used to tag situations in which an actor waits for a system to perform a task.

To detect an activity, a list of words which can be used to trigger a process was prepared (the same approach was used in the presented literature). For example,

Table 6.5: Exemplary activity types [83, 112, 113].

Activity type	**Exemplary step**
ADD	Author adds new comment.
CONFIRM	Author confirms the modifications.
DELETE	Author delete the comment.
DISPLAY	System displays available comments.
ENTER	Author enters data.
FINISH	Author finishes the task.
READ	Author browses posted comments.
SELECT	Author selects type of the movie.
UPDATE	Author updates the comment.
WAIT	Author waits for the email with results of analysis.
VALIDATE	System validates the comment.

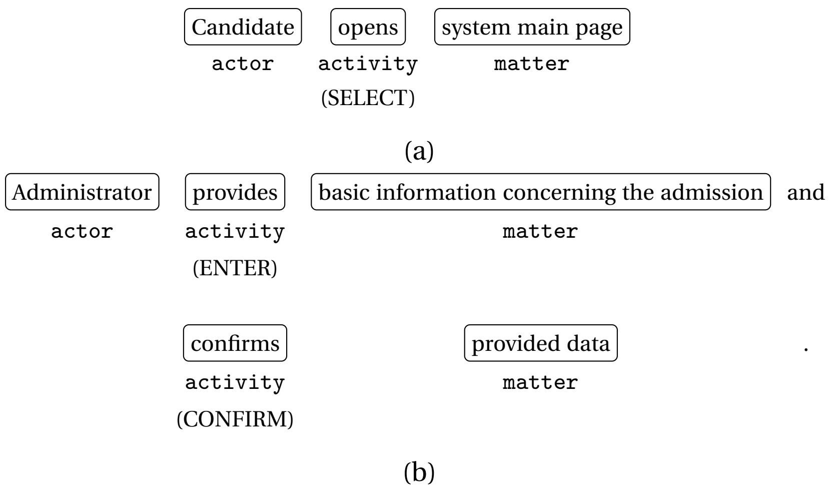


Figure 6.6: Annotated step with one activity (a) and two activities (b).

to detect an activity `ENTER` a system can search for the following words: *enter*, *type*, *provide*, etc.

After finding activities, a system searches for additional information which can be used while generating an explanation. As a result, each use case step is enriched with three additional variables: `actor` (which is a reference to an `Actor`), `activity` (which contains name of an activity), and `matter` (with supplementary information). Examples of annotated steps are provided in Figure 6.6.

Templates

To plan an *exemplary usage intermediate template*s are first selected. They are not evaluated, but merged into one template. Next, a test case is executed and the data required by *intermediate templates* are collected (e.g. screenshots). Finally, the resultant template is evaluated and the output is put into a user manual.

To select *intermediate templates*, a simple domain specification language was

```
Plan = Rule+
Rule = Statement+, "{", Template, "}";
Statement = ["(", Condition, ")"], Actor, Activities, TestSteps;
Activities = "<", Activity, [Quantity], {",", Activity, [Quantity]}, ">";
TestSteps = "[", Step, [Quantity], {",", Step, [Quantity]}, "]";
```

Figure 6.7: Domain Specification Language for template selection (in EBNF).

designed (see Figure 6.7, with grammar in EBNF [63]). This notation allows us to create a set of rules, and each rule consists of a set of statements and an *intermediate template* used for generation. A statement describes the relationship between actors, activities and test steps, while the associated template defines how to explain this situation.

`Statement` consists of a name of an actor (part `Actor`); an activity found in a use case step (`Activities`; a comma separated list of activities of Table 6.5) and a list of test steps used to check a given use case step (`Tests`; a comma separated list of commands used in test steps, e.g., `CLICK`, `SET`). For example, the statement

```
Student < SELECT > [ CLICK ] { ... }
```

means that if there is a use case step with an actor `Student` who triggers an activity `SELECT` by `CLICK`ing a widget, then a given template is used.

There are three additional facilities:

- If there is no need to precisely define the name of a command in a test step, a keyword `ANY` can be used (it represents any user action or any assertion).
- In the case of many activities or many commands, one can use a part `Quantity`, which enables one to define the number of occurrence (it supports quantifiers used in regular expressions, i.e. `+`, `*`, `{n,m}`, etc.).
- To refer to any actor other than a system, one can type `ANY_USER` as the actor's name.

For identical statements (and the need to use different templates), a `Condition` can be added—it allows one to decide whether to use a template on the basis of data in a project database. The syntax for conditions is similar to C language (including `&&` and `||`). Issues concerning data access are described below.

To specify an interaction between many actors, one can type multiple statements, e.g.:

```
System < DISPLAY > [ ANY* ]
User < ENTER, CONFIRM > [ SET+; CLICK ]
User < SELECT > [ CLICK ] {
     The browser should look like this:
     {{ $1.tests[0].screen["pre"]|screen }}
     Select {{ $1.ucstep.matter }} by clicking
     {{ $1.tests[0].component.screen["pre"]|screen('inline') }}.
}
```

Figure 6.8: A simple example of a rule for generating *exemplary usage*.

A statement is followed by an *intermediate template* (it is put in braces, i.e. {}). Figure 6.8 presents an example of a statement and a template (explained in the following section). Strings after the pipe character (|) are filters, specially designed functions which simplify the design of templates. To access data from a given use case step and a test case, a special variable was introduced. It starts with a dollar sign ($) and is followed by the number of a statement ($1 represents the first statement). This variable has the following variables:

- `actor`—information about an actor (see `Actor` in Appendix B.1);
- `activity`—name of an activity (see Figure 6.5);
- `ucstep`—access to a step in a use case (see `UseCase` in Appendix B.1);
- `teststeps`—access to test steps (square brackets used to access a given action or assertion (see `Series` in Appendix B.1);

Variables `ucstep` and `teststeps` both have an additional variable `parent`, which allows access to a use case and a test case. Additionally, `ucstep` has variable `matter` (see Section 6.7.3).

While checking the statement in a rule, the longest match is used.

6.7.4 Generating an exemplary usage

Narrative description is generated using *template approach* [104, 126], which assumes the existence of a pattern with gaps that need to be filled-in (e.g. by using data from a project database). Since all input data are linguistically consistent, one can create patterns which fit their grammatical form—no linguistic transformation is required (see the assumptions of Section 6.4). For example, using the use case step from Figure 6.6(a), the template of Figure 6.8, and a test step `CLICK OPEN`, one can generate the following output (the screen inside the template is replaced by `<img>` to reduce its size):

```
The browser should look like this:
<img>
Select system main page by clicking OPEN.
```

During our preliminary studies, it occurred that sometimes it is necessary to add an article or adjust the grammatical form of a word. This is not a hard task, but it complicates templates. To facilitate this, we added the filters `article` and `dict`. The former allows us to determine the indefinite article (it checks whether a word is countable and adds a proper article). While the latter allows us to change the grammatical form of a word or a phrase. For example, in the case of a template

```
{{ word|dict(pos='verb', number='singular', person=first') }}
```

and variable `word` with value *be* a string *am* is returned.

6.8 Glossary

A glossary in a generated user manual can contain terms from artifacts and field explanations [13]. However, project documentation can contain terms which are not dedicated to end-users (e.g. some of them may be to technical), thus we need to solve the following issue:

ISSUE 12. How can we list in a glossary only those entries which are used in a user manual?

To remove redundant items, we generate a document without a glossary, perform an NLU process so that the user manual is constructed from *lemma*s and use a *maximum matching algorithm* to search for terms. When all redundant terms are removed, a user manual with a tailored glossary can be generated.

6.9 Early evaluation

To evaluate the quality of a *complete user manual* generated with methods introduced in sections 6.6, 6.7, and 6.8, we carried out an exploratory study to eliminate possible weaknesses[2], implemented a prototype, generated a user manual and conducted an experimental evaluation.

6.9.1 Exploratory study

The study was conducted in two stages. At the beginning of the first stage we prepared three variants of the following components: *Introduction*, *Requirements*, *Information objects*, and *Tasks*[3]. Next, we organized meetings and discussed our propositions. There were four meetings, to which we invited two programmers (people with 1-3

[2]*Naive user manual* had its own exploratory study

[3]We focused on these components, since others are fully customizable.

Table 6.6: Assignment of components in the preliminary study (abbr. **P** stands for programmers, **U** for IT-laymen; size in number of pages).

Component	**Stage 1**						**Stage 2**					
	P1	**U1**	**P2**	**U2**	**Time**	**Size**	**P3**	**U3**	**P4**	**U4**	**Time**	**Size**
Introduction	✓	✓			20 min	3*1 pages	✓	✓			10 min	1 page
Requirements			✓	✓	20 min	3*1 pages			✓	✓	10 min	1 page
Information objects			✓	✓	20 min	3*1 pages			✓	✓	10 min	1 page
Tasks	✓	✓			40 min	3*2 pages	✓	✓			20 min	2 pages

years of commercial experience in software development, the average age was 28) and two IT-laymen (i.e. people who use a computer during their daily activities, but do not have any technical education, the average age was 41). There was one appointment per participant (we decided to organize separate meeting due to the age gap).

Each meeting had the following goal[4]:

GOAL. *Analyze two sets of components which can be used to create a user manual (each set contains three variants of a component) for the purpose of designing one suitable version with respect to web applications from the point of view of programmers and IT-laymen in the context of brainstorming.*

Each meeting was organized according to the following agenda: *1)* Present the goal of the meeting and the agenda, *2)* Read, discuss and improve the first set of components, *3)* Read, discuss and improve the second set of components. While discussing each set of components, first we presented three variants of components and allowed a particular time to read them, then we started a discussion in the form of brainstorming (e.g. a participant could select the most suitable version, recommend improvements or propose a new version). Table 6.6 presents the organization of the preliminary study (how components were assigned to participants).

After conducting all the meetings, we designed a new version of each component and started the second stage. This time, we presented one variant of each component and asked participants to improve it. The goal of the meeting was as follows:

GOAL. *Analyze two different components which can be used to create a user manual with respect to web applications from the view point of programmers and IT-laymen in the context of brainstorming.*

Observations and improvements for both stages are discussed in Section 6.9.2.

[4]The goal is created using the GQM approach [16]. However, questions and metrics were omitted since we used brainstorming sessions.

6.9.2 Improvements

To increase the readability of a generated user manual, a number of improvements were proposed:

OBSERVATION 1. The generated user manual can contain screenshots of a web application (which, e.g. present exemplary data or expected output). Unfortunately, some images can be very large and may not fit a page. Such a images can be resized, but this can result in them being unreadable.
SOLUTION 1. To get a good fit and readable content, a filter `screen` was introduced which allows us to crop an image to a desirable size (a predefined value). To prevent the removal of elements which are important to readers (e.g. fields which are filled in), localization of used web elements is stored—this allows us to determine which parts of a web page can be safely removed.

Filter `screen` allows us to put an image in the middle of a page and inline an image in a text (`screen('inline')`).

OBSERVATION 2. A web application can have a number of clickable elements. While reading a user manual, it may be difficult to find a required item on a screenshot.
SOLUTION 2. An element on a screenshot can be marked by a frame, or a pointer can be drawn on a margin (e.g. in the form of a dot or an arrow). Moreover, one can decide how to visualize clickable elements in a text description, i.e. whether to include a button label, text from the `alt` tag (in the case of clickable images) or to include a screenshot of an element.

OBSERVATION 3. Modification of some input fields in an application may be impossible or not recommended (e.g. in the case of pre-filled widgets).
SOLUTION 3. Fields which a user should not modify can be presented in a different colour.

OBSERVATION 4. The order of use cases in project documentation may not be similar to the order in which users performs tasks.
SOLUTION 4. Users' tasks are sorted according to their occurrence in use cases with the `level` set to `Business`.

OBSERVATION 5. It is impossible to refer to elements in a user manual by page number.
SOLUTION 5. To point out the description of other elements in a user manual, a filter `page` was introduced. It allows us to postpone page numbering, e.g. one can refer to a business object in the following way "`see page {{ project.bos["Admission Form"]|page}}`" and after Latex processing, a text *see page 12* can be generated.

Currently, the following elements can be referenced by a page number: *information objects*, *actors*, *use cases*, *nfrs*, and the terms in a *glossary*.

6.9.3 Empirical evaluation

A user manual for the *Plagiat.pl* [122] application was generated (see Appendix B.3) and evaluated using the COCA quality model and the Documentation Evaluation Test (DET)—methods designed to evaluate the quality of a user manual [10]. *Plagiat.pl* is an application which allows one to detect plagiarism in different types of document, e.g. an M.Sc. thesis.

Before generating a user manual, required artifacts were created on the basis of our experience with the *Plagiat.pl* application (including use cases and acceptance tests). When it was possible, we tried to reuse content from the original manual. The generated user manual was in the English language, but since most participants were speakers of Polish only, we decided to translate it manually into Polish. Moreover, while creating the COCA quality profile, a Polish manual was used as well.

Participants

The generated user manual was evaluated by 3 Experts, the same experts who checked a corresponding version created by humans [10]. There were 16 prospective users, all in their first year of study at university, on the computer science programme. All of them declared that they were unfamiliar with the *Plagiat.pl* application.

Experiment

The assessment was performed according to the procedure proposed by Alchimowicz and Nawrocki [10]. First we carried out an evaluation with the Experts to check the completeness and correctness of the user manual. This was done by asking three questions, originally introduced by the COCA quality model. The questions asked to the Experts and their answers are presented in Table 6.7 (questions Q1, Q2, and Q5; additional data from a quality profile are provided for comparison [10]). The following answers were available for each question: *Not at all*, *Weak*, *Hard to say*, *Good enough*, and *Very good*. To simplify data presentation, *Not at all* and *Weak* are collated in the column "–", and the results for *Good enough* and *Very good* are in column "+". Column *Hard to say* is abbreviated as *?*. Raw data are presented in Appendix B.6.

Since the results of the Experts were no worse than the data in the profile, we continued the procedure and evaluated the manual from the Prospective Users' standpoint. First, according to the DET method, each prospective user was asked to find an answer to a question and note the number of the page which presents the answer (or a mark that she/he was unable to find it[5]). For that purpose, 29 questions asked while evaluating the human version of the manual were used [10].

Next, using questions from the COCA quality model, completeness, operability, and appearance were evaluated (questions Q3, Q4, and Q6 of Table 6.7).

Interpretation

The answers provided by the Experts in both variants are similar. This is not surprising, since the generated variant is based on an original version (questions Q1, Q2, and Q5).

Prospective Users were able to find 85.13% of the correct answers (the results for the DET method are in Table 6.8) which is a better result than the original version and the quality profile (by 2.16% and 4.09% respectively). The average time was about 10% higher when compared with the human made version. However, the difference seems negligible. In the case of the COCA quality model and Prospective Users (questions Q3, Q4, and Q6 of Table 6.7). The percentage of positive answers (column "+") for all characteristics is higher than positive answers for the human-made version. However, when compared to the quality profile, results for completeness (question Q3) and operability (question Q4) are slightly lower (2% and 2.1% respectively). In the case of appearance (question Q6), the results are better for the generated version, when compared with the human-made version and the quality profile.

The presented results come from an early evaluation. To decide whether the generated manual is better than the hand made one, more experiments are required.

In the case of DET, we observed a small difference between the mean numbers of correctly answered questions in favour of the group assessing the generated *Plagiat.pl* user manual (around 2%). Our hypothesis was that the generated manual is no worse than the one created manually, therefore we decided to use the Wilcoxon rank-sum statistical test to investigate this hypothesis[6]. As a result, we were not able to reject the null hypothesis about the equality of median numbers of correctly answered questions (two-tailed test, α=0.05, p-value=0.75). Of course, the fact that we were not able to reject the null hypothesis does not confirm that such a difference does not exist between the populations. However, taking into account the fact that the

[5]Correct answers without the page number or an incorrect page number were counted as incorrect answers.

[6]We decided to use a non-parametric statistical test, because we suspected that the assumption about sample normality might be violated (obtained Shapiro-Wilk tests p-values were equal to 0.38 and 0.03).

Table 6.7: COCA quality characteristics of the user manual for *Plagiat.pl*.

Id	Characteristics and the associated question	Quality profile –	Quality profile ?	Quality profile +	Human made –	Human made ?	Human made +	Generated –	Generated ?	Generated +
Completeness										*responsible: Expert*
Q1	To what extent does the user documentation cover all the functionality provided by the system with the needed level of detail?	22.22%	29.63%	48.15%	0.00%	33.33%	66.67%	0.00%	33.33%	66.67%
Q2	To what extent does the user documentation provide information which is helpful in deciding whether the system is appropriate for the needs of prospective users?	3.70%	11.11%	85.19%	0.00%	0.00%	100.00%	0.00%	0.00%	100.00%
										responsible: Prospective User
Q3	To what extent does the user documentation contain information about how to use it with effectiveness and efficiency?	15.60%	7.40%	77.00%	25.00%	6.25%	68.75%	12.50%	12.50%	75.00%
Operability										*responsible: Prospective User*
Q4	To what extent is the user documentation easy to use and helpful when operating the system documented by it?	8.20%	14.90%	77.10%	31.25%	12.50%	56.25%	0.00%	25.00%	75.00%
Correctness										*responsible: Expert*
Q5	To what extent does the user documentation provide correct descriptions with the needed degree of precision?	18.52%	25.93%	55.56%	0.00%	33.33%	66.67%	0.00%	33.33%	66.67%
Appearance										*responsible: Prospective User*
Q6	To what extent is the information contained in the user documentation presented in an aesthetic way?	13.60%	12.20%	74.30%	31.25%	12.50%	56.25%	12.50%	6.25%	81.25%

observed normalized effect size[7], expressed as Cohen's d coefficient [30], was between "small" and "medium"[8] ($d = 0.29$) it would be difficult to detect the difference in the experiment if it truly existed between the populations (post-hoc power 1-β was equal to 0.11). Concluding, in our opinion it seems that it is unlikely that the true difference in the median numbers of correctly answered questions is higher for the population of participants assessing the original, manually created, *Plagiat.pl* user manual.

Documents concerning the evaluation required by the COCA quality model and the DET method are available in appendix (*Evaluation Mandate* is available in Appendix B.4 and *Evaluation Form* in B.5).

[7]Please note that the retrospectively calculated effect size only approximates the real effect size in the populations from which the samples were drawn.

[8]According to Cohen [30] effect size is perceived as "small" if the value of d is equal to 0.2, as "medium" if the value of d is equal to 0.5, and as "large" if d is equal to 0.8.

Table 6.8: DET operability of the user manual for *Plagiat.pl*.

	Profile	Version	
		Human made	Generated
Number of participants	148	16	16
Average time of searching all answers	49 min	39 min	42 min
Percentage of correct answers	81.04%	82.97%	85.13%

Threats to validity

To ensure that the correct conclusions are drawn from the results of the experiment, some threats to validity need to be discussed. We have classified them into two groups: internal validity and external validity.

Internal validity refers to factors in an experiment which were not fully controlled by the researchers and could affect the results.

