

UTRECHT UNIVERSITY

MASTER THESIS

Ontological Traceability for Software

Author:

Sean MARTENS

Supervisors:

prof. dr. Sjaak BRINKKEMPER

dr. Fabiano DALPIAZ

*A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Business Informatics*

in

Faculty of Science
Department of Organization and Information

September 14, 2018

UTRECHT UNIVERSITY

Abstract

Graduate School of Natural Sciences
Faculty of Science

Master of Science in Business Informatics

Ontological Traceability for Software

by Sean MARTENS

In the development process of a software product, the stakeholders and developers are often buried in a plethora of software artifacts. Examples of these software artifacts include: Requirements, architecture, code, test cases, documentation, and many more. The relations between these artifacts can provide important information for the software project, but unfortunately, these software artifacts are disconnected by nature.

The research field on software traceability aims to automate the creation of trace links between software artifacts, as the manual creation of these trace links is considered to be too much work by practitioners. The research community of software traceability has set the ambitious goal of ubiquitous traceability, meaning that traceability is always there, without having to think about getting it there.

This research project aims to provide a first step towards a ubiquitous solution to software traceability, by combining it with techniques from the field of ontology, and the field of natural language processing. The theory for Ontological Traceability for Software is established, after which a proof-of-concept for the theory is designed, implemented, and evaluated. We conclude that the method proposed in the theory has the potential to ultimately provide ubiquitous traceability, but many more research and technological advances in the fields of natural language processing and ontology are necessary to achieve this goal.

Acknowledgements

I can't believe that it is only nine months ago, that I walked into the room of Sjaak Brinkkemper at the University, asking for a meeting to discuss ideas for a master thesis. After eight months of hard work, I can safely say that walking into that room, and asking Sjaak to be my guiding teacher during the process of writing my master thesis, has been one of the best decisions I've made this year. I would like to thank Sjaak for his expert feedback on the many theories and documents I proposed, and for this enthusiasm concerning my research project. Thanks are also due to Fabiano Dalpiaz, for being my second guiding teacher, and being a great help in evaluating the theories that Sjaak and I came up with.

I'm also grateful to Sjaak for assembling the Grimm project group, where we would meet with fellow graduating master students, to discuss our theories and progress. This has been a fun and interesting experience, and special thanks go out to Remmelt Blessinga and Laurens Müter, for being there the entire process, and always providing great feedback.

Furthermore, I would like to thank my parents, for guiding me to this point for 25 years. Your continuous support has helped me get to the point where I am right now, and I couldn't be more grateful to you, as well as to the rest of my family.

Finally, I would like to express my gratitude to my loving girlfriend Mijntje, whom I met a couple of months before I started this adventure at the university, back in 2011. Your support has helped me through the toughest moments of the process, and I can't thank you enough.

Contents

Abstract	iii
Acknowledgements	v
1 Introduction	1
1.1 Problem Statement	1
1.2 Research Objective	2
1.3 Research Structure	3
2 Research Approach	5
2.1 Research Problem	5
2.2 Research Questions	5
2.3 Research Method	7
2.3.1 Experiment Design	7
2.4 Literature Research Protocol	8
3 Theoretical Foundation	9
3.0.1 The History of Software Traceability Research	12
3.1 Software Traceability in Practice	13
3.2 Automated Software Traceability	15
3.3 Ontological Traceability for Software	16
3.4 User Stories	17
3.5 Ontology Learning from Text	17
3.6 Ontology Visualization	19
3.7 Ontology Storage	22
3.8 Ontology Mapping	23
3.9 Literature Review Conclusions	23
4 Linguistic Tooling for Software Development: A Vision	25
4.1 Problem Statement	25
4.2 Vision	26
4.3 Needs	27
5 Ontological Traceability for Software Theory	31
5.1 Product Domain Ontology	31
5.1.1 Software Artifacts: Building Blocks for the PDO	32
5.1.2 Formalizing Software Artifacts	34

5.1.3	Creating the PDO	35
5.2	PDO Traceability Method	37
5.2.1	Trace Link Creation Theory	38
5.2.2	PDO Traceability Method Overview	39
5.2.3	Artifact Instance Selection	39
5.2.4	Linguistic Term Extraction	42
5.2.5	Sub-ontology Creation	43
	Statis vs. Run-time Sub-ontology Creation	43
5.2.6	Ontology Matching	44
5.2.7	Trace Link Creation	44
5.2.8	Implementation	46
6	Trace Link Integration in GitLab	47
6.1	Ontology Based Artifact Navigation	47
6.2	PDO Based Artifact Suggestions	49
6.3	User Story Testing	50
6.4	Requirement Impact Analysis	50
7	Proof-of-concept Implementation	53
7.1	Technology	53
7.2	Architecture	54
7.2.1	Functional Architecture	54
7.3	Linguistic Term Extraction	58
7.3.1	User Story	59
7.3.2	Code	59
	Model	60
	View	60
	Controller	60
7.3.3	Test Case	61
7.3.4	Linguistic Term Splitting	61
7.3.5	Linguistic Term Extraction Pseudocode	61
7.4	Sub-ontology Creation	62
7.4.1	Sub-ontology Creation Procedure	63
7.5	Trace Link Generation	64
7.6	Connecting the Modules	65
7.7	UTA Screenshots	66
8	Validation: Experimental Results	69
8.1	UTA Evaluation: Experiment Overview	69
8.1.1	Experiment: Second Iteration	70
8.1.2	Data Set: The UAS Software Product	70
8.1.3	PDO for the UAS	71
8.2	UTA Evaluation: Trace Links	71