Variation in the knowledge and experience of the participants is one of such threats. To mitigate this issue, the experiment was carried out with participants with levels of IT knowledge and experience similar to the participants who took part in the creation of quality profiles for the COCA quality model and DET method.

The next threat to validity is related to the fact that input artifacts used to generate user documentation for the *Plagiat.pl* system were reverse-engineered based on an existing, working application. As a result, the artifacts could differ from the ones that were developed internally by the company. An additional issue is that authors may have unintentionally introduced changes which made the generation of user documentation easier than it would have been in the case of original documents. In this context, however, it is important to emphasize our previous assumptions (see assumptions in Section 6.4) that all the required artifacts are available, and they contain all the information required to generate user documentation.

The necessity of translating the generated user manual into Polish is a threat as well. To prevent making unintentional improvements, the English templates were translated into Polish and then verified by a professional translator in order to check if they correspond to the originals.

External validity refers to any factors which could affect the possibility to generalize the results of the experiment to the wider population.

Here, the threats to validity concern representativeness of the sample, i.e. participants did not represent all the potential end-users in the target audience and the sample size was relatively small, there were only 16 Prospective Users. Nevertheless, it is important to emphasize that *Plagiat.pl* is mainly used by universities to find plagiarism in theses, thus students seems to be a good group to generalize the results to the population of typical users.

6.10 Related work

Most research focuses on generating content for technicians and experts. For example, a project dedicated to helping engineers is presented by *PLANDoc* by McKeown et al. [101]. It creates a summary of telephone network planning based on an output from dedicated planning software. The main purpose of this project is to provide the formal documentation required by auditors and public regulators. Unfortunately, educational purposes are not the main goal here. However, as the authors claim, the resultant description can be used to train new planning engineers.

An attempt to create an online help system was made by Reiter et al., who designed a tool which generates short messages used in an online help system for equipment for testing circuit boards [128]. As in the case of *PLANDoc*, the tool is designed for technical staff only. Moreover, a set of short messages can be considered as a complete user manual.

Another approach is *DRAFTER*, which was designed to create drafts of instructions (i.e. a description which tells you how to perform a task) on the basis of a domain knowledge base [118]. The knowledge base consists of (among other things) *actions, steps, objects, and a set of relations between them* [118]. Data are added manually, but there is a possibility to import GUI mock-ups. The content of instructions is set by a technical writer, who defines the input for a text generator by filling patterns with data from a knowledge base. The paper presents an example of a pattern *[person] schedule [appointment]*, which can be used to construct an input *reader schedule arbitrary appointment*, which can be further used to generate different variants of sentences, e.g. *Schedule the appointment* [118]. There are other tools which challenge the generation of instructions, e.g. *AGILE* [52] and *Isolde* [119]. Although these tools represent an interesting approach, they are insufficient for creating a complete user manual. Moreover, it seems that some of the information, which is provided manually, can be imported from documentation available in a project, especially from acceptance tests.

6.11 Conclusions

The goal of this work was to investigate whether it is possible to automatically generate a user manual that would be no worse than a corresponding handmade manual. This work describes initial research. It is assumed that the generation of a user manual is based on a business case, a software requirements specification (SRS), acceptance tests, and a working application (or GUI mock-ups). It was also assumed that SRS includes functional requirements (defined using use cases), non-functional requirements (defined using the Non-functional Requirement Template), and technical constraints (defined using the Technical Constraints Template).

The structure of the generated user manual is based on a literature study. Two variants of user manuals were proposed:

- *naive*—which can be created using only information found in listed artifacts, and
- *complete*–which requires existing content and generated content (this variant is compliant with ISO Std 26514:2008 [67] and guidelines recommended by Sun Technical Publications [145]).

To generate additional content, two methods were proposed:

- generating requirements concerning the operating environment—on the basis of non-functional requirements and technical constraints, and
- generating exemplary usages—on the basis of acceptance tests, use cases, and a working application (or GUI mock-ups).

An exemplary usage is generated on the basis of acceptance tests. To select the most representative one, two metrics were proposed:

- *widget coverage*—which measures GUI elements referred to by a test case,
- *event coverage*—which measures events (in use cases) checked by a test case.

Each exemplary usage is enriched with descriptions and screenshots (collected while running an acceptance test).

A user manual for a commercial application, *Plagiat.pl*, was generated and evaluated in a controlled experiment using the COCA quality model and the Documentation Evaluation Test. The results show that the quality of the generated manual is no worse than the corresponding commercial handmade user manual for any of the COCA criteria. The number of correct answers measured using DET method was 85% for the generated manual and 83% for the original one. To check whether a generated manual is no worse in other cases, additional experiments are required.

It seems, that the proposed methods can be used to create a tool which can automatically generate a user manual for web applications on the basis of a business case, software requirements specification, acceptance tests, and working software (or GUI mock-ups).

Acknowledgements

This work has been partially supported by the Polish National Science Center based on the decisions DEC-2011/03/N/ST6/03016.

Chapter 7

Conclusions

The aim of this thesis was to investigate the possibility of the automatic generation of user documentation (which comprises user manuals and field explanations) whose quality is no worse than that of the content created by a human. The achieved results confirm that it is possible to generate documentation for web applications with such quality.

More precisely, the following conclusions can be drawn from the research presented in this thesis:

CONCLUSION 1. *The quality of commercial manuals is not very high.*

COMMENT. None of the commercial user manuals used to create the COCA quality profile (Chapter 4) received 100% of *very good* answers on any of the questions used to evaluate quality characteristics. 29% of prospective users gave question Q4 the answer *very good*, and this is the highest result for any question (question Q4 concerns operability, see Table 4.6). Moreover, only 55% of prospective users gave question Q2 the answer *good enough* (this question concerns completeness). When the answer *good enough* and *very good* are aggregated as positive answers, Q2 is the highest scoring question with 85% of positive answers. When it comes to the DET method (see Chapter 4), the percentage of correct answers is between 77% and 87%, and the average value is about 81%. □

CONCLUSION 2. *Using a regular expression as input, one can generate a 3-fold field explanation—which consists of a narrative explanation, diagrams, and a set of examples—whose quality is no worse than the corresponding description written by a human.*

COMMENT. We proposed a number of methods which analyze a regular expression and generate an easy-to-understand explanation (see Chapter 5). The proposed methods were evaluated experimentally by checking 5 exemplary fields commonly used in web applications. In the conducted experiments, field explanations generated by the prototype tool received 84% of correct answers, while those written by humans received between 77% and 78%. □

CONCLUSION 3. *One can automatically generate a user manual on the basis of a business case, a software requirements specification (which includes use cases), acceptance tests, and working software (or GUI mock-ups), with quality no worse than a corresponding user manual created by humans.*

COMMENT. The experiment, based on the commercial system *Plagiat.pl* and the set of generation methods proposed in Chapter 6, shows that the generated manual is no worse than its commercial counterpart. More precisely, in the mentioned experiment, the generated manual was no worse than "hand-made" one for any of the COCA criteria. The DET method has also confirmed that the generated user manual is no worse than the one written by a human: the percentage of correct answers for the generated version was 85%, while for the hand-made it was 83%. □

CONCLUSION 4. *Good quality business case, use cases, and acceptance test scripts can be used to automatically generate user manuals, and thus can contribute to decreasing overall development costs.*

COMMENT. The automatic generation of manuals is not limited to educational objectives only. Berry et al. states that a user manual can be used as a software requirements specification [19]. Thus, a generated user manual can be used as an additional artifact useful from the point of view of quality assurance. □

Appendix A

COCA quality model for user documentation

A.1 Evaluation mandate – an example

ID	EM20130610
Software	ROPS: Registration and evaluation of curricula
Documentation name	User Manual
Documentation version	20130206
Filename	`ROPS-UserManual-20130206.pdf`
Evaluation deadline	12 June 2013
Purpose	Acceptance or rejection of the user documentation
Scope	Whole document
Evaluation approach	Individual review + evaluation form EF20130610

Evaluation grades

Final grades:

- *accept*
- *accept with minor revision* – necessary modifications are very easy to introduce and no other evaluation meeting is necessary
- *accept with major revision* – identified defects are not easy to fix and a new version should go through another evaluation procedure
- *reject* – quality of the submitted documentation is unacceptable and other corrective actions concerning the staff or process of writing must be taken

Standard answers to questions (5-level Lickert):

- *Not at all* (*N* for short)
- *Weak* (*w*)

- *Hard to say* (*?*)
- *Good enough* (*g*)
- *Very good* (*VG*)

Selection of quality questions

Question	Expert	Prosp. user
Completeness		
To what extent does the user documentation cover all the functionality provided by the system with the needed level of detail?	✓	
To what extent does the user documentation provide information which is helpful in deciding whether the system is appropriate for the needs of prospective users?	✓	
To what extent does the user documentation contain information about how to use it with effectiveness and efficiency?		✓
Operability		
To what extent is the user documentation easy to use and helpful when operating the system documented by it?		✓
Correctness		
To what extent does the user documentation provide correct descriptions with the needed degree of precision?	✓	
Appearance		
To what extent is the information contained in the user documentation presented in an aesthetic way?		✓

A.2 Evaluation form for Prospective Users – an example

ID	EM20130610
Software	ROPS: Registration and evaluation of curricula
Documentation name	User Manual
Documentation version	20130206
Filename	`ROPS-UserManual-20130206.pdf`
Evaluation deadline	12 June 2013
Name and surname	Eva Smith

Question	**N**	**w**	**?**	**g**	**VG**
Completeness					
To what extent does the user documentation contain information about how to use it with effectiveness and efficiency?					
Operability					
To what extent is the user documentation easy to use and helpful when operating the system documented by it?					
Appearance					
To what extent is the information contained in the user documentation presented in an aesthetic way?					

Comments and remarks:

Id	Place	Char.	Description	Type	Priority

A.3 Evaluation report – an example

ID	EA20130611
Software	ROPS: Registration and evaluation of curricula
Documentation name	User Manual
Documentation version	20130206
Filename	`ROPS-UserManual-20130206.pdf`
Evaluation deadline	12 June 2013
Evaluation date	11 June 2013
Purpose	Acceptance of rejection of the user documentation
Scope	Whole document
Evaluation approach	Individual review + evaluation form EF20130610

Results

Final grade	**reject**

Question		**N**	**w**	**?**	**g**	**VG**
Completeness						*responsible: Expert (1)*
To what extent does the user documentation cover all the functionality provided by the system with the needed level of detail?	ROPS	0.0%	0.0%	0.0%	0.0%	100.0%
	Profile	3.7%	18.5%	29.6%	44.4%	3.7%
To what extent does the user documentation provide information which is helpful in deciding whether the system is appropriate for the needs of prospective users?	ROPS	0.0%	0.0%	0.0%	100.0%	0.0%
	Profile	0.0%	3.7%	11.1%	55.6%	29.6%
						responsible: Prospective User (3)
To what extent does the user documentation contain information about how to use it with effectiveness and efficiency?	ROPS	0.0%	0.0%	33.3%	66.7%	0.0%
	Profile	6.1%	9.5%	7.4%	50.0%	27.0%
Operability						*responsible: Prospective User (3)*
To what extent is the user documentation easy to use and helpful when operating the system documented by it?	ROPS	33.3%	33.3%	33.3%	0.0%	0.0%
	Profile	1.4%	6.8%	14.9%	48.0%	29.1%
Correctness						*responsible: Expert (1)*
To what extent does the user documentation provide correct descriptions with the needed degree of precision?	ROPS	0.0%	0.0%	0.0%	100.0%	0.0%
	Profile	0.0%	18.5%	25.9%	44.4%	11.1%
Appearance						*responsible: Prospective User (3)*
To what extent is the information contained in the user documentation presented in an aesthetic way?	ROPS	0.0%	0.0%	33.3%	66.7%	0.0%
	Profile	1.4%	12.2%	12.2%	49.3%	25.0%

Comments and remarks

Id	**Place**	**Char.**	**Description**	**Author**	**Type**	**Priority**
1	p. 1	Coml.	Data about user documentation (name, version, etc.) are missing.	E1	missing	major
2	all	Coml.	No page number.	E1	missing	major
3	p. 3	Coml.	Role Guest is not described.	E1	missing	major
4	p. 3	Coml.	Abbr. OEK and KRK are not explained.	E1, P2	missing	major

Evaluation team

Decision Maker	*Jerzy Nawrocki*
Review Leader	*Bartosz Alchimowicz*
Experts (1)	E1 - *John Smith*
Prospective Users (3)	P1 - *Eva Smith*, P2 - *Adam Smith*, P3 - *Peter Smith*

A.4 Evaluation report for profile

Purpose	data collection
Evaluation approach	individual review + evaluation form

Results

	Software	N	w	?	g	VG
Completeness						*responsible: Expert*
To what extent does the user documentation cover all the functionality provided by the system with the needed level of detail?	Plagiarism.pl	0	0	1	1	1
	Deanery.XP	0	0	1	2	0
	Optivum Secr.	0	1	2	0	0
	nSzkoła	0	2	0	1	0
	Secr. DDJ	0	0	1	2	0
	LangSystem	1	1	1	0	0
	SchoolMgr.	0	0	1	2	0
	Hermes	0	1	1	1	0
	E-oceny	0	0	0	3	0
To what extent does the user documentation provide information which is helpful in deciding whether the system is appropriate for the needs of prospective users?	Plagiarism.pl	0	0	0	1	2
	Deanery.XP	0	0	1	2	0
	Optivum Secr.	0	0	1	2	0
	nSzkoła	0	0	1	2	0
	Secr. DDJ	0	0	0	2	1
	LangSystem	0	1	0	1	1
	SchoolMgr.	0	0	0	1	2
	Hermes	0	0	0	3	0
	E-oceny	0	0	0	1	2
						responsible: Prospective User
To what extent does the user documentation contain information about how to use it with effectiveness and efficiency?	Plagiarism.pl	3	1	1	11	0
	Deanery.XP	3	3	1	8	2
	Optivum Secr.	2	4	3	8	0
	nSzkoła	0	1	1	11	3
	Secr. DDJ	0	1	2	6	7
	LangSystem	1	2	1	7	6
	SchoolMgr.	0	2	2	8	5
	Hermes	0	0	0	8	8
	E-oceny	0	0	0	7	9
Operability						*responsible: Prospective User*
To what extent is the user documentation easy to use and helpful when operating the system documented by it?	Plagiarism.pl	0	5	2	7	2
	Deanery.XP	0	0	2	9	6
	Optivum Secr.	2	2	2	6	5
	nSzkoła	0	1	2	10	3
	Secr. DDJ	0	1	4	8	3
	LangSystem	0	0	2	12	3
	SchoolMgr.	0	0	3	7	7
	Hermes	0	0	3	7	6
	E-oceny	0	1	2	5	8
Correctness						*responsible: Expert*

To what extent does the user documentation provide correct descriptions with the needed degree of precision?	Plagiarism.pl	0	0	1	1	1
	Deanery.XP	0	0	0	2	1
	Optivum Secr.	0	2	1	0	0
	nSzkoła	0	1	0	2	0
	Secr. DDJ	0	0	0	2	1
	LangSystem	0	1	2	0	0
	SchoolMgr.	0	0	1	2	0
	Hermes	0	0	1	2	0
	E-oceny	0	1	1	1	0
Appearance						*responsible: Prospective User*
To what extent is the information contained in the user documentation presented in an aesthetic way?	Plagiarism.pl	2	3	2	9	0
	Deanery.XP	0	4	2	10	1
	Optivum Secr.	0	4	1	10	2
	nSzkoła	0	1	0	8	7
	Secr. DDJ	0	2	3	6	5
	LangSystem	0	3	3	6	5
	SchoolMgr.	0	0	4	11	2
	Hermes	0	1	3	5	7
	E-oceny	0	0	0	8	8

Evaluation team

System	Review Leader	Experts	Prospective Users
Plagiarism.pl	1	3	16
Deanery.XP	1	3	17
Optivum Secretariat	1	3	17
nSzkoła	1	3	16
Secretariat DDJ	1	3	16
LangSystem	1	3	17
SchoolManager	1	3	17
Hermes	1	3	16
E-oceny	1	3	16

Appendix B

Compiling software artifacts to generate user manuals

B.1 Project database

Information is stored in a project database as variables. For example, variable `name` contains the name of an application and variable `version` stores the software version.

B.1.1 Business Case

The problem description from *Business Case* is assigned to variable `problem`.

B.1.2 Software requirement Specification

The description of how an application solves a problem presented in *Business Case* is put into variable `scope`.

Functional requirements

Since many forms of use cases are used [33, 60, 77], we proposed a new generic model (see Figure B.1). It is based on the Use Cases Database (UCDB), UCWorkbench, and FUSE [11, 12, 108]. In comparison to the listed models, our version additionally supports the *1)* type of actor (human or external system) and how they use the available data (if an actor reads and/or provides data), and *2)* a list of fields in business objects.

Initially each step contains a string from input data and there are no relationships between use cases, actors, and business objects—a data analysis is required to find them.

Use cases are assigned to variable `ucs`. Further, use cases are accompanied by business objects (`bos`), and a list of actors (`actors`).

A use case (`UseCase`) consists of an `id`, a `title`, a main actor (variable `mainActor`), there are also secondary actors (`secondaryActors`), a `priority`, and a main `scenario` (a list of `steps`).

An actor (`Actor`) consists of an `id`, a `name`, a `type` (whether it is a human or a system) and a `direction` (what an actor does with business objects).

A business object (`BusinessObject`) consists of a `name` and a `description`. It is possible to list and describe all fields which form a given business object.

A step (`Step`) consists of an `id`, the content of a step (`text`), and optional `events`. Variable `mockup` is a reference to GUI mock-ups.

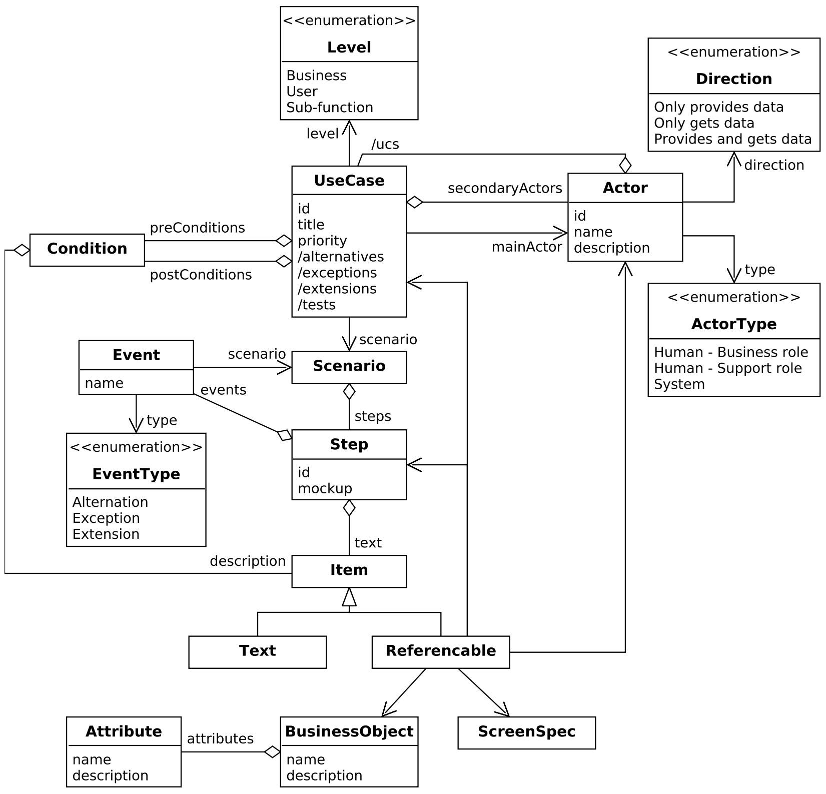


Figure B.1: UML model of functional requirements expressed as use cases

Variables `actors`, `ucs`, and `bos` are collections. To access an element, one can enter its index (e.g., `actors[0]`) or its label (e.g., `actors["Student"]`). Iteration over collections is available as well.