8.2.1	Trace Links Identified by Domain Experts	71
8.2.2	Trace Links Identified by UTA	73
8.2.3	Trace Link Comparison: User Stories to Feature Tests	74
8.2.4	Trace Link Comparison: User Stories to Models	75
8.2.5	Trace Link Comparison: User Stories to Controllers	76
8.2.6	UTA Performance: Combined Results	77
8.3	UTA Performance Discussion	77
8.3.1	Missing Domain Knowledge	78
8.3.2	Over-optimistic Sub-ontology Generation	78
8.3.3	General Remarks	79
9	Discussion	81
9.1	Sub Research Questions	82
9.1.1	Discussion: Software Traceability in Literature	83
9.1.2	Discussion: Ontological Traceability for Software Theory	84
9.1.3	Discussion: GitLab Integration	85
9.1.4	Discussion: Proof-of-concept Performance	86
9.2	Main Research Question	87
9.2.1	Establishing Trace Links Using an Ontology	89
9.3	Conclusions	90
9.3.1	Ontological Traceability for Software	90
9.3.2	Software Traceability in Practice	92
9.4	Validity Threats	93
9.4.1	External Validity	93
9.4.2	Internal Validity	93
9.4.3	Reliability	93
9.4.4	Limitations	94
9.5	Future Research	94
9.5.1	Extensive Evaluation of the UTA	94
9.5.2	Improvement of Ontology Learning in PDO Creation	94
9.5.3	Integration of the UTA in GitLab	95
9.5.4	Evaluate Static and Run-time Sub-ontology Creation	95
	Bibliography	97
A	UAS User Stories	101
B	UAS Test Cases	103
C	Scientific Paper	105
D	Trace Links Created by Practitioners	107
D.1	User Stories to Feature Tests	107
D.2	User Stories to Models	108

D.3	User Stories to Controllers	109
E	Trace Links Created by UTA	111
E.1	User Stories to Feature Tests	111
E.2	User Stories to Models	112
E.3	User Stories to Controllers	113

List of Abbreviations

PDO	Product Domain Oontology
UTA	Ubiquitous Traceability Artisan

Chapter 1

Introduction

1.1 Problem Statement

The development of a software product often yields a lot of artifacts, such as requirements, architecture, source code, and documentation. The relations between these software artifacts contain useful information that can enable various activities in the development process, such as change impact analysis, bug searching, and requirements coverage analysis. (Bouillon, Mäder, and Philippow, 2013)

Unfortunately, software artifacts are often disconnected, which complicates the creation of these relations between software artifacts. (Cleland-Huang, Gotel, and Zisman, 2012) Software traceability is the common naming for the set of practices that attempt to establish navigable trace links between different software artifacts that would otherwise be disconnected, and it is defined as follows:

"Software traceability is the ability to interrelate any uniquely identifiable software engineering artifact to any other, maintain required links over time, and use the resulting network to answer questions of both the software product and its development process." (Cleland-Huang et al., 2014)

In 1994, Gotel and Finkelstein already found that, despite its potential, software traceability was not used as widespread in practice at the time. (Gotel and Finkelstein, 1994) Over a decade later, Blaauwboer et al. also concluded that the use of software traceability in software development is not very common in practice. (Blaauwboer, Sikkel, and Aydin, 2007) A possible explanation for this phenomenon is that the cost, effort and discipline needed to create and maintain trace links often outweighs their benefits. This is mostly caused by the fact that these benefits are not always fully realized due to lacking processes or the absence of effective tooling. (Arkley and Riddle, 2005)

In 2005, the Center of Excellence for Software Traceability (CoEST) was formed by a group of researchers in the United States. Their goal was to spark the research community to work together to tackle open challenges in the field of software traceability. (Antoniol et al., 2017) They formulated an ultimate goal for the research community called ubiquitous software traceability, which is defined as follows:

"Ubiquitous Software Traceability is software traceability that is always there, without ever having to think about getting it there, as it is built into the engineering process; traceability has effectively; 'disappeared without a trace.' Achieved only when traceability is established and sustained with near zero effort." (Cleland-Huang et al., 2014)

The assumption of this research is that ubiquitous software traceability can be achieved by using natural language processing (NLP). Using NLP, domain knowledge of the software product under analysis can be extracted from available software artifacts. This knowledge can then be used to construct a domain ontology for the software product. This research investigates whether it is possible to use such a domain ontology to establish bidirectional, navigable trace links between the available software artifacts. A widely used definition for an ontology is the following definition by Grüber:

*"An **ontology** is an explicit specification of a conceptualization. An ontology specifies the concepts, relationships, and other distinctions that are relevant for modelling a domain." (Grüber, 1995)*

In this research, we envision 'ontological traceability for software', where we use NLP to construct an ontology for a software product, and we use this ontology to interrelate different software artifacts in an automated manner. We aim to investigate whether this approach will ultimately be able to achieve the goal of ubiquitous software traceability. We define ontological traceability for software as follows:

*"**Ontological traceability for software** is the automated creation and maintenance of software artifact trace links by means of an ontology, that can be used to answer questions about the software product and its development process."*

In this research, we take a first step in the realization of ontological traceability for software, by presenting a proof-of-concept in which we relate a set of software artifacts using a domain ontology. We have baptized the domain ontology that describes a software product the **Product Domain Ontology**. As a basis for this ontology, we use conceptual models that have been learned from user story requirements, as presented by Lucassen et al. (Lucassen et al., 2016a)

1.2 Research Objective

The goal of this research is to investigate whether the advantages that the ontological approach to software traceability pose can be realized in practice. The hypothesis of this research is that the ontological approach can ultimately achieve ubiquitous software traceability.

The main artifact that is delivered by this research is a proof-of-concept of ontological traceability for software, using the Product Domain Ontology to create trace

links between a set of software artifacts. Next to demonstrating ontological traceability for software, this proof-of-concept will also demonstrate the key advantages that the ontological approach has in comparison with other automated approaches to software traceability.

1.3 Research Structure

This research has the following structure: In the next chapter, the research method is outlined, after which, an extensive literature review is presented in chapter 3. A vision for linguistic tooling in combination with software development is presented in chapter 4. After which, in chapter 5, we propose two theories: A theory for the Product Domain Ontology, and a theory for the usage of the Product Domain Ontology for the creation of trace links. We present a vision on how these theories can be implemented in a large revision management platform, including the features that software traceability enables. Chapter 7 explains how the theories proposed in chapter 5 are implemented in a proof-of-concept: The Ubiquitous Traceability Artisan (UTA). In chapter 8, we evaluate the performance of the UTA, after which we discuss our findings in chapter 9, together with the conclusions of this research project. We also propose some future research projects, to continue on this research project.

Chapter 2

Research Approach

2.1 Research Problem

The following research problem elaborates the problem context and the goals of the stakeholders. The research problem also displays how the designed artifact realizes these goals.

“Improve software development, by means of ontological traceability for software, such that traceability becomes ultimately ubiquitous, in order to fully utilize the benefits of software traceability.”

The context of the research project is elaborated in the information systems research framework depicted in Figure 2.1. The artifact that will be developed in this research project aims to fulfill the business needs of software development companies, as well as software development tooling companies. It does so by applying knowledge from the fields of software traceability, software production and ontologies.

2.2 Research Questions

In order to realize ontological traceability for software, research must be done on how to establish trace links between different software artifacts, using an ontology.

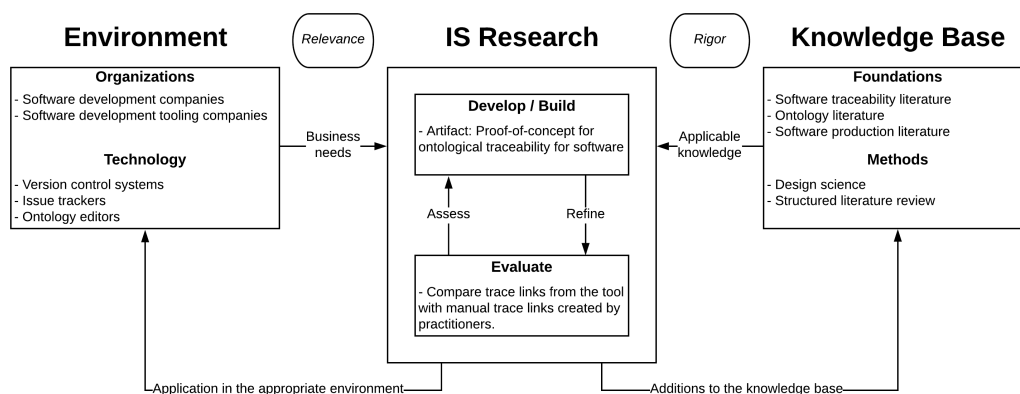


FIGURE 2.1: Information Systems Research Framework

This research goal is illustrated in the main research question of this research project:

RQ: "How can bidirectional trace links between software artifacts be realized using an ontology?"

The artifact produced in this research project, is a proof-of-concept for ontological traceability for software. In order to determine the trace links that will be present in this proof-of-concept, two sub-questions have been formulated. The goal of these sub-questions is to investigate the possibilities of software traceability, and to investigate which trace links are of interest to practitioners. The most relevant trace links will be implemented in the proof-of-concept.

SQ1: "Which trace links can possibly be found between software artifacts and an ontology?"

SQ2: "Which trace links between software artifacts are of the greatest interest by practitioners?"