Non-functional requirements and technical constraints

See Appendix B.2

GUI mock-ups

One GUI mock-up (layout of one web page) is represented by one screen defined in ScreenSpec [115]. Each screen consists of a number of components (see class `ScreenSpec` in UML class diagram in Figure B.2, presented together with acceptance tests). A component can be a widget or a collection of components. HTML widgets are represented using class `Simple`. Each widget has an id (in variable `label`), `values`, and a `type`[1]. For example, to define a button with the id *btn* and description *Save*, it is required to use class `Simple` with variable `label` set to `btn`, `values` set to `Save`, and `type` set to `BUTTON`. Collections of widgets are represented by a `Group`, which consists of a list of `components`.

For more details about ScreenSpec and GUI mock-ups please refer to paper by Olek et. al (see [115]).

[1]Typeless elements are achieved by setting `type` to `NONE`.

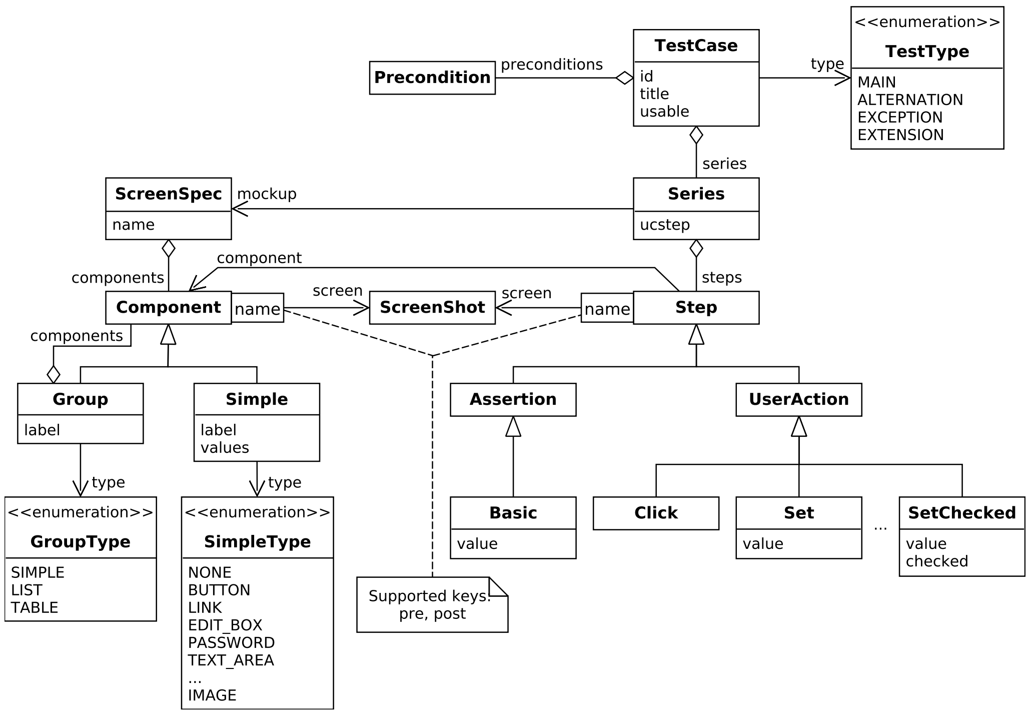


Figure B.2: Simplified UML model of acceptance tests (defined using TDL) and GUI mock-ups (ScreenSpec)

B.1.3 Acceptance tests

The structure of acceptance tests is based on the Test Description Language introduced by Olek et. al [114]. This language allows defining a number of test steps (user actions and/or assertions) and grouping them by use case steps. For example, for step *User provides data about an order* one can provide a number of test steps which interact with a system (one test step fills one field in a form, presses one button, etc.).

UML class diagram of acceptance tests is presented in Figure B.2. Each test case (class `TestCase`) has an `id`, `title`, and a group of steps (`series`). A test step can be an action (`UserAction`) or an assertion[2] (`Assertion`). A test case can be connected with a GUI mock-up[3] (via variable `component` in a `Step`) and a use case (via variable `ucstep` in `Series`). Additionally, each test case can be categorized (by setting its `type`) and selected to be used to generate a user manual example (`usable` set to `True`)—this requires analysis of information stored in a project database. While running a test step it is possible to collect screen shots. One can view a captured web page before and after a test step is executed (`screen["pre"]` and `screen["post"]` respectively).

All test cases are accessible through variable `tests`. Tests dedicated to a particular use case are in variable `tests` of the given use case (see Figure B.1).

For more details about TDL please refer to the publication by Olek et. al (see [114]).

B.1.4 Glossary

A term in the glossary consists of a `name` and a `description`. Terms are accessible via variable `glossary` (which is a collection).

[2]Currently only basic assertions are supported.

[3]This gives access to a type of widget and access to its screen shot.

B.2 Non-functional Requirement Templates and Technical Constraint Templates

Abbreviations:

- NFR—Non-Functional Requirement
- NoRT—Non-functional Requirement Template
- TC—Technical Constraint
- TeCT—Technical Constraints Template
- UM—User Manual

B.2.1 Cover

ID	NoRT32
NoRT	`'The user manual' ['for' <actors:`*actor*`>] 'shall comply with' <template:text> ['and shall be delivered in' <format:text>].`
NFR example	The user manual for Student shall comply with generic template and be delivered in a pdf file.
Parameters	`actors` – a list of actors (each actor needs to be defined in SRS, optional) `template` – a name or a reference to a template for generating of a user manual `format` – the format/type of the target file of a user manual (optional)

UM template	`User manual %`
	`{% if project.nfrs[32] %}%`
	`{% if project.nfrs[32].actors|length == 1 %}%`
	`for {{ project.nfrs[32].actors[0] }}`
	`{% else %}%`
	`the following users:`
	`\begin{itemize}`
	`{% for actor in project.nfrs[32].actors %}`
	`\item {{actor}}`
	`{% endfor %}`
	`\end{itemize}`
	`{% endif %}`
	`{% endif %}`

UM example	User manual for Student
Comments	`length` is a built-in filter which counts the number of elements in a list

B.2.2 Introduction

Web page address

ID	NoRT123
NoRT	`'System shall be available at' <webaddress:url> [ '(' <IP address:text> ')' ].`
NFR example	System shall be available at `http://example.com`.
Parameters	`webaddress` – a URL of the system `ip` – IP address of a web application (optional)

UM template	`{% if project.nfrs[123] %}`
	`\subsection{Address of the web page}`
	`Web page is available at {{ project.nfrs[123].webaddres }}`
	`{% if project.nfrs[123].ip %}({{ project.nfrs[123].ip }}){% endif %}.`
	`{% endif %}`

UM example	The application is available at `http://example.com`.

B.2.3 Requirements concerning operating environment

Environment

ID	NoRT16
NoRT	`'The minimal required amount of resources is:' <resource type:text>',' ('no less then' | 'more than' | 'equal to') <amount and unit:text>.`
NFR example	The minimal required amount of resources is: RAM, no less than 512MB; processor, more than 2.6GHz .
Parameters	`resources` – a list of resources each item in the `resources` list contains the following attributes: `type` – the type of the resource `relation sign` – one symbol from the following list: ==, <, >, <=, >= `amount` – the amount of resources `unit` – the unit of measurement
Preprocessing	Additionally, there is a computed attribute: `similar` – if all resources have the same `relation sign`, the similar attribute is set to this sign, otherwise this attribute is set to `False`
UM example	User's computer should be equipped with a minimum of: - 512 MB RAM memory - 2.6 GHz processor

ID	NoRT29
NoRT	`'The workstation of (<actor:`*actor*`> | user) shall have the following protections:' <protection mechanism:text>.`
NFR example	The workstation of user shall have the following protections: an anti-virus software installed with an up-to-date virus database.
Parameters	`protection mechanizms` – a list of required protections
UM example	It is recommended that the user's computer is protected with an up-to-date anti-virus software installed with an up-to-date virus database.

ID	NoRT66
NoRT	`'The following' <type:text> 'of environments should be supported:' <name:text> [<version/id:text>] [by <vendor:text>].`
NFR example	The following environments should be supported: browser Firefox 17 and newer, Chrome 34 and newer, IE 8 and newer by Microsoft.
Parameters	`type` – the type of the environment, for our scope only Internet browsers are analyzed, hence it is set to *browser* `software` – a list of supported browsers each item in `software` list contains the following details about a browser: `name` – name `version` – version (optional) `later` – `True`, if newer (later) versions are to be supported (optional) `vendor` – vendor (optional)
UM example	To run the application, one of the following web browsers is required: FireFox, version 17 or newer; Chrome, version 34 or newer; Internet Explorer, version 8 or newer.

ID	NoRT124
NoRT	`'The system shall run on displays with' ('minimum' | 'recommended' | 'exactly') <resolutions:text>.`
NFR example	The system shall run on displays with the minimum resolution of 1024x768.
Parameters	`minimum` – minimum screen resolution `recommended` – recommended screen resolution `exact` – exact screen resolution (other attributes are ignored)
UM example	The recommended screen resolution is no less than 1024x768.

ID	TeCT1
TC	`'To use the system one' ('needs to' | 'is recommended to') 'have installed' [<type of environment:text>] <vendor:text> <environment name:text> <version/id:text>, ['or newer'] .`
TC example	To use the system one needs to have Flash Player v.11.0 or newer.
Parameters	`required` – a list of additional required applications `recommended` – a list of additional recommended applications each item in the `required` and the `recommended` lists contains the following details about the application: `name` – its name `version` – its version (optional) `later` – `True`, if later versions are also supported (optional) `vendor` – its vendor (optional)
UM example	It is **required** to additionally install: Flash Player, version 11 or newer.

Knowledge and experience

ID	NoRT58
NoRT	`'To operate the system one needs to possess the following knowledge and skills:' <name:text>, 'certificate of' <exam name:text>.`
NFR example	To operate the system one needs to possess the following knowledge and skills: ECDL.
Parameters	`knowledge` – a list of abilities each item in the `knowledge` list contains the following attributes: `name` – the name of an ability or skill `exam name` – the name of a certificate
UM example	To use the application one needs to know how to use a web browser.

B.3 The user manual generated for the Plagiat.pl web application

This section contains the user manual generated for the commercial application *Plagiat.pl*. The following data were used:

			Artifact		
Component	**Variable**	**Data type**	*Configuration file*	*Business case*	*Software Requirement Specification*
Cover	`name`	Imported	✓		
	`version`	Imported	✓		
Table of contents	N/A				
Conventions	`convs`	Fixed	✓		
Introduction:					
Problem description	`problem`	Imported		✓	
System description	`scope`	Imported			✓
	`actors`	Imported[A)]			✓
	`ucs`	Imported			✓
Web page	`nfrs`	Generated			✓
Requirements concerning operating environment	`nfrs`	Generated			✓
	`adets`	Generated			✓
Information objects	`bos`	Imported			✓
Tasks:					
Actor	`actors`	Imported			✓
Scenarios	`ucs`	Imported			✓
Examples	`examples`	Generated			✓
Glossary	`glossary`	Imported			✓

Notes:
A) Context diagram is generated

Plagiat.pl

User Manual

Version 2012

Contents

The following document contains quotes from *Instrukcja Użytkownika Indywidualnego dla Internetowego Systemu Antyplagiatowego Plagiat.pl* which are not sourced as they appear within the text. This is done to more accurately reflect the user manual format in research conducted at the Poznań University of Technology.

Instrukcja Użytkownika Indywidualnego dla Internetowego Systemu Antyplagiatowego Plagiat.pl is property of Plagiat.pl LLC.

1 Using the manual

The following manual concerns the usage of the *Plagiat.pl* system in its 2012 version.

In order to make the information more understandable to new users of the system, the manual has been divided into several parts. The most important segments are:

- introduction, which describes the program's uses;
- requirements, which outlines necessary conditions for using the program;
- information objects, which aims to acquaint the user with data stored and processed by the system;
- supported goals, which provides step-by-step instructions for using the software to carry out particular tasks.

Certain conventions have been adopted to aid ease of description. Seeing as more than one person can work at the same task, the word *role* is proposed in place of *user.* This term can mean all program users that perform the same, or similar, actions. To make the manual more accessible, all tasks have been categorized according to the role they play within the system (chapter *Tasks*).

2 Introduction

2.1 Problem description

In order to obtain a vocational or academic degree, a tertiary-level student is obliged to submit a diploma project, typically in the form of a written dissertation.

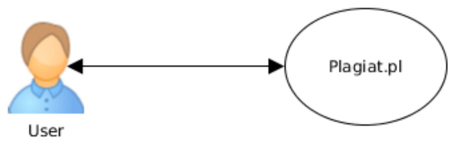
The purpose of a diploma project is to prove that the author (or authors, in group projects) possesses the necessary knowledge, skills and competence within their chosen subject; as such, it should be verified to be an original and independent creation. Unfortunately the multitude of available material makes effective "manual" authentication unfeasible, if not impossible. A computer system capable of automatic analyses of this sort would be a significant advantage.

2.2 System description

The *Plagiat.pl* system identifies borrowings in the analyzed text. The results of this analysis are placed in a *Similarity report*, which contains five *Similarity coefficients.* Each of these indicates the amount of borrowings found in the text, be it from the Internet, the Database of Legal Acts or other sources (e.g. other academic theses).

Plagiat.pl does not determine whether or not the document is plagiarism – the decision must be made by a person with the authority to do so (such as the thesis Supervisor), and the generated *Similarity Report* merely provides supportive information.

The following roles use the system:



Supported goals are described in section *Tasks.*

A typical script for using the system is as follows:

1. *User* creates an account by filling out and submitting the *Registration form* (p. 6)
2. *User* logs into the system (p. 7)
3. *User* adds *Document* to be checked (p. 8)
4. *User* purchases *Tokens* (p. 10)
5. *User* views *Similarity report* (p. 11)

The system also allows:

- Modification of *User* data (p. 12)

2.3 Website

The application is available at `http://www.plagiat.pl`

3 Requirements concering the operating environemnt

Browser
To run the application, one of the following web browsers is required:

- FireFox, version 17 or newer
- Chrome, version 34 or newer
- Internet Explorer, version 8 or newer

Additional software
It is recommended to additionally install:

- Flash Player, version 11 or newer

Screen resolution
The recommended screen resolution is no less than 1024x768.

Hardware requirements
The user's computer should be equipped with, at minimum:

- 512 MB RAM memory
- 2.6 GHz processor

4 Information objects

User data

Information about the *User*.

This object contains the following elements:

- Name
- Surname
- Login (e-mail)
- Registration date
- Position
- Telephone number
- Agreement to receive e-mails with similarity reports
- Agreement to receive e-mails with the bulletin
- Agreement to receive e-mails with the newsletter
- Password – the password must be at least 8 characters long and contain at least 2 digits

Document

The text to be checked (e.g. Master's thesis).

A document to be checked can be submitted in one of two ways:

- pasting the content into the program (copy and paste),
- uploading the content in a file.

Maximum file size is 20 MB.
Accepted file formats: doc, docx, rtf, odt.
Documents up to 500 characters can be checked free of charge.

Code

A series of characters that the *User* receives (in a text message) after using the SMS Premium service to purchase *Token*s.

To receive a *Code*, the user sends a text message to 79068, putting AP.PLGT in the content. The code must then be entered into a form to exchange it for tokens.

A code purchased through SMS Premium expires after 2 weeks.

Warning: if anything other than AP.PLGT makes it into the content of the text message, the fee will be lost.

See also: *Token*

Registration form

A form containing all the data necessary to create an account.

If the *User* wishes not to provide some of the information, the word "ERSATZ" can be entered into the chosen fields as a placeholder.

The minimum required input is an e-mail address. It is necessary to check boxes marking acceptance of the Terms and Conditions, becoming acquainted with the Protection of Personal Data policy and agreement to processing personal information.

This object includes the following elements:

- Name
- Surname
- E-mail address – required
- Firm
- Telephone number
- Agreement to receive the bulletin
- Request for information about the *Plagiat.pl* system
- Declaration of agreement to Terms and Conditions – required
- Agreement to processing personal data as well as the tenets of the Protection of Personal Data policy – required
- Agreement to process personal data for commercial purposes

Document submission form

Form used to submit text for analysis.

The object possesses the following elements:

- Upload method (File Upload or Copy and Paste)
- *Document specification*

Document list

List containing the results of anti-plagiarism analyses. It is the *User*'s default view.

This object possesses the following elements:

- Title
- Similarity coefficient 1 – see Glossary
- Similarity coefficient 2 – see Glossary
- Similarity coefficient 3 – see Glossary
- Similarity coefficient 4 – see Glossary
- Similarity coefficient 5 – see Glossary
- Status – whether or not the analysis is completed and the *Similarity report* available
- Similarity report

Document specification

This object possesses the following elements:

- Author
- Title
- Skip the following websites – list of Internet sites to be ignored during the document analysis
- Text

Similarity report

A *Similarity report* provides information about the borrowings identified in the analyzed text.

The waiting period for a *Similarity report* is usually under 24 hours.

The user is notified about the availability of the report (after the document analysis is completed) via e-mail.

Plagiat.pl informs of similarities by use of the following:

- green color – borrowings from Internet sources
- blue background – borrowings from the Database of Legal Acts

The report comes in two varieties:

- *Full similarity report*
- *Short similarity report*

Short similarity report

The short version of the report contains *Similarity coefficients (1-5)* and a list of similar documents.

This object possesses the following elements:

- Similarity coefficient 1 – see Glossary
- Similarity coefficient 2 – see Glossary
- Similarity coefficient 3 – see Glossary
- Similarity coefficient 4 – see Glossary
- Similarity coefficient 5 – see Glossary
- Documents containing similar fragments: from the *Database of Legal Acts*
- Documents containing similar fragments: from the Internet

See also: *Similarity report*

Full similarity report

The full version of the similarity report contains *Similarity coefficients (1-5)*, a list of similar documents, and the text of the document with marked fragments which were found in Internet sources and the *Database of Legal Acts*.