The ontology that is generated from software artifacts needs to be visualized and stored, so that it can be used by the proof-of-concept. Furthermore, the resulting ontology needs an all-embracing name. The following research question has been formulated for these tasks:

SQ3: "What is the optimal name, visualization, and storage mechanism for an ontology that serves software traceability?"

The first three sub-questions can be answered by a means of a literature study and interviews with practitioners. The actual creation of the proof-of-concept is a design problem, for which two sub-questions have been formulated. These two sub-questions can be seen as two sides of the same coin. This is necessary because of the 'bidirectional' property that the trace links need to have.

SQ4: "How can a trace link between an ontology and a set of software artifacts be realized?"

SQ5: "How can a trace link between a software artifact and an ontology be realized?"

Once we can generate trace links, we want them to be beneficial to practitioners, which is why we also investigate the integration of software traceability in a popular collaborative revision tool: GitLab. This design is guided by the following research question:

SQ6: "How can trace links provide benefits to practitioners in a collaborative revision tool such as GitLab?"

After the proof-of-concept has been implemented, its performance will be evaluated through an experiment. The proof-of-concept will generate a set of trace links for a selected number of software artifacts, in a real software product provided by a company. The evaluation process is guided by the following research question:

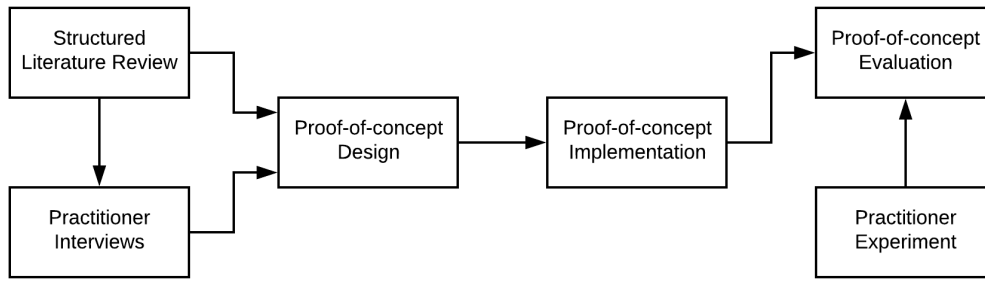


FIGURE 2.2: Research Method

SQ7: "How well does the proof-of-concept for ontological traceability for software perform in comparison with humans?"

In the next section, a research method is proposed in order to answer the main research question and the sub-questions that are stated above.

2.3 Research Method

Sub-questions SQ1, SQ2 and SQ3 are answered using findings that resulted from a structured literature review. In addition to the literature review, an open format interview with practitioners at the company where the proof-of-concept is realized provides additional insights for the answering of SQ2.

The research method that is followed in this research is the design science method for information systems and software engineering, as described by Wieringa. (Wieringa, 2014) The design of the proof-of-concept answers sub-questions SQ4 and SQ5, after which the proof-of-concept will be implemented, and tested using a data set provided by the company Bytestack. We propose ways to use the trace link in a collaborative revision tool by answering SQ6. The evaluation of the proof-of-concept is guided by sub-question SQ7, and will be done using an experiment. The experiment design is elaborated in the next section. After the evaluation has taken place, a number of conclusions can be drawn on ontological traceability for software.

A full overview of the research method can be seen in the diagram in Figure 2.2.

2.3.1 Experiment Design

To evaluate the performance of the proof-of-concept for ontological traceability for software, an experiment will be conducted. A questionnaire will be distributed among practitioners at the Bytestack company. The questionnaire will exist of questions where 2 software artifact instances are compared to each other, to see if the practitioner would establish a trace link between these artifacts. The artifact instances are part of a software product that has been created by Bytestack. The

questionnaire is analyzed, which results in a set of trace links that the practitioners would create for the software product.

Next, the proof-of-concept will analyze the software artifact instances, and will generate a set of trace links using the ontological approach. This set of trace links can then be compared to the trace links created by practitioners, in order to analyze the performance of the proof-of-concept. The results of this analysis allows for the answering of sub-question SQ7.

2.4 Literature Research Protocol

The literature research protocol (figure 3) for this research project consists of four phases. In the first phase, important literature reviews and roadmaps are identified, to provide a solid baseline for the literature study. A paper is selected for each of the following identified research fields: Software traceability, ontologies, and the application of ontologies in software production. After the identification of the important papers, snowballing is used to find the relevant literature in each research field, in order to get an overview of what has already been achieved in the research field. Reverse snowballing is applied afterward to identify recent advances in the research field. The final phase of the literature research protocol is the identification of missing papers by using search strings. This ensures that the most important literature is covered by the literature review. The following search strings have been used:

- Ontology AND (Learning OR “Learning from Text” OR Storage OR Visualization)
- (Software OR Requirements) AND Traceability
- “Natural Language Processing” AND (Ontology OR “Requirements Engineering”)

Due to the large amount of search results these search strings deliver, only the 10 most cited papers have been analyzed for their potential relevance to the research project.

Chapter 3

Theoretical Foundation

The first research activity in the field of software traceability can be ‘traced back’ to 1994, when Gotel and Finkelstein published a study on the ‘requirements traceability problem’. Gotel and Finkelstein defined requirements traceability as follows:

“Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction.” (Gotel and Finkelstein, [1994](#))

Requirements traceability can be seen as a specific case of software traceability, where we only observe the trace links that exist between the requirements artifacts and other software artifacts. This definition was later extended by Gotel et al. to include traceability between any different software artifact, rather than just the traceability of requirements. (Gotel et al., [2012a](#))

After years of research, the tights within the community have been strengthened by the formation of the Center of Excellence for Software Traceability (CoEST). The combined efforts of the researchers related to the CoEST eventually lead to the publication of a roadmap for software traceability research. (Gotel et al., [2012b](#)) The roadmap provides seven goals for software traceability: Purposed, cost-effective, configurable, trusted, scalable, portable and valued software traceability. (Cleland-Huang et al., [2014](#)) Furthermore, they specify a single grand challenge: Ubiquitous traceability, which means that traceability is always there, without ever having to think about getting it there. The complete definition of ubiquitous software traceability can be seen in the introduction chapter of this thesis. With the formulation of the ubiquitous traceability goal, the CoEST envisioned the complete automation of software traceability. In order to achieve ubiquitous traceability, it must fulfill all the requirements of the seven goals for software traceability. Therefore, the seven goals will be described briefly in the next section. These descriptions are a paraphrasing of the descriptions provided by Cleland-Huang et al. in their paper on the trends and future directions of software traceability. (Cleland-Huang et al., [2014](#))

The first goal is ‘purposed’ traceability, which means that the traceability is requirements driven. When stakeholder requirements for traceability are clearly defined and measurable, it can be demonstrated that the traceability is fit-for-purpose. The second goal states that traceability must be ‘cost-effective’, meaning that the

return on investment from using traceability is clear and outweighs the costs of establishing it. Naturally, in the special case of ubiquitous traceability, the costs of establishing traceability should be near zero. The third goal states that traceability must be ‘configurable’, meaning that trace links can be configured to suit the current needs of the project. Furthermore, trace links must be able to be reconfigured, in order to match changing stakeholder needs. The fourth goal states that traceability must be ‘trusted’ by stakeholders. This means that the quality of the traceability must be maintained, in order for the stakeholders to be able to depend on it. The fifth goal states that traceability must be ‘scalable’, meaning that varying types of artifacts can be traced, and across organizational and business boundaries. The sixth goal states that traceability must be ‘portable’. This means that the established traceability can be reused across projects, organizations, domains, product lines and supporting tools. This goal is especially relevant in this research, as the ontological approach appears to suit the portability of traceability across different domains. The seventh and final goal is ‘valued’ traceability, meaning that the stakeholders see traceability as a strategic priority. For this goal, it is essential that the business benefits of traceability become clear and accessible to all stakeholders.

By achieving these seven goals in traceability, a significant step to the realization of the grand challenge, ubiquitous traceability, is made. The assumption of this research is that ubiquitous software traceability can potentially be achieved using natural language processing techniques. When domain knowledge is properly extracted from the different software artifacts, relations between them can be identified. In this research project, we represent the domain knowledge extracted from the artifacts as an ontology.

Guarino provides a description of the terminology of the domain of ontology and ontologies. (Guarino, 1998) Guarino makes an important distinction between four different types of ontologies:

- **Top-level Ontology**
Describe very general concepts like space, time, object etc., which independent of a particular problem or domain.
- **Domain Ontology**
Describe the vocabulary related to a generic domain, by specializing the terms introduced in the top-level ontology.
- **Task Ontology**
Describe the vocabulary related to a generic task or activity, by specializing the terms introduced in the top-level ontology.
- **Application Ontology**
Describe concepts depending both on a particular domain and task, which are often specializations of both the related ontologies.

A graphical representation of relation between the different types of ontologies can be seen in Figure 3.1. This figure was first proposed by Guarino in 1997. (Guarino, 1997)

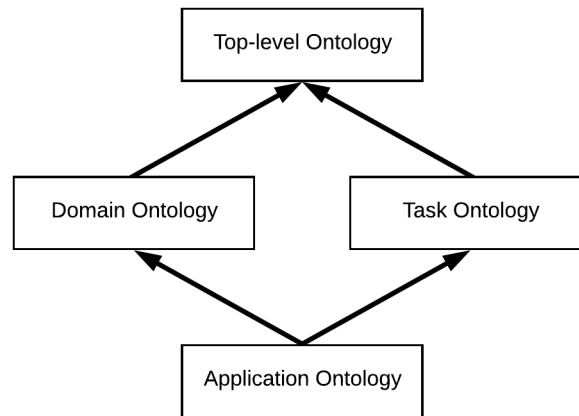


FIGURE 3.1: Ontology Types (Guarino, 1997)