This object possesses the following elements:

- Title – the title of the text
- Author – the author of the document
- Date of the report – date and time of generating the report
- Similarity coefficient 1 – see Glossary
- Similarity coefficient 2 – see Glossary
- Similarity coefficient 3 – see Glossary
- Similarity coefficient 4 – see Glossary
- Similarity coefficient 5 – see Glossary
- Phrase length for Similarity coefficient 2
- Number of words
- Number of characters
- Skipped URL addresses – a list of website addresses
- Longest fragments identified as similar
- Documents containing similar fragments: from the Database of Legal Acts
- Documents containing similar fragments: from the Internet

See also: *Similarity report*

Terms and Conditions

Rules and regulations of using the *Plagiat.pl* system.

Software such as *Adobe Reader* is necessary to view the *Terms and Conditions*.

Token

Tokens are used to pay for text analyses. One token allows for checking up to 20,000 characters of text. Checking a further fragment of text, up to 20,000 characters, requires spending an additional token.

Information about the required number of tokens appears a document is submitted for analysis. One page of text typically contains around 2000 characters, so one token should allow checking about 10 pages.

Tokens can be purchased using several payment methods: credit card, DOTPAY, PayPal, SMS Premium code, postal money order and bank transfer.

Documents up to 500 characters long can be checked free of charge.

5 Tasks

User

A user is the person submitting documents to analyze and check for unauthorized borrowings.

☞ Register in the system

The script of performing this activity is as follows:

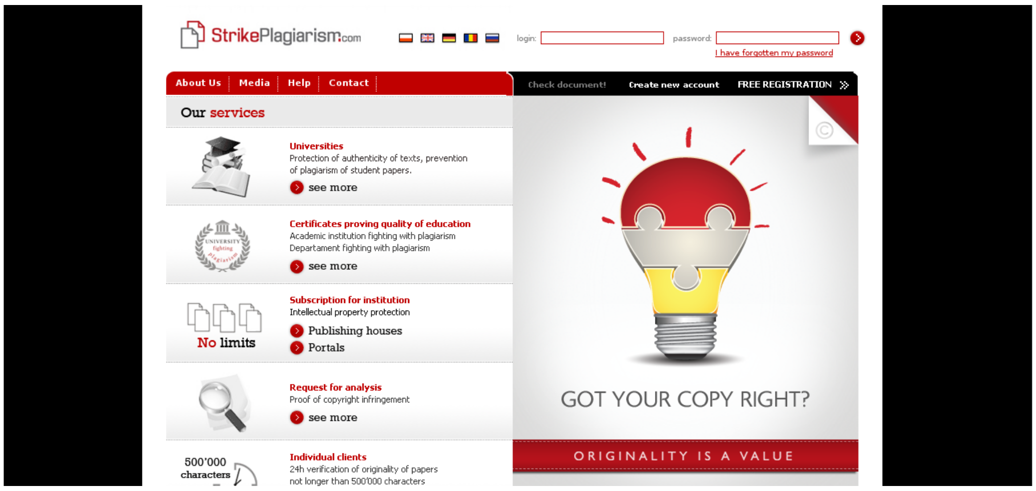
1. *User* chooses the account creation option
2. *System* presents the *Registration form*
3. *User* fills in all necessary information and submits the form
4. *System* informs of successful account creation
5. *System* sends a message with the login details

The following exceptions may occur during this activity:
In step 3:
Terms and Conditions not accepted
1. *System* informs of the requirement to accept the *Terms and Conditions*, necessary to create an account
2. Go to step 3

In step 3:
Data in form incomplete
1. *System* informs of the requirement to provide all necessary information or use the placeholder ERSATZ
2. Go to step 3

✓ Example

- The browser should look like this:



Universities
Protection of authenticity of texts, prevention of plagiarism of student papers.

Certificates proving quality of education
Academic institution fighting with plagiarism
Departament fighting with plagiarism

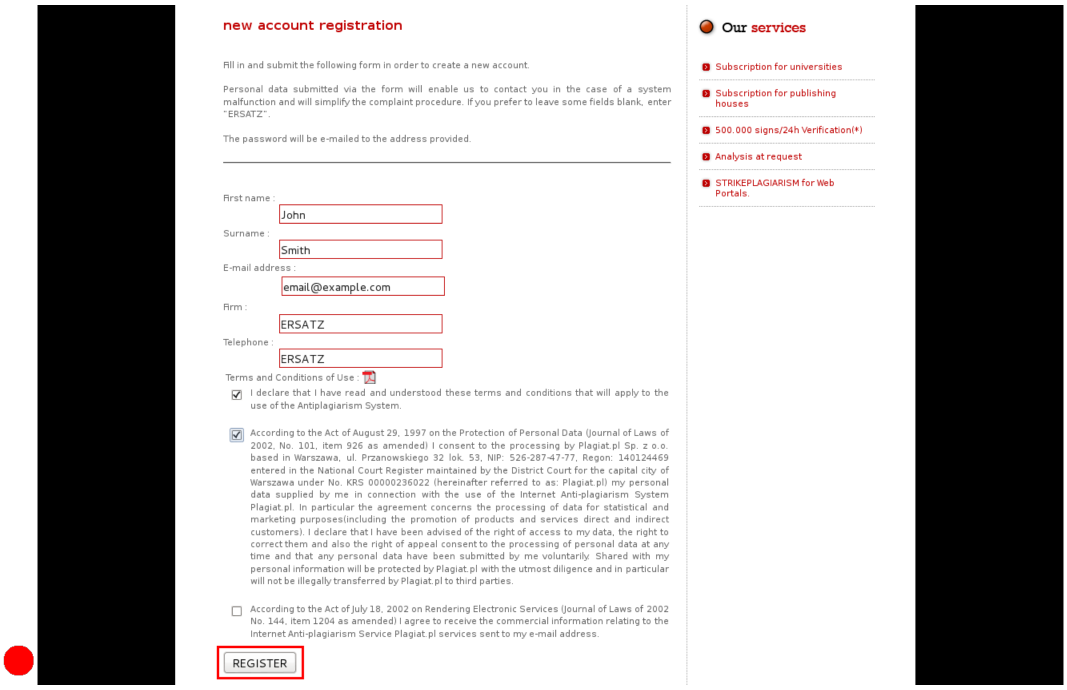
Subscription for institution
Intellectual property protection
Publishing houses
Portals

Request for analysis
Proof of copyright infringement

Individual clients
24h verification of originality of papers not longer than 500'000 characters

Click **Create new account**

- *System* presents the *Registration form*



new account registration

Fill in and submit the following form in order to create a new account.

Personal data submitted via the form will enable us to contact you in the case of a system malfunction and will simplify the complaint procedure. If you prefer to leave some fields blank, enter "ERSATZ".

The password will be e-mailed to the address provided.

First name :	John
Surname :	Smith
E-mail address :	email@example.com
Firm :	ERSATZ
Telephone :	ERSATZ

Terms and Conditions of Use :

☑ I declare that I have read and understood these terms and conditions that will apply to the use of the Antiplagiarism System.

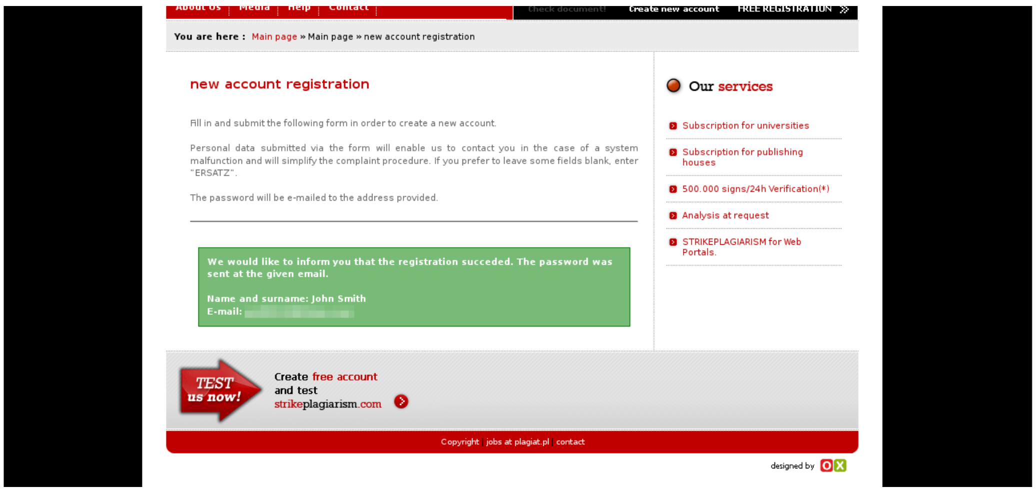
☑ According to the Act of August 29, 1997 on the Protection of Personal Data (Journal of Laws of 2002, No. 101, item 926 as amended) I consent to the processing by Plagiat.pl Sp. z o.o. based in Warszawa, ul. Przanowskiego 32 lok. 53, NIP: 526-287-47-77, Regon: 140124469 entered in the National Court Register maintained by the District Court for the capital city of Warszawa under No. KRS 00000236022 (hereinafter referred to as: Plagiat.pl) my personal data supplied by me in connection with the use of the Internet Anti-plagiarism System Plagiat.pl. In particular the agreement concerns the processing of data for statistical and marketing purposes(including the promotion of products and services direct and indirect customers). I declare that I have been advised of the right of access to my data, the right to correct them and also the right of appeal consent to the processing of personal data at any time and that any personal data have been submitted by me voluntarily. Shared with my personal information will be protected by Plagiat.pl with the utmost diligence and in particular will not be illegally transferred by Plagiat.pl to third parties.

☐ According to the Act of July 18, 2002 on Rendering Electronic Services (Journal of Laws of 2002 No. 144, item 1204 as amended) I agree to receive the commercial information relating to the Internet Anti-plagiarism Service Plagiat.pl services sent to my e-mail address.

REGISTER
Our services
Subscription for universities
Subscription for publishing houses
500.000 signs/24h Verification(*)
Analysis at request
STRIKEPLAGIARISM for Web Portals.

Fill in the necessary information (an example is provided in the illustration) and confirm it by clicking REGISTER.

- *System* informs of successful account creation



You are here : Main page » Main page » new account registration

new account registration

Fill in and submit the following form in order to create a new account.

Personal data submitted via the form will enable us to contact you in the case of a system malfunction and will simplify the complaint procedure. If you prefer to leave some fields blank, enter "ERSATZ".

The password will be e-mailed to the address provided.

We would like to inform you that the registration succeded. The password was sent at the given email.

Name and surname: John Smith
E-mail: [redacted]

Our services
Subscription for universities
Subscription for publishing houses
500.000 signs/24h Verification(*)
Analysis at request
STRIKEPLAGIARISM for Web Portals.
TEST us now!
Create free account and test strikeplagiarism.com
Copyright | jobs at plagiat.pl | contact
designed by OX

- *System* sends a message with the login details

☞ Login to the system

Before proceeding, make sure you have completed:

- Creating an account (p. 6)

The script of performing this activity is as follows:

1. *User* chooses the log in option
2. *System* presents the *Log in form*
3. *User* provides and confirms log in data
4. *System* presents a list of documents submitted for analysis

The following exceptions may occur during this activity:

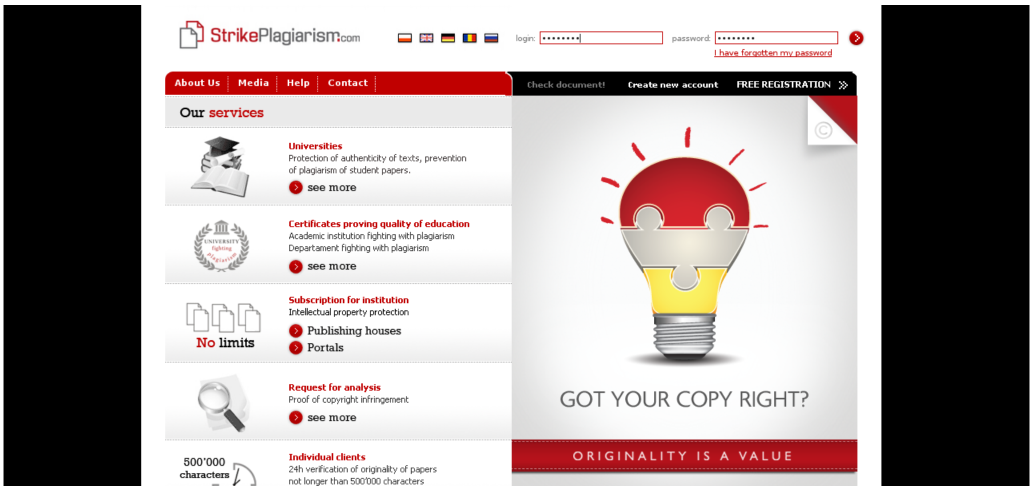
In step 3:

Provided information is incomplete

1. *System* informs of the requirement to provide complete information
2. Go to step 3

✓ **Example**

- The browser should look like this:



StrikePlagiarism.com

login: •••••••• password: ••••••••

I have forgotten my password

About Us | Media | Help | Contact

Check document! Create new account FREE REGISTRATION

Our services

Universities
Protection of authenticity of texts, prevention of plagiarism of student papers.
see more

Certificates proving quality of education
Academic institution fighting with plagiarism
Departament fighting with plagiarism
see more

No limits

Subscription for institution
Intellectual property protection
Publishing houses
Portals

Request for analysis
Proof of copyright infringement
see more

500'000 characters

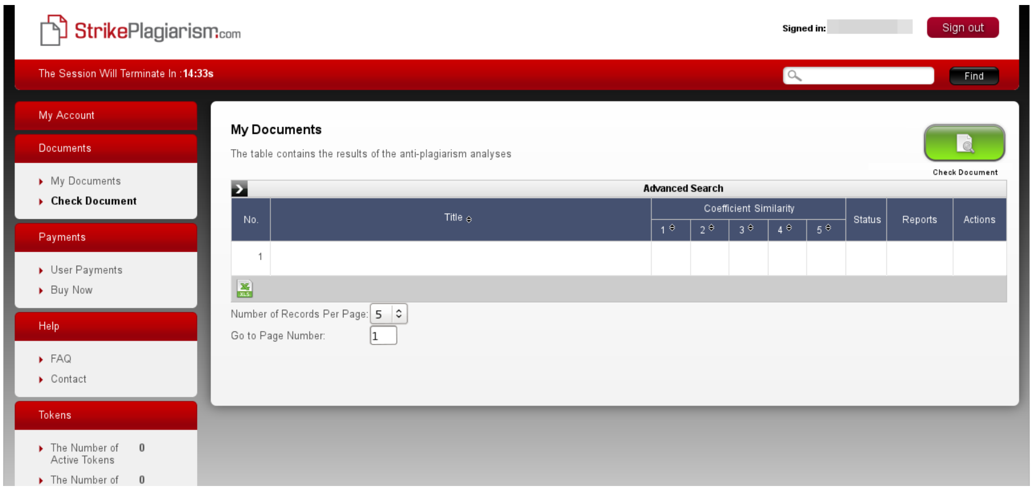
Individual clients
24h verification of originality of papers
not longer than 500'000 characters

GOT YOUR COPY RIGHT?

ORIGINALITY IS A VALUE

Type user information into the login and password fields (example data in the illustration is obscured), then click .

- *System* presents a list of documents submitted for analysis



StrikePlagiarism.com

Signed in: [redacted] Sign out

The Session Will Terminate In :**14:33s** Find

My Account

Documents
- My Documents
- **Check Document**

Payments
- User Payments
- Buy Now

Help
- FAQ
- Contact

Tokens
- The Number of Active Tokens 0
- The Number of 0

My Documents

The table contains the results of the anti-plagiarism analyses

Check Document

Advanced Search

No.	Title	Coefficient Similarity					Status	Reports	Actions
		1	2	3	4	5			
1									

Number of Records Per Page: 5

Go to Page Number: 1

Check a document

Before proceeding, make sure you have completed:

- Logging into the system (p. 7)

The script of performing this activity is as follows:

1. *User* chooses the upload document for analysis option
2. *System* presents the *Document upload form*
3. *User* chooses a document upload method (copy and paste or file upload)
4. *System* presents the *Document specification form*
5. *User* provides and submits data
6. *System* presents a list of *Documents* to be checked

7. *User* chooses *Documents* to be checked
8. *System* informs of initiating the analysis of a chosen *Document*
9. *System* informs of completing the analysis of a chosen *Document*

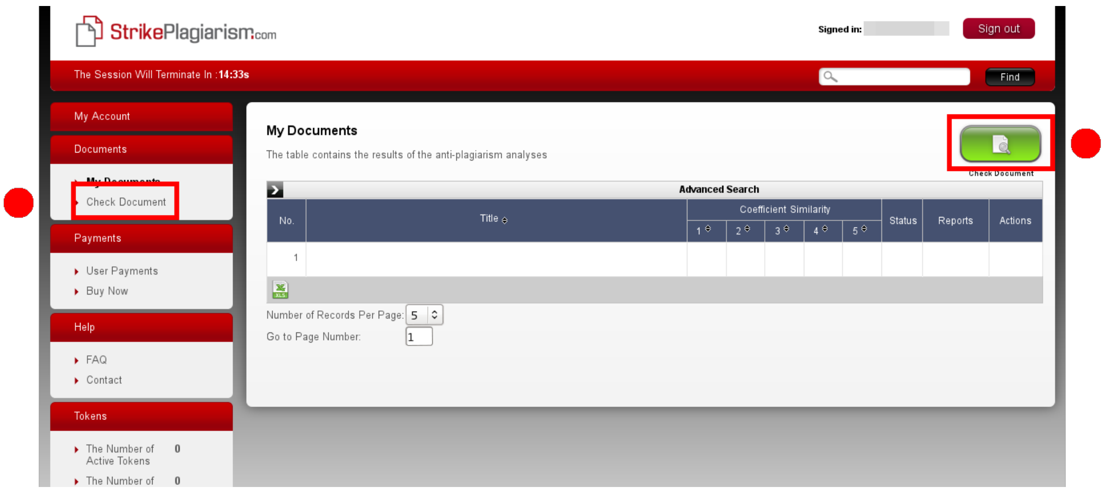
The following exceptions may occur during this activity:
In step 5:
File size exceeded
1. *System* informs that a document exceeds the maximum file size
2. Go to step 5

In step 7:
Insufficient tokens
1. *System* informs that the user lacks tokens necessary to check *Document*
2. *User* purchases additional *Tokens* (see p. 10)
3. Go to step 7

In step 7:
User decides to change Document check order
1. *User* removes all *Documents* from the list
2. *User* re-uploads all *Documents* in the appropriate order
3. Go to step 8

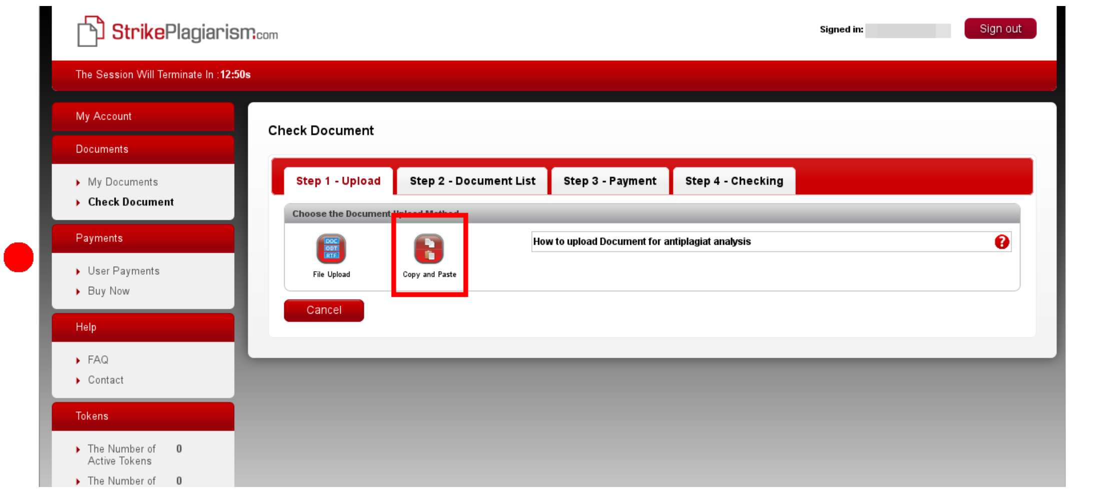
✓ **Example**

- The browser should look like this:



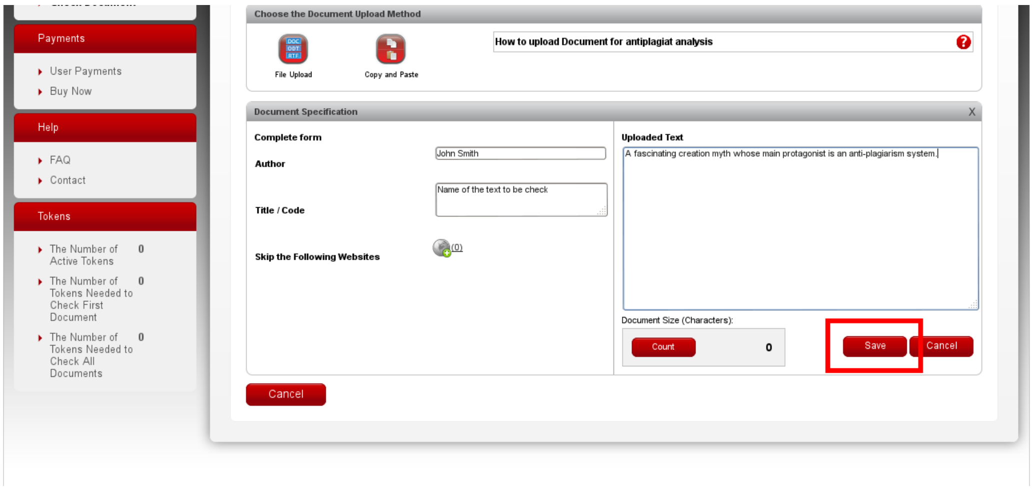
Choose the upload document option by clicking Check Document.