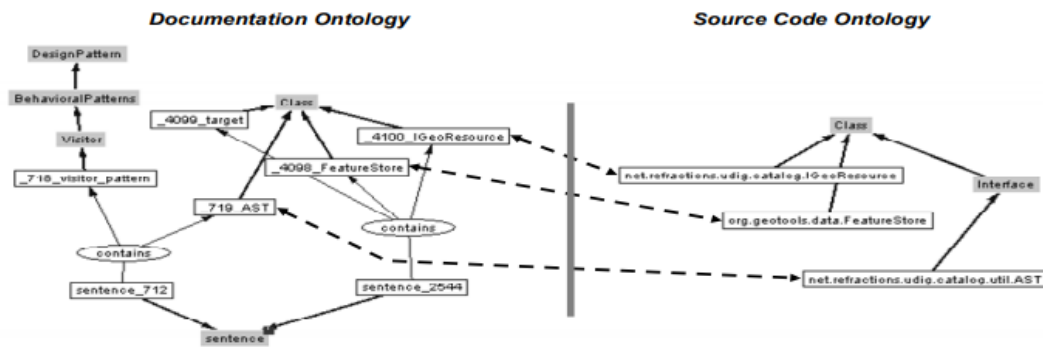


FIGURE 3.2: Ontology Matching (Zhang et al., 2008)

Ontologies allow us to identify the trace links, by comparing the ontologies that have been constructed from the different artifacts. The applicability of this approach has already been demonstrated by Zhang et al. (Zhang et al., 2008) They present an ontological approach to the retrieval of trace links between source code and documentation. Their work relies on an ontology that has been explicitly designed for text mining, rather than an ontology created from automatic construction. This is where the research of Zhang et al. diverges from our research. (Zhang et al., 2008) An interesting statement they make is that current methods for automated ontology construction do not provide the necessary level of detail for software traceability. They however do believe that a future combination of both approaches is the optimal solution to ontological traceability. An example of the use of ontologies in the work of Zhang et al., can be seen in Figure 3.2.

This section provides an elaborate overview of the related literature in both the fields of software traceability and ontologies. It serves as a theoretical framework

for this research project, and illustrates that there is a gap in the literature. Furthermore, it contains an overview of additional literature on the application of ontological traceability for software in software production. A timeline of software traceability research is provided in the section below.

3.0.1 The History of Software Traceability Research

- Early 90's: Extensive study with industrial practitioners by Gotel and Finkelstein.
 - Title: "An analysis of the Requirements Traceability Problem." (Gotel and Finkelstein, [1994](#))
 - Contribution: Highlights several traceability challenges, especially regarding the practice and processes of establishing traceability in an industrial setting.
- 2001: A study of requirements traceability in practice by Ramesh and Jarke.
 - Title: "Towards reference models of requirements traceability." (Ramesh and Jarke, [2001](#))
 - Contribution: Traceability metamodels for various software engineering tasks.
- 2002: A first attempt at achieving automated software traceability by Antoniol et al.
 - Title: "Recovering traceability links between code and documentation." (Antoniol et al., [2002](#))
 - Contribution: Describes the use of information retrieval techniques to automatically construct trace links between documentation and code.
- 2005: The Center of Excellence for Software Traceability is launched by a group of researchers. The goal was to identify the grand challenges of software traceability, and to create enthusiasm in the research community, in order to advance the field. This endeavour led to a series of workshops.
- 2012: The Grand Challenges of Traceability is formulated by the CoEST, in two papers.
 - Title: "The Grand Challenge of Traceability. (v1.0)" (Gotel et al., [2012a](#))
 - Title: "The quest for ubiquity: A roadmap for software and systems traceability research." (Gotel et al., [2012b](#))
 - Contribution: A grand challenge for software traceability research, as well as a set of quality goals for software traceability. This paper failed to fully describe the practical and technical research challenges for the community.
- 2014: The shortcomings of the roadmap are presented in a paper by a smaller group of researchers, affiliated with the CoEST.
 - Title: "Software Traceability: Trends and Future Directions." (Cleland-Huang et al., [2014](#))

- Contribution: A set of specific research areas, mapped to quality goals, traceability processes, and open technical challenges. These challenges are grouped into three main areas of planning and managing, creating and maintaining traces, and using traces.
- 2017: The Grand Challenges of Traceability event brings the research community together, to look back on the past decade of research, and to identify two specific challenges to be addressed in the next decade.
 - Title: “Grand Challenges of Traceability: The Next Ten Years.” (Antoniol et al., 2017)
 - Contribution: An overview of the research of the past decade, and specific research goals for the following decade.

3.1 Software Traceability in Practice

Back in 1994, Gotel and Finkelstein conducted a series of empirical studies among over a 100 practitioners, on the subject of requirements traceability. (Gotel and Finkelstein, 1994) They observed a weak support for requirements traceability in tools, and state that requirements traceability is unfortunately not used very often in practice. They propose a definition (as seen in the previous section), and a preliminary research agenda for requirements traceability. Furthermore, they provide an outline of the (manual) techniques for requirements traceability available at the time.

Other studies among practitioners in the late 1990’s and early 2000’s yield similar negative results. Ramesh reports some critical factors in the adoption of requirements traceability. This includes the finding that managers claim they do not derive much benefit from traceability practices, and see it as a necessary evil. Furthermore, there appears to be a lack of interest from developers, because of the lack of organizational commitment and management support. (Ramesh, 1998)

In 2007, Blaauboer et al. investigated the organizational process of deciding to adopt requirements traceability. This resulted in the remarkable finding that project leaders in the case study were not aware of the concept of traceability, which is why it was never even considered. Another interesting finding is that the benefits for traceability reside mostly outside the different projects, meaning that there is little incentive for project managers to properly embrace traceability practices. (Blaauboer, Sikkel, and Aydin, 2007)

Fortunately, not all research on the adoption of software traceability in practice has yielded negative results. Mäder and Egyed have conducted an experiment in 2011 among 52 subjects, that compared the performance of developers on software maintenance tasks, where half of the subjects had access to traceability navigation tools. Their main finding is that traceability between requirements and code had a significant effect on the quality of the change tasks executed by the subjects, showing that participants were 21% faster and 60% more correct. Participants also adopted

traceability without any training. The study suggests that traceability has the ability to improve basic software maintenance tasks. (Mäder and Egyed, 2011)

Bouillon et al. provide a set of usage scenarios for requirements traceability, based on a survey among practitioners. Their main finding is that practitioners use traceability mainly for the following tasks: (Bouillon, Mäder, and Philippow, 2013)

- Finding the origin and rationale of requirements
- Documenting a requirement's history
- Tracking requirement or task implementation state

However, the practitioners did mention that they struggle with the bad cost-benefit ratio of the traceability practices. This finding is supported by Arkley and Riddle, who also report a lack of perceived direct benefit for the software development processes. This is often referred to as the 'traceability benefit problem', which means that the people establishing the traceability are not the ones who experience the benefits from it. (Arkley and Riddle, 2005)

It becomes apparent from the literature that even though traceability might have benefits for a software development project, it doesn't always outweigh the costs, due to the large amount of manual work necessary in the creation and maintenance of trace links. This observation is reflected by two of the goals set by the CoEST, namely the 'cost-effective' goal and the 'valued' goal, which have been described in section 3. (Cleland-Huang et al., 2014) The case studies among practitioners that have been conducted so far, for example the case studies by Mäder et al. show that these goals are not yet achieved in practice, and more research needs to be done in order to fully achieve these goals. The benefits reported by the surveys among practitioners suggests that practitioners would greatly benefit from a fully automated software traceability solution. (Mäder and Egyed, 2011)

A deeper understanding of the concept of software traceability, its application in practice, and related terminology can be obtained from the book 'Software and Systems Traceability', edited by Cleland-Huang, Gotel and Zisman. (Cleland-Huang, Gotel, and Zisman, 2012) The first section of the book provides a glossary of the related terminology of the domain of software traceability, the costs and benefits of traceability, and the acquiring of tool support for traceability. The following two sections describe traceability creation and traceability maintenance, outlining the different available techniques for the process. The fourth section outlines different usage scenarios for traceability, for example in combination with agile software projects, and the tracing of non-functional requirements. The final section of the book poses the grand challenges for software traceability. This section contains one of the papers that has been cited before in this document, by Gotel et al., which first described the grand challenge of ubiquitous traceability. (Gotel et al., 2012a)

As can be seen in the section above, the general consensus in the literature is that software traceability can be of great value to the stakeholders of a software product under development. However, due to the traceability benefit problem (Arkley and

Riddle, 2005) and the lack of tool support for software traceability (Bouillon, Mäder, and Philippow, 2013), most practitioners do not adopt it. A possible solution is the automation of software traceability, which is studied in this research project.

3.2 Automated Software Traceability

One possible way to improve the cost-effectiveness of software traceability is to fully automate trace link creation and maintenance. The earliest attempts at automated software traceability date back to the early 2000's. Egyed and Grünbacher describe the 'Trace Analyzer' method, which takes known or hypothesized dependencies between artifacts, and establishes a trace if the artifacts have an overlapping node in a common ground (for example, source code). It then builds a graph of the nodes that contain these overlaps. The method however doesn't result in fully automated software traceability, but aides the developer in the process of establishing trace links. (Egyed and Grünbacher, 2002)

Most of the research into automated software traceability investigates a single type of trace link, between two software artifacts. The early work of Antoniol et al. (Antoniol et al., 2002), an information retrieval (IR) method to recover trace links between source code and free text documentation is described. In another paper, Antoniol et al. propose a technique for the recovery of trace links between design models and source code. (Antoniol et al., 2001)

In 2014, Borg et al. published a systematic literature review of all research into IR approaches to software traceability, providing an excellent overview of the current state-of-the-art. (Borg, Runeson, and Ardö, 2014) One of their findings is that the majority of IR methods aim to establish horizontal trace links between requirements, or trace links between requirements and source code. Another remarkable finding is that the documentation of design decisions in the development of IR methods for software traceability is rather poor. This means that replicating a certain experiment on the effectiveness of the specific IR method can lead to different outputs. One of the honorable mentions in the literature review is the work of De Lucia et al., who have established a more elaborate IR-based software traceability, containing trace links between use cases, functional requirements, source code and tests cases. (Lucia et al., 2007) Their most interesting finding is that while IR methods provide a useful support to the identification of trace links, they are not able to identify all trace links. The major problem with IR based methods, is the high amount of false positives they yield, forcing the software engineer to manually check the trace links. (Lucia et al., 2007) This has a large impact on the cost-effectiveness of the IR based traceability methods, decreasing their potential for the grand challenge of ubiquitous traceability.