- *System* presents the *Document upload form*



Choose the copy and paste method by clicking Copy and Paste.

- *System* presents the *Document specification form*


Payments
User Payments
Buy Now
Help
FAQ
Contact
Tokens
The Number of Active Tokens 0
The Number of Tokens Needed to Check First Document 0
The Number of Tokens Needed to Check All Documents 0
Choose the Document Upload Method
File Upload
Copy and Paste
How to upload Document for antiplagiat analysis
Document Specification
X
Count
Save
Cancel
Cancel

Complete form

Author	John Smith
Title / Code	Name of the text to be check
Skip the Following Websites	(0)

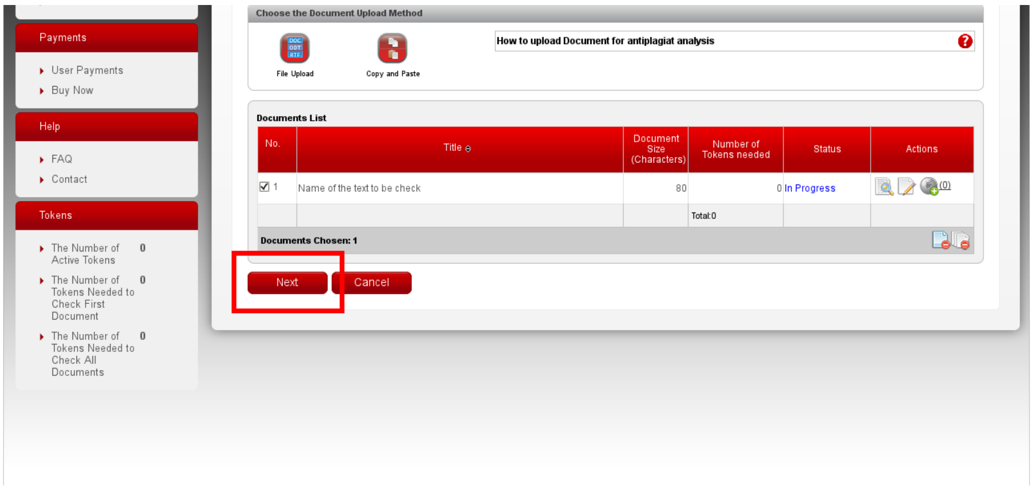
Uploaded Text

A fascinating creation myth whose main protagonist is an anti-plagiarism system.

Document Size (Characters): 0

Fill in the necessary information (an example is provided in the above illustration) and confirm it by clicking Save.

- *System* presents the list of *Documents* to be checked


Payments
User Payments
Buy Now
Help
FAQ
Contact
Tokens
The Number of Active Tokens 0
The Number of Tokens Needed to Check First Document 0
The Number of Tokens Needed to Check All Documents 0
Choose the Document Upload Method
File Upload
Copy and Paste
How to upload Document for antiplagiat analysis
Next
Cancel

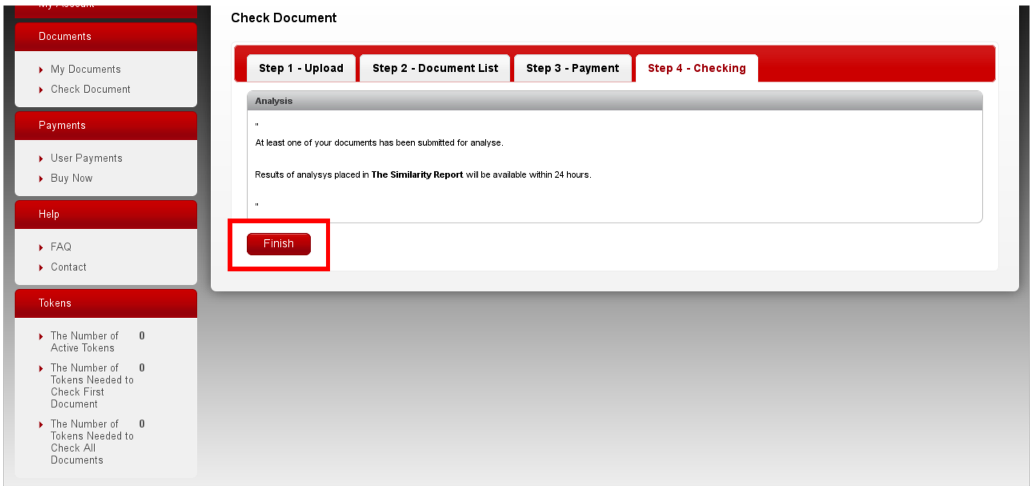
Documents List

No.	Title	Document Size (Characters)	Number of Tokens needed	Status	Actions
☑ 1	Name of the text to be check	80	0	In Progress	(0)
			Total:0		

Documents Chosen: 1

Choose an element and click Next.

- *System* informs of initiating the analysis of a chosen *Document*



Check Document

Analysis

"

At least one of your documents has been submitted for analyse.

Results of analysys placed in **The Similarity Report** will be available within 24 hours.

"

Click Finish.

- *System* informs of completing the analysis of a chosen *Document*

☞ **Purchase tokens**

Before proceeding, make sure you have completed:

- Logging into the system (p. 7)

The script of performing this activity is as follows:

1. *User* chooses the token purchase option
2. *System* presents the *Token purchase form*
3. *User* chooses a payment method
4. *User* chooses tokens to purchase and confirms the choice
5. *System* requests payment
6. *User* carries out payment
7. *System* informs that tokens are available

This activity may also be performed differently:
In step 3:
User chooses SMS Premium

1. *System* provides text message content and the number to send it to
2. *User* sends text message
3. *User* receives text message with *Code*
4. *User* enters *Code*
5. *System* informs of additional *Token*s
6. Finish

☞ **View a similarity report**

Before proceeding, make sure you have completed:

- Checking a document (p. 8)

The script of performing this activity is as follows:

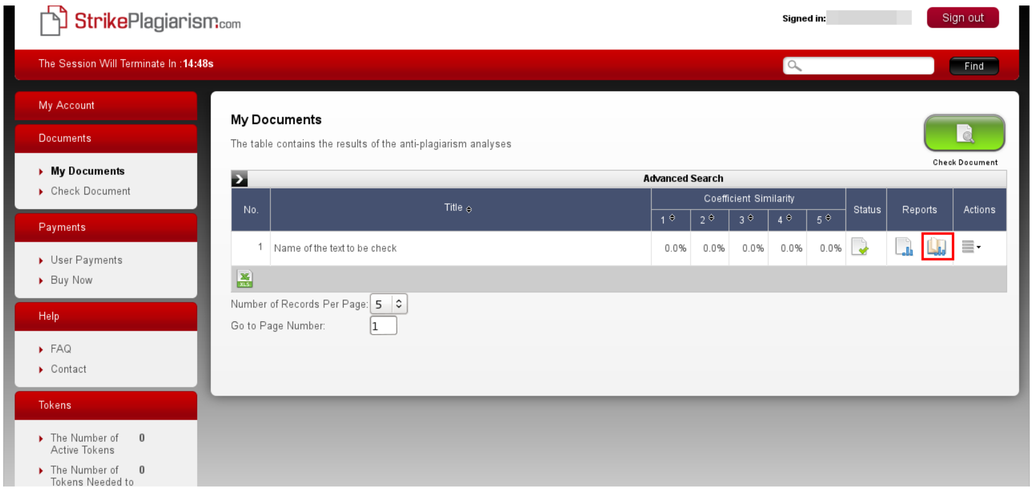
1. *User* chooses the option to view the *Document list*
2. *System* presents the *Document list*
3. *User* chooses *Full similarity report* for a chosen *Document*
4. *System* presents the *Full similarity report*
5. *User* views the *Full similarity report*

This activity may also be performed differently:
In step 3:
User chooses Short similarity report

1. *User* chooses *Short similarity report* for a chosen *Document*
2. *System* presents the *Short similarity report*
3. *User* views the *Short similarity report*
4. Finish

✓ **Example**

- The browser should look like this:



StrikePlagiarism.com

Signed in: Sign out

The Session Will Terminate In : **14:48s**

Find

My Account

Documents

- **My Documents**
- Check Document

Payments

- User Payments
- Buy Now

Help

- FAQ
- Contact

Tokens

- The Number of Active Tokens 0
- The Number of Tokens Needed to 0

My Documents

The table contains the results of the anti-plagiarism analyses

Check Document

Advanced Search

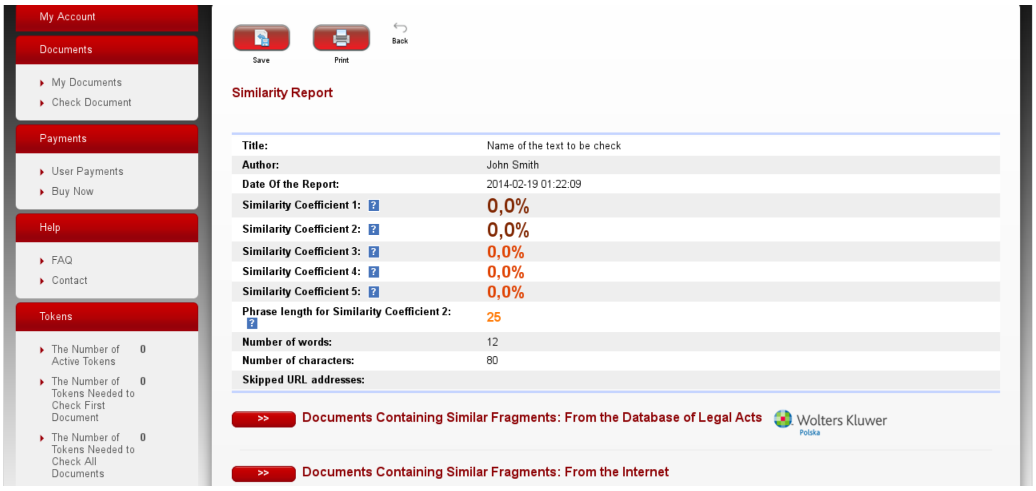
No.	Title	Coefficient Similarity 1	2	3	4	5	Status	Reports	Actions
1	Name of the text to be check	0.0%	0.0%	0.0%	0.0%	0.0%			

Number of Records Per Page: 5

Go to Page Number: 1

Click

- *System* presents the *Full similarity report*



My Account

Documents

- My Documents
- Check Document

Payments

- User Payments
- Buy Now

Help

- FAQ
- Contact

Tokens

- The Number of Active Tokens 0
- The Number of Tokens Needed to Check First Document 0
- The Number of Tokens Needed to Check All Documents 0

Save Print Back

Similarity Report

Title:	Name of the text to be check
Author:	John Smith
Date Of the Report:	2014-02-19 01:22:09
Similarity Coefficient 1: ?	0,0%
Similarity Coefficient 2: ?	0,0%
Similarity Coefficient 3: ?	0,0%
Similarity Coefficient 4: ?	0,0%
Similarity Coefficient 5: ?	0,0%
Phrase length for Similarity Coefficient 2: ?	25
Number of words:	12
Number of characters:	80
Skipped URL addresses:	

>> Documents Containing Similar Fragments: From the Database of Legal Acts Wolters Kluwer Polska

>> Documents Containing Similar Fragments: From the Internet

☞ **Update User data**

Before proceeding, make sure you have completed:

- Logging into the system (p. 7)

The script of performing this activity is as follows:

1. *User* chooses the modify *User data* option
2. *System* presents the *User data*
3. *User* chooses the edit *User data* option
4. *System* enables the modification of *User data*
5. *User* enters and confirms changes
6. *System* saves the changes

The following exceptions may occur during this activity:
In step 3:

Provided information is incomplete

1. *System* informs of the requirement to provide complete information
2. Go to step 5.

6 Glossary

? Alert

Information within the *Similarity report* indicating the presence of non-Latin characters. The *Alert* is meant to bring possible unwarranted use of non-Latin characters to the supervisor's attention, as they may be an attempt to falsify coefficient values in the *Similarity report. Documents* with an *Alert* are highlighted yellow on the *Document list*, and the corresponding *Similarity report* is marked with an exclamation point.

? Similarity report

A document generated by the *Plagiat.pl* system, containing information about borrowings identified in the analyzed text.

? Similarity coefficient 1

Value (expressed in percents) expressing the ratio of borrowings found in Internet sources, consisting of at least five words. Exceeding the accepted values of *Similarity coefficient 1* may indicate overuse of borrowings (content authored by other persons). Taking into account the fact that many fixed phrases consisting of five or more words are commonly in use, exceeding the value of *Similarity coefficient 1* can only be taken as a general pointer to the possibility of copied material, and as a rule requires further verification by an authorized person (e.g. supervisor).

The limit/maximum value of *Similarity coefficient 1* recommended by Plagiat.pl is 50%.

? Similarity coefficient 2

Value (expressed in percents) expressing the ratio of borrowings found in Internet sources, consisting of at least twenty five words. Exceeding the accepted values of *Similarity coefficient 2* is a strong indication of the student's overuse of unauthorized borrowings. Identical phrases of over 25 words are practically nonexistent in common language, and exposing them in a document is reliable evidence of borrowing. Every instance of an exposed borrowing requires verification by an authorized person (e.g. supervisor), as it can also be a valid reference to other authors (e.g. in a properly marked quote).

The limit/maximum value of *Similarity coefficient 2* recommended by Plagiat.pl is 5%.

? Similarity coefficient 3

A percentile value calculated analogously to *Similarity coefficient 1*, but also including fragments found by the *Plagiat.pl* system in the *Database of Legal Acts.*

? Similarity coefficient 4

A percentile value calculated analogously to *Similarity coefficient 2*, but also including all those phrases of 25 words or more found by the *Plagiat.pl* system in the *Database of Legal Acts.*

? Similarity coefficient 5

A percentile value indicating what portion of the analyzed document consists purely of legal phrases consisting of 8 words or more found in the *Database of Legal Acts.*

B.4 Evaluation mandate

Software	Plagiat.pl
Documentation name	User Manual
Documentation version	20140609
Filename	`DU-20140609-meet-du-example-plagiat.pl.pdf`
Evaluation deadline	25 June 2014
Purpose	Data collection
Scope	Whole document
Evaluation approach	Evaluation form EF20140609

Evaluation grades

Standard answers to questions (5-level Lickert): *Not at all* (*N* for short); *Weak* (*w*); *Hard to say* (*?*); *Good enough* (*g*); *Very good* (*VG*).

Selection of quality questions

Id	**Question**	**Expert**	**Prospective user**
Completeness			
Q1	To what extent does the user documentation cover all the functionality provided by the system with the needed level of detail?	✓	
Q2	To what extent does the user documentation provide information which is helpful in deciding whether the system is appropriate for the needs of prospective users?	✓	
Q3	To what extent does the user documentation contain information about how to use it with effectiveness and efficiency?		✓
Operability			
Q4	To what extent is the user documentation easy to use and helpful when operating the system documented by it?		✓
Correctness			
Q5	To what extent does the user documentation provide correct descriptions with the needed degree of precision?	✓	
Appearance			
Q6	To what extent is the information contained in the user documentation presented in an aesthetic way?		✓

B.5 Evaluation form for Prospective User (simplified)

ID	EF20140609
Software	Plagiat.pl
Documentation name	User Manual
Documentation version	20140609
Filename	`DU-20140609-meet-du-example-plagiat.pl.pdf`
Evaluation deadline	25 June 2014

COCA

Question	N	w	?	g	VG
Completeness					
To what extent does the user documentation contain information about how to use it with effectiveness and efficiency?					
Operability					
To what extent is the user documentation easy to use and helpful when operating the system documented by it?					
Appearance					
To what extent is the information contained in the user documentation presented in an aesthetic way?					

DET

One examplary question. Originally there are 29 questions in Polish language.

Question no 2	
The following items are included into a similarity report:	
Choose one of the proposed answers:	*Correct?*
A) Info about whether a given document is plagiarised	
B) Similarity coefficients and a list of similar documents	
C) Similarity coefficients, a list of similar documents and whether a given document is plagiarised	
D) Similarity coefficients, a list of similar documents and fragments of the document which have been found in another document	
The answer is in the user documentation on page:	
I could not find the answer:	

B.6 Evaluation report

Purpose	data collection
Evaluation approach	evaluation form

Results

COCA

Id	Characteristics and the associated question	N	w	?	g	VG
Completeness						*responsible: Expert*
Q1	To what extent does the user documentation cover all the functionality provided by the system with the needed level of detail?	0	0	1	1	1
Q2	To what extent does the user documentation provide information which is helpful in deciding whether the system is appropriate for the needs of prospective users?	0	0	0	1	2
						responsible: Prospective User
Q3	To what extent does the user documentation contain information about how to use it with effectiveness and efficiency?	0	2	2	12	0
Operability						*responsible: Prospective User*
Q4	To what extent is the user documentation easy to use and helpful when operating the system documented by it?	0	0	4	10	2
Correctness						*responsible: Expert*
Q5	To what extent does the user documentation provide correct descriptions with the needed degree of precision?	0	0	1	1	1
Appearance						*responsible: Prospective User*
Q6	To what extent is the information contained in the user documentation presented in an aesthetic way?	0	2	1	8	5

DET

Average time of searching an answer	42 min
Percentage of correct answers	85.13%

Evaluation team

System	Review Leader	Experts	Prospective Users
Plagiat.pl - generated	1	3	16

Appendix C

Generating Syntax Diagrams from Regular Expressions

Preface

This appendix contains the paper: *Bartosz Alchimowicz and Jerzy Nawrocki:* Generating Syntax Diagrams from Regular Expressions*, Foundations of Computing and Decision Sciences, 36(2), pp. 81–97, 2011.* My contribution to this paper included the following tasks: *1)* co-design of visual representation; *2)* design and implementation of the prototype tool; *3)* early evaluation.

The goal of this appendix is to present how syntax diagrams are generated on the basis of regular expressions.

C.1 Introduction

Most of web applications require an input data for proper operation. Those data have often a form of a string that users have to enter into a field. But some strings are meaningless as they are syntactically incorrect. In some cases it is not obvious why a given string is incorrect. A well written user-manual with an explanation of the fields might be helpful in such a case, but so difficult fields do not happen frequently, thus user-manual writers often skip explanation of the fields. This approach makes end-users left without any help and they must solve such problems on their own. A solution might be to supplement user-manual with required explanation, but this requires additional time and costs. Another solution could be to present regular expression used for data validation. Unfortunately regular expressions have a form similar to source code and they may be incomprehensible to end-users. An alternative solution is to automatically generate such an explanation from a regular expression. Hence a question arise, if it is possible to automatically generate a user-friendly explanation of a field syntax.

An example of a tool created to explain regular expression is YAPE [144]. Unfortunately, since this tool is created for software developers it presents only a description of meta-characters used in a regular expression, not the strings themselves. For IT-layman such a description is (almost) useless. Another solution is presented by Ranta [123]. It uses a Xerox Finite State Tool to generate a verbal explanation of a regular expressions. The weakness of the tool is that an explanation is limited to just verbal description - no visual description nor examples are generated.

The paper is organized as follows. In Section C.2 a proposition of an automatic explanation system is presented. Section C.3 describes regular expressions. Next, in Section C.4, a proposition of a visual representation is presented, by describing syntax diagrams and augmented extensions. Then, in Section C.5, generation process of a field explanation is outlined. In Sections C.6 an early evaluation is described. Finally, in Section C.7, the most important findings are discussed.

C.2 Overview of the proposed automatic explanation system

Figure C.1 presents a schema of the proposed automatic explanation system. The input consists of two items: a regular expression used by a programmer for string validation and a field name. As an output one obtains:

- verbal explanation (similar to one provided by Ranta),
- examples,
- visual representation.

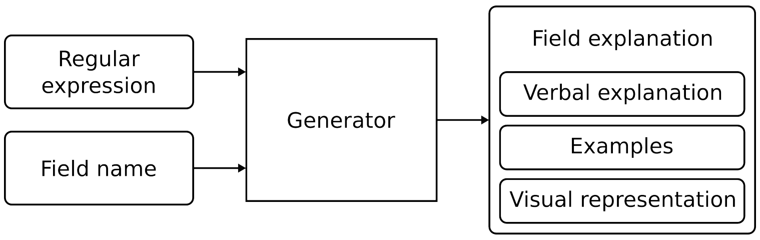


Figure C.1: Black box schema of the proposed automatic explanation system

Verbal explanation provides a description of a field in a natural language. It is the most popular way of knowledge representation and it is widely used in user manuals.

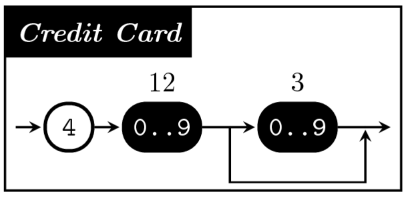
This description is supplemented by a number of examples, presenting correct and incorrect strings. The third element of a field explanation is a visual representation of all the correct strings as syntax diagrams. It seems that such diagrams are a natural choice for presenting syntax of any kinds of strings, including strings entered by users into fields of web applications. Other possible option could be UML, but it is too complicated and could overwhelm prospective end-users – for this reason it has been rejected. All three parts, i.e. verbal explanation, examples, and syntax diagrams form so-called three-part explanation of the language defined by a regular expression.

A simple example is presented in Figure C.2 and C.3. Syntax of a field called Credit Card, is represented as a regular expression (see Figure C.2) and on this basis a field explanation is generated (see Figure C.3).

```
CreditCard = ^4[0-9]{12}([0-9]{3})?$
```

Figure C.2: Input to the generator

Credit Card is described in following diagram:



It consists of digit 4, followed by 12 digits and followed by optional three digits.

Example	Correct?
4056324648328	Yes
4295324322567	Yes
4056324648328123	Yes
056324648328	No (absence of digit 4)
40566236489281234	No (too long)

Figure C.3: Field explanation

The three-part explanation starts with a short introduction, which contains a field name. This is followed by a syntax diagram. Next, one can found a verbal explanation of the elements used in the diagram. At the end there is a table with examples of correct and incorrect input. In case of incorrect string a short justification is given in brackets. For the sake of readability names of fields (or subfields) are in bold.

In the subsequent sections a focus will be on presenting how to generate syntax diagrams from regular expressions.

C.3 Describing field syntax with regular expressions

Regular expressions have been invented by Stephan Kleene [88] and they have become quite popular. They have been incorporated into some programming languages (e.g. Perl [48]) and compiler generators, like Lex [93]. Later on they have been standardized by POSIX [147]. It was decided to use the Lex version of regular expressions. Consequently, meta-characters presented in Table C.1 are used.

Regular expressions can contain also names of other regular expressions. The resulting regular expression is obtained by superposition, i.e. by replacing a name with a regular expression corresponding to it. Here is a simple example. Assume the following definitions are given:

```
name = [a-zA-Z]+
ListOfNames = {name}(, {name})*
```

Then the `ListOfNames` is equivalent to the following regular expression:

```
[a-zA-Z]+(, [a-zA-Z]+)*
```

Table C.1: Meta-characters used by the explanation system

Meta character	**Description**
`.`	any character
`[ ]`	character class
`[^ ]`	exclusive character class
`^`	start of a line
`$`	end of a line
`( )`	groups
`?`	optional
`*`	optional sequence
`+`	non-empty sequence
`{m}`	sequence consisting of `m` elements
`{m,}`	sequence consisting of at least `m` elements
`{m,n}`	sequence consisting of `m` to `n` elements
`|`	alternative

C.4 Explaining regular languages with syntax diagrams

C.4.1 Classical syntax diagrams

Syntax diagrams are known since long ago. They have been used in 1973 by Niklaus Wirth to described syntax diagrams of the Pascal programming language [154]. We use them to graphically represent syntax of a field of web application.

A syntax diagram is a directed graph. Nodes of such a graph can be terminal (representing simple strings) or non-terminal (those represent more complicated strings which are described with a separate regular expression).

Figure C.4 contains a syntax diagram, that presents simple regular expression `fl(i|a)p` (it describes two words: `flip` or `flap`). All nodes on that diagram are terminal ones.

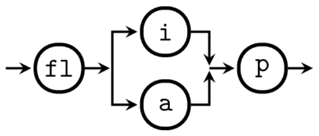


Figure C.4: Simple syntax diagram

Figure C.5 presents a syntax diagram with a non-terminal. The name of the non-terminal is Options. Assume that Figure C.4 presents the string associated with this non-terminal. Then the following strings would correspond to the diagram of Figure C.5: `"My favorite word is: flip."` and `"My favorite word is: flap."`.

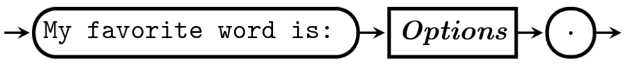


Figure C.5: Syntax diagram with non-terminal

Since syntax diagrams does not support all meta-characters presented in Table C.1, a number of extensions were introduced.

C.4.2 Extended syntax diagrams

Range of characters

Regular expressions allow to specify a range of characters. For example, if a user is required to type in a decimal digit into a field, syntax of such a field can be described as `[0-9]`. The question arises how to represent this on a syntax diagram. It was decided to represent a range of characters (e.g. `[0-9]`) as a node in inverse colours and to separate the first and last element of the range with double dot. An example presented in Figure C.6 describes a set of strings with letter `A` followed by a decimal digit (here are correct strings: `A0`, `A1`, `A2`, ...).

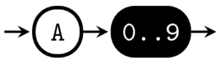


Figure C.6: A syntax diagram with a range of characters

Bound repetitions

Regular expressions support bound repetitions. Bound repetitions have the following form: `r{m,n}`. It means that string represented by `r` can be repeated `m` times, or `m+1` times, or ..., up to `n` times. For example `[a-c]{2,3}` requires a string consisting of letters `a`, `b`, `c` and there are from 2 to 3 such letters (e.g. `aa`, `ab`, `ac`, `aaa`, `aab`, ...). That regular expression could be represented by a syntax diagram in Figure C.7. This approach has the following weakness: if `n` >> `m` then such a diagram would be huge.

To solve this problem it is proposed to put the expressions `m..n` above the repeated element. Using this notation the diagram in Figure C.7 would be compressed to the one in Figure C.8.

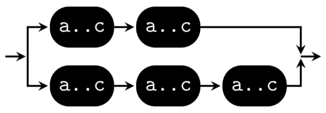


Figure C.7: Syntax diagram for a regular expression `[a-c]{2,3}`

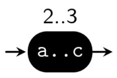


Figure C.8: New syntax diagram for a regular expression `[a-c]{2,3}`

Alternative element selection

Alternative element selection is a shorthand for diagrams representing a number of options. Let us consider a diagram with options presented in Figure C.9. Using alternative element selection the same diagram could be replaced by one of Figure C.10. The advantage of this extension is ease of specifying quantity for repetition in a way presented in Figure C.8.

Space character

Space is a non-visible character and its presence may be ambiguous for readers. Those situations often take place when the space is the last character or the only one in a terminal. For example, Figure C.11 presents the following regular expression " `|msg ` " (for better space recognitions the expression was placed in quotes).

To solve this problem the space character is replaced by a visible representation. In case of one character in a terminal the word *space* in italics is used. When the space is the last character in a string it is replaced by symbol ␣. A new version of the discussed diagram is presented in Figure C.12.

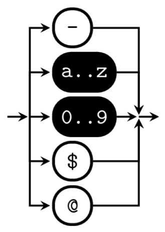


Figure C.9: Visual representation of `[-a-z0-9$@]`

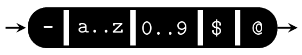


Figure C.10: Alternate visual representation of `[-a-z0-9$@]`

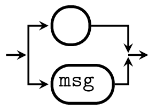


Figure C.11: Syntax diagram with the spaces character

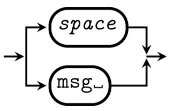


Figure C.12: New space representation

C.5 Generation of a syntax diagram

C.5.1 Overview of the generator

The purpose of the generator presented in Figure C.1 is to provide a three-part explanation. One of the parts is visual representation. The generator of the visual representation (i.e. generator of syntax diagrams) consists of the following units (see Figure C.13):

- Expression parser,
- Decorator,
- Explanation planner,
- Generator of referring expressions,
- Surface producer.

The purpose of expression parsing is to generate a parse tree of an expression being explained. The tree is next decorated with attributes that are used by the explanation planner. The planner decides about layout of the diagram to be generated. If, according to planner's decisions, a regular expression is to be explained with more than one diagram there must be a reference from one diagram to another. That reference will have a form of a name: an auxiliary diagram will be given a name and that name appear on the master diagram marking a place in which the auxiliary diagram could be inserted (it resembles superposition of regular expressions mentioned in Section C.3). The problem is how to choose a name that would be appropriate for the end-user (i.e. the reader). That is the mission of the generator of referring expressions. When layout prepared by the explanation planner is ready and – in case of a set of diagrams – the diagrams are connected with names somehow meaningful to the reader, the only task is to draw the diagrams and that is performed by the surface producer.

Each of the elements presented in Figure C.13 is described more deeply in subsequent subsections.

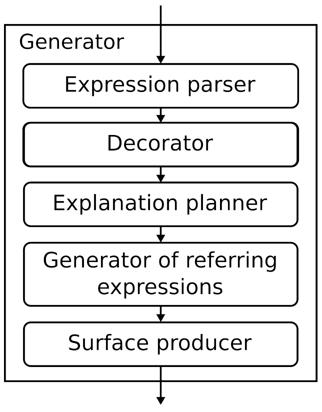


Figure C.13: Schema of the proposed field explanation generator

C.5.2 Expression parser

Input to the generator of syntax diagrams consists of three parts (e.g. Figure C.14):

- name of the described field,
- predefined names for auxiliary diagrams (optional),
- regular expression describing syntax of the field.

Second part is used to provide additional information to the generator. It may contains parts of a regular expression with names assigned to them. Those names will be used as referring names for auxiliary diagrams. The generator expects that the name of the described field and a regular expression describing its syntax are the last line of input, while the preceding lines contained predefined names.

The expression parser takes a regular expression as its input and generates a parse tree. Each node represents a small part of a regular expression and contains information about its type and attributes. For example, for character class `[A-Z]` the corresponding node would be `InCharClass` and the node would contain no attribute. This node would have one child, its node type would be `Range` and it would contain two attributes `min` and `max`, which describe the range of characters.

Figures C.15 and C.17 present examples of parse trees. The first figure contains parse tree for character class `[A-Z]` and the second one shows parse tree generated for input data from Figure C.14.

Tables C.2 and C.3 contain information about the nodes that may appear in a parse tree. Table C.2 contains parts of a regular expression with node types used to represent them. Table C.3 presents attributes which are used by the nodes (if the attribute is used, a letter 'X' is placed). Additionally, information about the type of attribute is provided in the table's header (VDM notation is used for this purpose). If the types of an attribute vary between the nodes (e.g. `value`) the column is divided into multiple parts to represent all possible types.

```
ElementType = [A-Z][0-9]{2}
ElementID = {ElementType}-[0-9]{5}-[0-9]{5}
```

Figure C.14: An example of input date to the generator

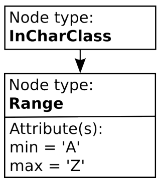


Figure C.15: Simple parse tree

The definition of the `NodeType` used in Table C.3 is presented in Figure C.16. To simplify the notation two artificial nodes were added. The first one is called `CharClass` and is used to represent nodes connected with character classes (i.e. `InCharClass` and `ExCharClass`). The second one is `Quantity` and it may be used instead of nodes designed to represent quantity nodes, like `AtLeastQuantity` and `ExactQuantity`.

The information stored in Table C.3 allow to construct a record in VDM. For example `Char` consists of attributes value and length, thus the declaration of this

```
Quantity = OptionQuantity | OptionSeqQuantity | SeqQuantity |
    ExactQuantity | AtLeastQuantity | RangeQuantity;
CharClass = InCharClass | ExCharClass;
NodeType = Concat | Char | Range | String | CharClass | ReStart |
    ReEnd | Quantity;
```

Figure C.16: VDM definition of the `NodeType`

node would be the following:

```
Char :: value : char
        length : ℕ
```

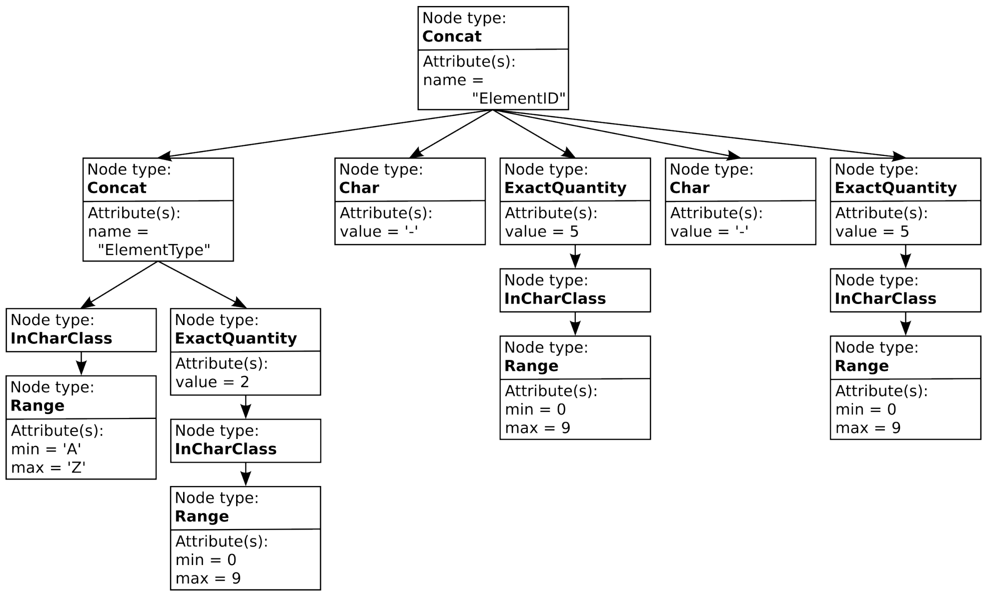


Figure C.17: Parse tree for a regular expression from Figure C.14

C.5.3 Decorator

The purpose of the decoration is to compute attributes. The attributes provide information which will help to determine the layout of diagrams.

The following attributes of nodes are computed in the decoration stage:

- `duplicated` – `true`, if the content of a node occurs more then ones,
- `length` – number of characters required to present a node in a syntax diagram (including children).

Table C.2: Node types supported be the generator

Regular expression	**Node type**
concatenation	Concat
character	Char
range	Range
string	String
[ ]	InCharClass
[^]	ExCharClass
^	ReStart
$	ReEnd
?	OptionQuantity
*	OptionSeqQuantity
+	SeqQuantity
{m}	ExactQuantity
{m,}	AtLeastQuantity
{m,n}	RangeQuantity

Table C.3: Attributes supported by the generator

NodeType	**Attributes**													
	value:			min:		max:		length:$\mathbb{N}$	duplicate:$\mathbb{B}$	autonomous:$\mathbb{B}$	internal:$\mathbb{B}$	name:char*	size:$\mathbb{N}$	children:NodeType*
	char	char*	$\mathbb{N}$	char	$\mathbb{N}$	char	$\mathbb{N}$							
Concat								X	X	X	X	X	X	X
Char	X							X						
Range				X	X	X	X	X						
String		X						X						
CharClass								X	X	X	X	X	X	X
ReStart								X						
ReEnd								X						
OptionQuantity								X	X	X	X	X	X	X
OptionSeqQuantity								X	X	X	X	X	X	X
SeqQuantity								X	X	X	X	X	X	X
ExactQuantity			X					X	X	X	X	X	X	X
AtLeastQuantity					X			X	X	X	X	X	X	X
RangeQuantity					X		X	X	X	X	X	X	X	X

Duplication detection is similar to common expression elimination presented by Aho et al.[8] which uses hash table to store information about the nodes. Inside this data structure a well know key and value are used. For the purpose of key, a regular expression represented by the nodes is used. Inside the value, a list of references to similar nodes is stored. To find duplicates, the nodes in parse tree are visited using the post-order walk. When a node that supports `duplicated` attribute is found its presence is noted in the hash table. After the walk ends, the analysis of collected information begins. If a node have duplicates (more then one reference stored in the value) all references in parent nodes are replaced to the first occurrence of the node and attribute `duplicated` is set to `true`. If there are no duplicates, the attribute `duplicated` is set to `false`.

Attribute `length` is computed bottom-up, also using post-order walk. Table C.4 presents the way of its computation, for this purpose the VDM notation was used.

Figure C.18 presents an example of decorated tree from Figure C.14. Nodes were decorated with attributes. Dotted arrows shows the references to the existing node.

Moreover, all the nodes which contain children (e.g. `InCharClass`, `Concat`) are provided with one additional attribute. This attribute is called `size` and it defines number of children. For example, the root node in Figure C.18 contains 5 children, thus attribute `size=5` (since it is not a decorated attribute it is not placed in the figure).

Table C.4: Computation of the length attribute

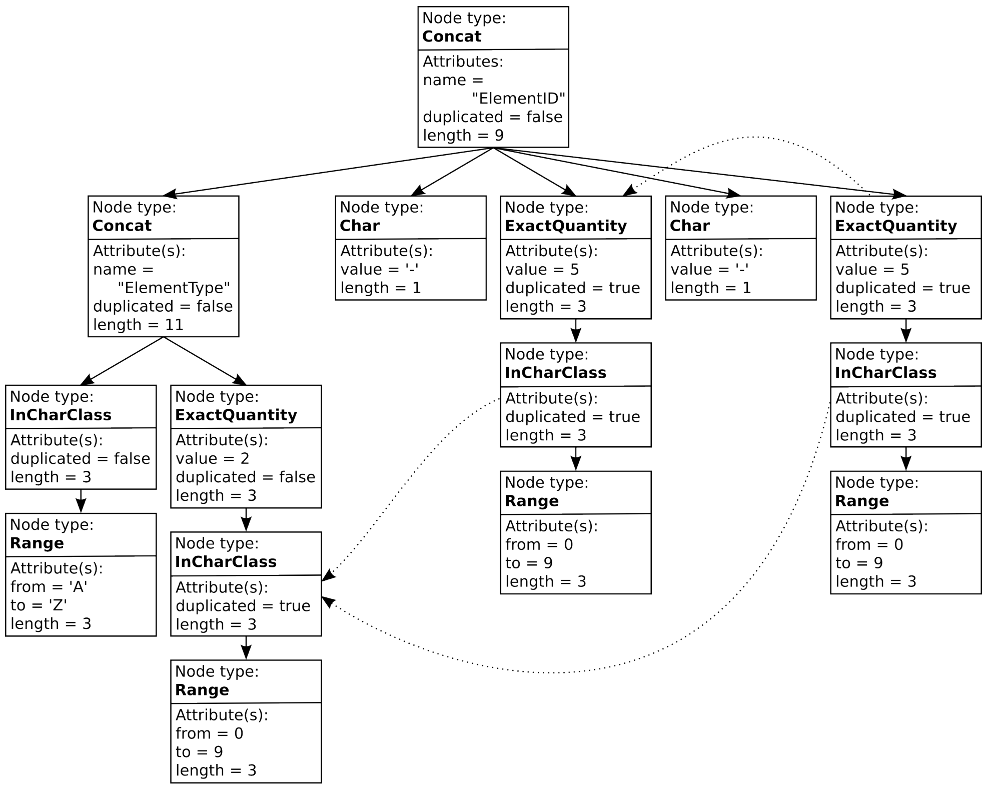
Node type	**Length**
`Concat`	`Concat.length := len Concat.name;`
	`if Concat.length = 0 then`
	`    for node in Concat.children do`
	`        Concat.length := Concat.length + node.length;`
`Char`	`Char.length := 1;`
`Range`	`Range.length := 3;`
`String`	`String.length := len String.value;`
`CharClass`	`CharClass.length := 0;`
	`for node in CharClass.children do`
	`    CharClass.length := CharClass.length + node.length;`
	`CharClass.length := CharClass.length + CharClass.size-1;`
`ReStart`	`ReStart.length := 0;`
`ReEnd`	`ReEnd.length := 0;`
`Quantity`	`Quantity.length := Quantity.children(1).length;`
	there is only one child in Quantity node

C.5.4 Explanation planner

The layout of the diagrams is determined by the following two attributes:

- `autonomous` – `true`, if the node (together with its children) could be represented as a separate diagram (internal or external, default: `false`),
- `internal` – `true`, if the separate diagram could be represented inside the parent diagram, it is used only when the autonomous attribute is set to `true` (default: `false`).

The above attributes are computed by a set of rules (similar to YACC). Each rule is build from two sections: a grammar rule and an action. The first one defines a sequence of nodes in a parse tree. When the sequence is matched the corresponding action is invoked. Figure C.19 presents a simple example of a rule. In this example the grammar rule is the following string: `Concat : CharClass` (two node types separated by a colon). The action is placed in curly brackets and it contains an action to invoke. If the ordering restrictions are not important then instead of a node type an asterisk sign can by used (see Figure C.20).



Node type: Concat
Attributes: name = "ElementID" duplicated = false length = 9
Node type: Concat
Attribute(s): name = "ElementType" duplicated = false length = 11
Node type: Char
Attribute(s): value = '-' length = 1
Node type: ExactQuantity
Attribute(s): value = 5 duplicated = true length = 3
Node type: Char
Attribute(s): value = '-' length = 1
Node type: ExactQuantity
Attribute(s): value = 5 duplicated = true length = 3
Node type: InCharClass
Attribute(s): duplicated = false length = 3
Node type: ExactQuantity
Attribute(s): value = 2 duplicated = false length = 3
Node type: InCharClass
Attribute(s): duplicated = true length = 3
Node type: InCharClass
Attribute(s): duplicated = true length = 3
Node type: Range
Attribute(s): from = 'A' to = 'Z' length = 3
Node type: InCharClass
Attribute(s): duplicated = true length = 3
Node type: Range
Attribute(s): from = 0 to = 9 length = 3
Node type: Range
Attribute(s): from = 0 to = 9 length = 3
Node type: Range
Attribute(s): from = 0 to = 9 length = 3

Figure C.18: An example of a decorated parse tree

Figure C.20 presents the default set of rules provided with the generator. They are kept in a separate file and can be modified if required.

To simplify the notation it is possible to use two artificial nodes introduced in Section C.5.2 and provide information about expected quantity. In case of the node `AtLeastQuantity{0,}` it is possible to change its type to `OptionQuantity` (see the ending of rules in Figure C.20).

```
Concat : CharClass
    { if (CharClass.size > 2) { CharClass.autonomous = true } }
```

Figure C.19: An example of a planner rule

```
Quantity    : CharClass
    { if (CharClass.size > 1) { Quantity.autonomous = true } }
Quantity    : Concat
    { if (Concat.size > 1) { Concat.autonomous = true } }
Concat      : CharClass
    { if (CharClass.duplicated == true and CharClass.size != 1)
          { CharClass.autonomous = true }
    }
*           : Concat
    { if (Concat.length > 40) { Concat.autonomous = true } }
Alterantion : Concat { Concat.autonomous = true }
Concat      : Alterantion { Concat.autonomous = true }
Concat      : CharClass
    { if (CharClass.size > 2) { CharClass.autonomous = true } }
*           : Concat
    { if (Concat.duplicated == false and Concat.size <= 5
              and Concat.autonomous == true)
          { Concat.internal = true }
    }
*           : AtLeastQuantity{0,}
    { AtLeastQuantity = OptionQuantity }
}
```

Figure C.20: Default rules in planner unit

C.5.5 Generator of referring expressions

The purpose of generator of referring expressions is to provide meaningful referential names for auxiliary diagrams. This goal is achieved by a set of encoded rules which try to guess the most suitable name.

The generator is equipped with a set of predefined names. Those names can be divided into the following categories:

- auxiliary diagrams at the beginning: e.g. Beginning, Header;
- auxiliary diagrams at the end: e.g. Ending, Footer;
- auxiliary diagrams that are used only once: e.g. Ingredient.

When generating a referring expression only a category of a referring expression is known, not a particular word. That word is draw from a category randomly.

Additionally to presented categories there are two additional mechanisms. The first one tries to generate a name by analyzing the content of a regular expression. This solution works only for character class `[a-z]` concatenated with additional characters. For example `[a-z]+\.` may by referenced by using "Word with a dot". The last mechanisms name auxiliary diagrams using pattern "Part" with a unique upper case letter, e.g. PartA, PartB, etc.

If the results are not satisfactory, then names in referring expressions may be overwritten by providing more meaningful names in an input to the generator (see Section C.5.2).

C.5.6 Surface producer

When layout of master and auxiliary diagrams is ready, and reference names are provided syntax diagrams generation may commence.

Figure C.22 presents an example of a syntax diagram which contains a web page address. This diagram was generated from an input data presented in Figure C.21. The name of a master diagram is "HTTP Identifier". Inside this diagram there are two auxiliary diagrams: "Word with a dot" and "Ending" (both names were provided by the generator of referring expressions).

```
HTTPIdentifier = http[s]?://([a-z]+)̇+[a-z]2,(:[0-9]+)?[/]?
```

Figure C.21: Input data for a HTTP Identifier

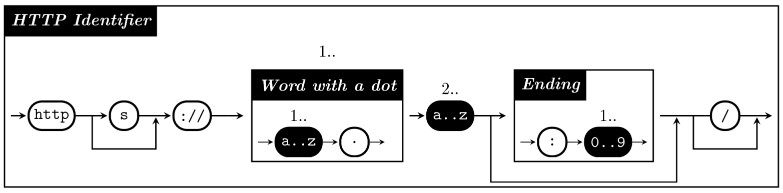


Figure C.22: A syntax diagrams of a HTTP Identifier

C.6 Early evaluation

To check the efficiency of the syntax diagrams an experiment was performed. For that purpose a survey with an explanation of diagram notation and test questions was prepared. There were prepared three diagrams with nine questions (three questions per syntax diagram).

In the experiment 56 students participated, all from a non-computer science faculty. The results show that 82.34% of answers were correct, which means that after short introduction users can understand the syntax diagrams without major problems.

C.7 Conclusions

The problem described in the article concerns automatic generation of syntax diagrams that can explain a regular expression in an ease to understand way (that the end-user could read and understand what should be typed into a field in a web application). As it was mentioned in the beginning of the article this is only a sub part of a bigger work, which purpose is to automatically generate a three-part explanation comprising verbal explanation, examples and visual representation. We have described an experiment concerning automatic generation of syntax diagrams from regular expressions and the results of early evaluation are promising.

Acknowledgement

This paper was financially supported by "Scholarship support for Ph.D. students specializing in majors strategic for Wielkopolska's development", Sub-measure 8.2.2 Human Capital Operational Programme, co-financed by European Union under the European Social Fund.

Bibliography

[1] OpenNLP. URL `http://opennlp.apache.org`. [Online; accessed 17 August 2014].

[2] Stanford Parser: A statistical parser, . URL `http://nlp.stanford.edu/software/lex-parser.shtml`. [Online; accessed 17 August 2014].

[3] Stanford Log-linear Part-Of-Speech Tagger, . URL `http://nlp.stanford.edu/software/tagger.shtml`. [Online; accessed 17 August 2014].

[4] Stanford Tokenizer, . URL `http://nlp.stanford.edu/software/tokenizer.shtml`. [Online; accessed 17 August 2014].

[5] *Common European Framework of Reference for Languages: Learning, Teaching, Assessment.* Applied Linguistics Non Series. Cambridge University Press, 2001. ISBN 9780521005319.

[6] John Aberdeen, John Burger, David Day, Lynette Hirschman, Patricia Robinson, and Marc Vilain. MITRE: description of the Alembic system used for MUC-6. In *Proceedings of the 6th conference on Message understanding*, pages 141–155. Association for Computational Linguistics, 1995.

[7] Steve Adolph and Paul Bramble. *Patterns for Effective Use Cases.* Addison Wesley, Boston, 2002. ISBN 978-0201721843.

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2006. ISBN 0321486811.

[9] Bartosz Alchimowicz and Jerzy Nawrocki. Generating syntax diagrams from regular expressions. *Foundations of Computing and Decision Sciences*, 36(2):81–97, 2011.

[10] Bartosz Alchimowicz and Jerzy Nawrocki. The COCA quality model for user documentation. *Software Quality Journal (not assigned to an issue yet)*, 2014.

[11] Bartosz Alchimowicz, Jakub Jurkiewicz, Mirosław Ochodek, and Jerzy Nawrocki. Building Benchmarks for Use Cases. *Computing and Informatics*, 29(1):27–44, 2010.

[12] Bartosz Alchimowicz, Jakub Jurkiewicz, Jerzy Nawrocki, and Mirosław Ochodek. Towards use-cases benchmark. In *Software Engineering Techniques*, pages 20–33. Springer Berlin Heidelberg, 2011.

[13] Bartosz Alchimowicz, Jerzy Nawrocki, and Mirosław Ochodek. Towards automatic explanation of field syntax in web applications. Technical Report RA-10/2014, Politechnika Poznańska, 2014.

[14] Carl Martin Allwood and Tomas Kalén. Evaluating and improving the usability of a user manual. *Behaviour & Information Technology*, 16(1):43–57, 1997.

[15] Srinivas Bangalore, Owen Rambow, and Steve Whittaker. Evaluation metrics for generation. In *Proceedings of the first international conference on Natural language generation*, volume 14, pages 1–8. Association for Computational Linguistics, 2000.

[16] Victor R. Basili, Gianluigi Caldiera, and Dieter H. Rombach. *The Goal Question Metric Approach*, volume I. John Wiley & Sons, 1994.

[17] Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. RegViz: visual debugging of regular expressions. In *ICSE Companion*, pages 504–507, 2014.

[18] Anja Belz and Ehud Reiter. Comparing Automatic and Human Evaluation of NLG Systems. In *In Proc. EACL'06*, pages 313–320, 2006.

[19] Daniel M Berry, Khuzaima Daudjee, Jing Dong, Igor Fainchtein, Maria Augusta Nelson, Torsten Nelson, and Lihua Ou. User's manual as a requirements specification: case studies. *Requirements Engineering*, 9(1):67–82, 2004.

[20] Alan F. Blackwell. Your Wish is My Command. In *Your Wish is My Command*, chapter SWYN: A Visual Representation for Regular Expressions, pages 245–270. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-688-2.

[21] Alan F Blackwell. See what you need: Helping end-users to build abstractions. *Journal of Visual Languages & Computing*, 12(5):475–499, 2001.

[22] IEEE-SA Standards Board. *IEEE Std 1063-2001, IEEE standard for Software User Documentation*. Institute of Electrical and Electronics Engineers, 2001.

[23] Taylor L Booth. Probabilistic representation of formal languages. In *Switching and Automata Theory, 1969., IEEE Conference Record of 10th Annual Symposium on*, pages 74–81. IEEE, 1969.

[24] Thorsten Brants. TnT: a statistical part-of-speech tagger. In *Proceedings of the sixth conference on Applied natural language processing*, pages 224–231. Association for Computational Linguistics, 2000.

[25] Eric Brill. A simple rule-based part of speech tagger. In *Proceedings of the workshop on Speech and Natural Language*, pages 112–116. Association for Computational Linguistics, 1992.

[26] Ivan Budiselic, Sinisa Srbljic, and Miroslav Popovic. RegExpert: A Tool for Visualization of Regular Expressions. In *EUROCON, 2007. The International Conference on "Computer as a Tool"*, pages 2387–2389. IEEE, 2007.

[27] J. Byrne. *Scientific and Technical Translation Explained: A Nuts and Bolts Guide for Beginners*. Translation Practices Explained. Taylor & Francis, 2014. ISBN 9781317642046.

[28] José Coch. Evaluating and comparing three text-production techniques. In *Proceedings of the 16th conference on Computational linguistics*, volume 1, pages 249–254. Association for Computational Linguistics, 1996.

[29] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000. ISBN 0201702258.

[30] Jacob Cohen. Statistical power analysis. *Current Directions in Psychological Science*, 1(3):98–101, 1992.

[31] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[32] W. J. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, 3rd edition, 1999.

[33] 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. ISBN 0-201-92478-1.

[34] Microsoft Corporation. *Microsoft Manual of Style*. Microsoft Press Series. Microsoft Press, 4th edition, 2012. ISBN 9780735648715.

[35] Croton Research. Graphrex, 2014. URL `http://crotonresearch.com/graphrex/`. [Online; accessed 10 February 2014].

[36] Robert Dale, Hermann Moisl, and Harold Somers. *Handbook of natural language processing*. CRC Press, 2000.

[37] F. DeRespinis, J. Jenkins, International Business Machines Corporation, A. Laird, P. Hayward, and L.I. McDonald. *The IBM Style Guide: Conventions for Writers and Editors*. IBM Press Series. IBM Press/Pearson, 2011. ISBN 9780132101301.

[38] Steven J DeRose. Grammatical category disambiguation by statistical optimization. *Computational Linguistics*, 14(1):31–39, 1988.

[39] Magdalena Derwojedowa, Maciej Piasecki, Stanisław Szpakowicz, and Magdalena Zawisławska. Polish WordNet on a shoestring. In *Proceedings of Biannual Conference of the Society for Computational Linguistics and Language Technology, Tübingen*, pages 169–178, 2007.

[40] George Doddington. Automatic evaluation of machine translation quality using n-gram co-occurrence statistics. In *Proceedings of the second international conference on Human Language Technology Research*, pages 138–145. Morgan Kaufmann Publishers Inc., 2002.

[41] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[42] Martin Erwig and Rahul Gopinath. Explanations for Regular Expressions. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 394–408, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28871-5.

[43] José Figueira, Salvatore Greco, and Matthias Ehrgott. *Multiple criteria decision analysis: state of the art surveys*, volume 78. Springer, 2005.

[44] Julie Fisher. User Satisfaction and System Success: considering the development team. *Australasian Journal of Information Systems*, 9(1):21–29, 2001. ISSN 1449-8618.

[45] Karl Fogel. *Producing open source software: How to run a successful free software project*. O'Reilly Media, Inc., 2005.

[46] W. Nelson Francis and Henry Kučera. *Frequency analysis of English usage: lexicon and grammar*. Houghton Mifflin, 1982.

[47] GW French, JR Kennaway, and AM Day. Programs as visual, interactive documents. *Software: Practice and Experience*, 44:911–930, 2014.

[48] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2002. ISBN 0596002890.

[49] Gregory Grefenstette and Pasi Tapanainen. *What is a word, what is a sentence?: problems of Tokenisation*. Rank Xerox Research Centre, 1994.

[50] Robert Gunning. *The technique of clear writing*. New York: McGraw-Hill International, 1952.

[51] David Hardcastle and Donia Scott. Can we evaluate the quality of generated text? In Bente Maegaard Joseph Mariani Jan Odijk Stelios Piperidis Daniel Tapias Nicoletta Calzolari (Conference Chair), Khalid Choukri, editor, *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco, may 2008. European Language Resources Association (ELRA). ISBN 2-9517408-4-0.

[52] Anthony Hartley, Donia Scott, John Bateman, and Danail Dochev. AGILE–A system for multilingual generation of technical instructions. In *MT Summit VIII, Machine Translation in the Information Age, Proceedings*, pages 145–150. Santiago de Compostela, Spain, 2001.

[53] Alan Hartman. Software and hardware testing using combinatorial covering suites. In *Graph Theory, Combinatorics and Algorithms*, pages 237–266. Springer, 2005.

[54] Jonathan Hayward. *Django JavaScript Integration: AJAX and jQuery*. Packt Publishing, 2011. ISBN 1849510342, 9781849510349.

[55] Richard Hazlett. Measurement of user frustration: a biologic approach. In *CHI '03 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '03, pages 734–735, New York, NY, USA, 2003. ACM. ISBN 1-58113-637-4. doi: 10.1145/765891.765958.

[56] Marti A. Hearst. Multi-paragraph segmentation of expository text. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 9–16, Las Cruces, New Mexico, USA, June 1994. Association for Computational Linguistics.

[57] Adrian Holovaty and Jacob Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right*. Expert's voice in Web development. Apress, 2nd edition, 2009. ISBN 978-1430219361.

[58] Dag Hovland. The inclusion problem for regular expressions. In *Language and Automata Theory and Applications*, pages 309–320. Springer, 2010.

[59] Shihong Huang and Scott Tilley. Towards a documentation maturity model. In *Proceedings of the 21st Annual International Conference on Documentation*, pages 93–99. ACM, 2003.

[60] Russ Hurlbut. A survey of approaches for describing and formalizing use cases. *Expertech, Ltd*, 1997.

[61] IEEE. *IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications*. Institute of Electrical and Electronics Engineers, 1998.

[62] IEEE. *IEEE Std 1028-2008, IEEE Standard for Software Reviews and Audits*. Institute of Electrical and Electronics Engineers, 2008.

[63] ISO/IEC. *ISO/IEC 14977:1996 - Information technology – Syntactic metalanguage – Extended BNF*. International Organization for Standardization, Geneva, Switzerland, 1996.

[64] ISO/IEC. *ISO/IEC 9126-1:2001 - Software engineering – Product quality – Part 1: Quality model*. International Organization for Standardization, Geneva, Switzerland, 2001.

[65] ISO/IEC. *ISO/IEC 25000:2005 - Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. International Organization for Standardization, Geneva, Switzerland, 2005.

[66] ISO/IEC. *ISO/IEC 12207:2008 - Systems and software engineering – Software life cycle processes*. International Organization for Standardization, Geneva, Switzerland, 2008.

[67] ISO/IEC. *ISO/IEC 26514:2008 - Systems and software engineering – Requirements for designers and developers of user documentation*. International Organization for Standardization, Geneva, Switzerland, 2008.

[68] ISO/IEC. *ISO/IEC 26513:2009 - Systems and software engineering – Requirements for testers and reviewers of user documentation*. International Organization for Standardization, Geneva, Switzerland, 2009.

[69] ISO/IEC. *ISO/IEC 25010:2011 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. International Organization for Standardization, Geneva, Switzerland, 2011.

[70] ISO/IEC/IEEE. *ISO/IEC/IEEE 24765:2010 - Systems and software engineering – Vocabulary.* International Organization for Standardization, Geneva, Switzerland, 2010.

[71] ISO/IEC/IEEE. *ISO/IEC 26512:2011 - Systems and software engineering – Requirements for acquirers and suppliers of user documentation.* International Organization for Standardization, Geneva, Switzerland, 2011.

[72] ISO/IEC/IEEE. *ISO/IEC/IEEE 29148:2011 - Systems and software engineering – Life cycle processes – Requirements engineering.* International Organization for Standardization, Geneva, Switzerland, 2011.

[73] ISO/IEC/IEEE. *ISO/IEC/IEEE 42010:2011 Systems and software engineering – Architecture description.* International Organization for Standardization, Geneva, Switzerland, 2011.

[74] ISO/IEC/IEEE. *ISO/IEC 26511:2012 - Systems and software engineering – Requirements for managers of user documentation.* International Organization for Standardization, Geneva, Switzerland, 2012.

[75] ISO/IEC/IEEE. *ISO/IEC 26515:2012 - Systems and software engineering – Developing user documentation in an agile environment.* International Organization for Standardization, Geneva, Switzerland, 2012.

[76] Ivar Jacobson. *Concepts for Modeling Large Real Time Systems.* Royal Institute of Technology, Department of Telecommunication Systems-Computer Systems, 1985.

[77] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004. ISBN 0201403471.

[78] Stephen C Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[79] T. Capers Jones. *Estimating Software Costs: Bringing Realism to Estimating.* McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 2007. ISBN 9780071483001.

[80] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring software product quality: A survey of ISO/IEC 9126. *Software, IEEE*, 21(5):88–92, 2004.

[81] Daniel Jurafsky and James H. Martin. *Speech and Language Processing.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 2009. ISBN 0131873210.

[82] Bryan Jurish and Kay-Michael Würzner. Word and Sentence Tokenization with Hidden Markov Models. *Journal of Language Technology and Computational Linguistics*, 28(2):61–83, 2013.

[83] Jakub Jurkiewicz. *Identification of Events in Use Cases.* PhD thesis, Poznań University of Technology, Poznań, Poland, 2013.

[84] Haruhiko Kaiya, Tomonori Sato, Akira Osada, Naoyuki Kitazawa, and Kenji Kaijiri. Toward quality requirements analysis based on domain specific quality spectrum. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 596–601, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-753-7.

[85] Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, DTIC Document, 1965.

[86] Ninus Khamis, René Witte, and Juergen Rilling. Automatic quality assessment of source code comments: the javadocminer. In *Natural Language Processing and Information Systems*, pages 68–79. Springer, 2010.

[87] Tibor Kiss and Jan Strunk. Unsupervised multilingual sentence boundary detection. *Computational Linguistics*, 32(4):485–525, 2006.

[88] S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[89] S. Kopczynska and J. Nawrocki. Using non-functional requirements templates for elicitation: A case study. In *Requirements Patterns (RePa), 2014 IEEE 4th International Workshop on*, pages 47–54, Aug 2014.

[90] Emiel Krahmer and Kees Van Deemter. Computational generation of referring expressions: A survey. *Computational Linguistics*, 38(1):173–218, 2012.

[91] W.H. Kruskal and W.A. Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952.

[92] Irene Langkilde-Geary. An empirical verification of coverage and correctness for a general-purpose sentence generator. In *Proceedings of the 12th International Natural Language Generation Workshop*, pages 17–24, 2002.

[93] M. E. Lesk and E. Schmidt. UNIX Vol. II. In A. G. Hume and M. D. McIlroy, editors, *UNIX Vol. II*, chapter Lex – a Lexical Analyzer Generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990. ISBN 0-03-047529-5.

[94] Lamport Leslie. *LATEX: A Document Preparation System.* Addison-Wesley Publishing Company, Inc., 2nd edition, 1994.

[95] James C Lester and Bruce W Porter. Developing and empirically evaluating robust explanation generators: The KNIGHT experiments. *Computational Linguistics*, 23(1):65–101, 1997.

[96] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014.

[97] Tomasz Marciniak and Michael Strube. Classification-based generation using TAG. In *Natural Language Generation*, pages 100–109. Springer, 2004.

[98] Mike Markel. *Technical Communication.* Bedford/St. Martin's, 2012. ISBN 9780312679484.

[99] Jiří Maršík and Ondřej Bojar. TrTok: A Fast and Trainable Tokenizer for Natural Languages. *The Prague Bulletin of Mathematical Linguistics*, 98:75–85, 2012.

[100] Steve McConnell. *Software Estimation: Demystifying the Black Art.* Microsoft Press, Redmond, WA, USA, 2006. ISBN 9780735605350.

[101] Kathleen McKeown, Karen Kukich, and James Shaw. Practical issues in automatic documentation generation. In *Proceedings of the fourth conference on Applied natural language processing*, pages 7–14. Association for Computational Linguistics, 1994.

[102] G Harry McLaughlin. SMOG grading: A new readability formula. *Journal of Reading*, 12(8): 639–646, 1969.

[103] Susan W. McRoy, Songsak Channarukul, and Syed S. Ali. Yag: A template-based generator for real-time systems. In *Proceedings of the first international conference on Natural language generation*, volume 14, pages 264–267. Association for Computational Linguistics, 2000.

[104] Susan W. Mcroy, Songsak Channarukul, and Syed S. Ali. An augmented template-based approach to text realization. *Natural Language Engineering*, 9(4):381–420, December 2003. ISSN 1351-3249.

[105] Chris Mellish and Robert Dale. Evaluation in the context of natural language generation. *Computer Speech & Language*, 12(4):349–373, 1998.

[106] George A Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11): 39–41, 1995.

[107] Ruslan Mitkov and Le An Ha. Computer-aided generation of multiple-choice tests. In *Proceedings of the HLT-NAACL 03 workshop on Building educational applications using natural language processing*, volume 2, pages 17–22. Association for Computational Linguistics, 2003.

[108] Jerzy R. Nawrocki and Łukasz Olek. Use-Cases Engineering with UC Workbench. In Krzysztof Zielinski and Tomasz Szmuc, editors, *Software Engineering: Evolution and Emerging Technologies*, volume 130 of *Frontiers in Artificial Intelligence and Applications*, pages 319–329. IOS Press, 2005. ISBN 978-1-58603-559-4.

[109] Hermann Ney. Dynamic programming parsing for context-free grammars in continuous speech recognition. *Signal Processing, IEEE Transactions on*, 39(2):336–340, 1991.

[110] David G. Novick and Karen Ward. What users say they want in documentation. In Shihong Huang, Rob Pierce, and John W. Stamey Jr., editors, *SIGDOC*, pages 84–91. ACM, 2006. ISBN 1-59593-523-1.

[111] David G. Novick and Karen Ward. Why don't people read the manual? In Shihong Huang, Rob Pierce, and John W. Stamey Jr., editors, *SIGDOC*, pages 11–18. ACM, 2006. ISBN 1-59593-523-1.

[112] Mirosław Ochodek and Jerzy Nawrocki. Automatic Transactions Identification in Use Cases. In *Balancing Agility and Formalism in Software Engineering: 2nd IFIP Central and East European Conference on Software Engineering Techniques CEE-SET 2007*, volume 5082 of *LNCS*, pages 55–68. Springer Verlag, 2008.

[113] Mirosław Ochodek, Bartosz Alchimowicz, Jakub Jurkiewicz, and Jerzy Nawrocki. Improving the reliability of transaction identification in use cases. *Information and Software Technology*, 53(8): 885–897, 2011.

[114] Łukasz Olek, Bartosz Alchimowicz, and Jerzy Nawrocki. Acceptance testing of web applications with test description language. *Computer Science*, 15(4):459–477, 2014.

[115] Łukasz Olek, Jerzy Nawrocki, and Miroslaw Ochodek. Enhancing Use Cases with Screen Designs. In Zbigniew Huzar, Radek Kocí, Bertrand Meyer, Bartosz Walter, and Jaroslav Zendulka, editors, *CEE-SET*, volume 4980 of *Lecture Notes in Computer Science*, pages 48–61. Springer, 2008. ISBN 978-3-642-22385-3.

[116] Maryoly Ortega, María Pérez, and Teresita Rojas. Construction of a systemic quality model for evaluating a software product. *Software Quality Journal*, 11(3):219–242, 2003.

[117] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[118] Cécile Paris and Keith Vander Linden. DRAFTER: An interactive support tool for writing multilingual instructions. *IEEE Computer*, 29(7):49–56, 1996.

[119] Cecile Paris, Nathalie Colineau, Shijian Lu, and Keith Vander Linden. Automatically generating effective online help. *International Journal on E-learning*, 4(1):83–103, 2005.

[120] Arancha Pedraz-Delhaes, Muhammad Aljukhadar, and Sylvain Sénécal. The effects of document language quality on consumer perceptions and intentions. *Canadian Journal of Administrative Sciences/Revue Canadienne des Sciences de l'Administration*, 27(4):363–375, 2010.

[121] Jacob Perkins. *Python Text Processing with NLTK 2.0 Cookbook*. Packt Publishing, 2010. ISBN 1849513600, 9781849513609.

[122] Plagiat.pl. *Instrukcja Użytkownika Internetowego Systemu Antyplagiatowego Plagiat.pl*. Plagiat.pl Sp. z o.o., 2012. URL `https://www.plagiat.pl/cms_pdf/Plagiat_pl_instrukcja_uzytkownika_indywidualnego.pdf`. [Online; accessed 2 July 2014].

[123] Aarne Ranta. A Multilingual Natural-language Interface to Regular Expressions. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, FSMNLP '09, pages 79–90, Stroudsburg, PA, USA, 1998. Association for Computational Linguistics. ISBN 975-7679-34-8.

[124] Janice Ginny Redish. Adding value as a professional technical communicator (extract). *Technical communication*, 42(1):26–39, 1995.

[125] Ehud Reiter. NLG vs. templates. In *Proceedings of the Fifth European Workshop on Natural Language Generation*, pages 95–105, Leiden, the Netherlands, 1995.

[126] Ehud Reiter and Robert Dale. *Building Natural Language Generation Systems*. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-62036-8.

[127] Ehud Reiter and Somayajulu Sripada. Should corpora texts be gold standards for nlg. In *Proceedings of 2nd International Conference on Natural Language Generation*, volume 2, pages 97–104, 2002.

[128] Ehud Reiter, Chris Mellish, and Jon Levine. Automatic Generation of Technical Documentation. *Journal of Applied Artificial Intelligence*, 9(3):259–287, 1995.

[129] Marc Rettig. Nobody Reads Documentation. *Communications of the ACM*, 34(7):19–24, July 1991.

[130] Stefan Riezler and John T Maxwell. On some pitfalls in automatic evaluation and significance testing for MT. In *Proceedings of the ACL workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 57–64, 2005.

[131] Brian Roark. Probabilistic top-down parsing and language modeling. *Computational Linguistics*, 27(2):249–276, 2001.

[132] Armin Ronacher. Jinja2 Documentation, Version 2.8, 2014. URL `http://jinja.pocoo.org/`. [Online; accessed 20 August 2014].

[133] Christopher Scaffidi, Brad Myers, and Mary Shaw. Topes: reusable abstractions for validating data. In *Proceedings of the 30th international conference on Software engineering*, pages 1–10. ACM, 2008.

[134] Lenhart Schubert. Computational Linguistics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring, 2014.

[135] Ken Schwaber. *Agile project management with Scrum*. Microsoft Press, 2004.

[136] S.S. Shapiro and M.B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[137] Cathy J. Spencer. A Good User's Guide Means Fewer Support Calls and Lower Support Costs. *Technical Communication*, 42(1):52–55(4), February 1995.

[138] Somayajulu Sripada, Ehud Reiter, and Lezan Hawizy. Evaluation of an NLG System using Post-Edit Data: Lessons learnt. In *Proceedings of European Natural Language Generation Workshop*, pages 133–139, 2005.

[139] Somayajulu G Sripada, Ehud Reiter, Jim Hunter, and Jin Yu. Exploiting a parallel text-data corpus. *ENE*, 25:12, 2003.

[140] Stack Overflow. Find complement of regular expression. URL `http://stackoverflow.com/questions/15452353/find-complement-of-regular-expression`. [Online; accessed 15 March 2015].

[141] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *2013 IEEE 21st International Conference on Program Comprehension (ICPC)*, pages 83–92. IEEE, 2013.

[142] Andreas Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201, 1995.

[143] William Strunk. *The Elements of Style*. Filiquarian Publishing, LLC, 2007. ISBN 9781599869339.

[144] Gene Sullivan. Yape Regex Explain 4.01, 2010. URL `http://search.cpan.org/~gsullivan/`. [Online; accessed 7 February 2014].

[145] Sun Technical Publications. *Read Me First!: A Style Guide for the Computer Industry*. Prentice Hall, 3rd edition, 2010. ISBN 9780137058266.

[146] O.E. Swan. *A Grammar of Contemporary Polish*. Slavica, 2002. ISBN 9780893572969.

[147] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition*. IEEE, New York, NY, USA, 2004.

[148] TSO. *Managing successful projects with PRINCE2*. HM Government – Best management practice. Stationery Office, 2009. ISBN 9780113310593.

[149] University of Chicago Press. *The Chicago Manual of Style*. Chicago Manual of Style. University of Chicago Press, 2010. ISBN 9780226104201.

[150] Kees Van Deemter, Emiel Krahmer, and Mariët Theune. Real Versus Template-Based Natural Language Generation: A False Opposition? *Computational Linguistics*, 31(1):15–24, March 2005. ISSN 0891-2017.

[151] Rini van Solingen and Egon Berghout. *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*. McGraw-Hill, 1999.

[152] Pierre Wellner, Mike Flynn, Simon Tucker, and Steve Whittaker. A Meeting Browser Evaluation Test. In *CHI '92: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 0 2005. ACM Press. ISBN 1-59593-002-7.

[153] Sandra Williams and Ehud Reiter. Generating readable texts for readers with low basic skills. In *Proceedings of the 10th European Workshop on Natural Language Generation (ENLG'05*, pages 15–23, 2005.

[154] Niklaus Wirth. *The Programming Language Pascal: Revised Report*. Berichte // Zuerich ETH Dept Informatik. Eidgenössische Technische Hochschule, 1973.

[155] R Michael Young. Using grice's maxim of quantity to select the content of plan descriptions. *Artificial Intelligence*, 115(2):215–256, 1999.

[156] Stelios H Zanakis, Anthony Solomon, Nicole Wishart, and Sandipa Dublish. Multi-attribute decision making: A simulation comparison of select methods. *European Journal of Operational Research*, 107(3):507–529, 1998.

Institute of Computing Science
Poznań University of Technology, Poznań, Poland

Typeset using LaTeX in Adobe Utopia.

BibTeX entry:

```
@phdthesis{balchimowicz2015phd,
        author = "Bartosz Alchimowicz",
        title = "Automatic generation of user manual for web applications",
        school = "Pozna{\'n} University of Technology",
        address = "Pozna{\'n}, Poland",
        year = "2015",
}
```