As can be seen in the literature overview in this section, the majority of the research activities in the field of automated software traceability focuses on the use of IR approaches to automated software traceability. Even though there are a lot of

research efforts into this approach, the implementations appear to be disconnected from each other. As the IR based methods to software traceability do not have the potential to identify all trace links (Lucia et al., 2007), we prefer an ontological approach in this research project, which has the potential to provide a more generic, ubiquitous solution.

3.3 Ontological Traceability for Software

The potential solution for ubiquitous software traceability that is investigated in this research project is the ontological approach. In this approach, an ontology is constructed from different software artifacts, and this ontology is then used as a common ground to establish trace links between different software artifacts. The ontology will be constructed automatically, through a process called ontology learning, which is elaborated upon in section 3.5. One previous research effort has taken a similar approach. Zhang et al. present an approach for the semantic recovery of trace links between source code and documentation, using ontologies. (Zhang et al., 2006) (Zhang et al., 2008) In the approach of Zhang et al., an ontology is created for each artifact, and trace links are established using ontology alignment. One of the main findings is that the resulting ontological model can support software maintenance. As stated before, the approach of Zhang et al. diverges from our approach because the ontologies have been constructed from a manual design, rather than an automated solution.

Learning ontologies from software artifacts has been researched before as well. Bontcheva et al. present an approach that learns an ontology from source code and documentation software artifacts. (Bontcheva and Sabou, 2006) This ontology is then used to aid the exploration through the different software artifacts in the system. This can be seen as a form of trace link usage, however the term software traceability is not used in the work. The ontology is only used as a semantic navigation tool, which is only a specific usage scenario for software artifact trace links.

Noll et al. propose the integration of ontologies in the Unified Process, in order to provide concept-based traceability. (Noll and Ribeiro, 2007) Even though the same techniques are used, the approach is conceptually different from ontological traceability for software, as only a single ontology for the Unified Process has been created, regardless of the software product domain. This is a substantial difference, as this ontology only shows the concepts and relations that exist in relation to software development artifacts, which our approach creates an ontology from the domain knowledge in the software artifacts. The ontology proposed by Noll et al. is a top-level ontology, which is an ontology that describes general concepts, that are independent of a particular domain. The ontology proposed this research is a domain ontology, which describes the concepts and relations for a specific domain. (Guarino, 1998) We use a domain ontology, as we are interested in creating a central source of truth between the software artifacts, containing all the relevant domain knowledge.

Despite its potential, research on the use of ontologies in software traceability is very limited. This research project aims to fill this gap in the literature by presenting a proof-of-concept for ontological traceability for software.

3.4 User Stories

When it comes to automated software traceability, the requirements specification software artifact is of particular interest in the literature. A user story is a semi-structured approach to the specification of requirements, that captures who it is for, what it expects from the system, and (optionally) why it is important. (Wautelet et al., 2014) Most practitioners use the so called Connextra template for user stories. (Lucassen et al., 2016c) The Connextra template can be seen below, as well as an accompanying example user story.

“As a type of user, I want goal, [so that some reason].” (Lucassen et al., 2016c)

Example: “As an event organiser, I want to sell tickets, so that my customers can attend my event.”

The simplicity and structure of user stories makes them particularly usable for natural language processing. The possibilities for user stories and natural language processing are extensively analyzed in the dissertation of Garm Lucassen (Lucassen et al., 2017). This research project builds on his work, and therefore takes user stories as a starting point for ontological traceability for software. His work, together with fellow researchers, includes a behaviour driven method for ubiquitous software traceability between requirements and source code, as well as the extraction of conceptual models from user stories. (Robeer et al., 2016) The conceptual models described by Lucassen et al. can also be defined as a lightweight ontology. A lightweight ontology is a directed graph containing concepts and relations, but no axioms. (Wong, Liu, and Bennamoun, 2012) This lightweight ontology is extracted from a set of user stories using natural language processing techniques. (Lucassen et al., 2016a) An example of such a lightweight ontology, can be seen in Figure 3.3.

3.5 Ontology Learning from Text

Buitelaar et al. provide an overview paper for the field of ontology learning from text. They make a clear distinction between ontology population and ontology learning, defining both processes as follows:

“Ontology population is the process of defining and instantiating a knowledge base.”

“Ontology learning is the (semi-)automatic support in ontology development.” (Buitelaar, Cimiano, and Magnini, 2005)

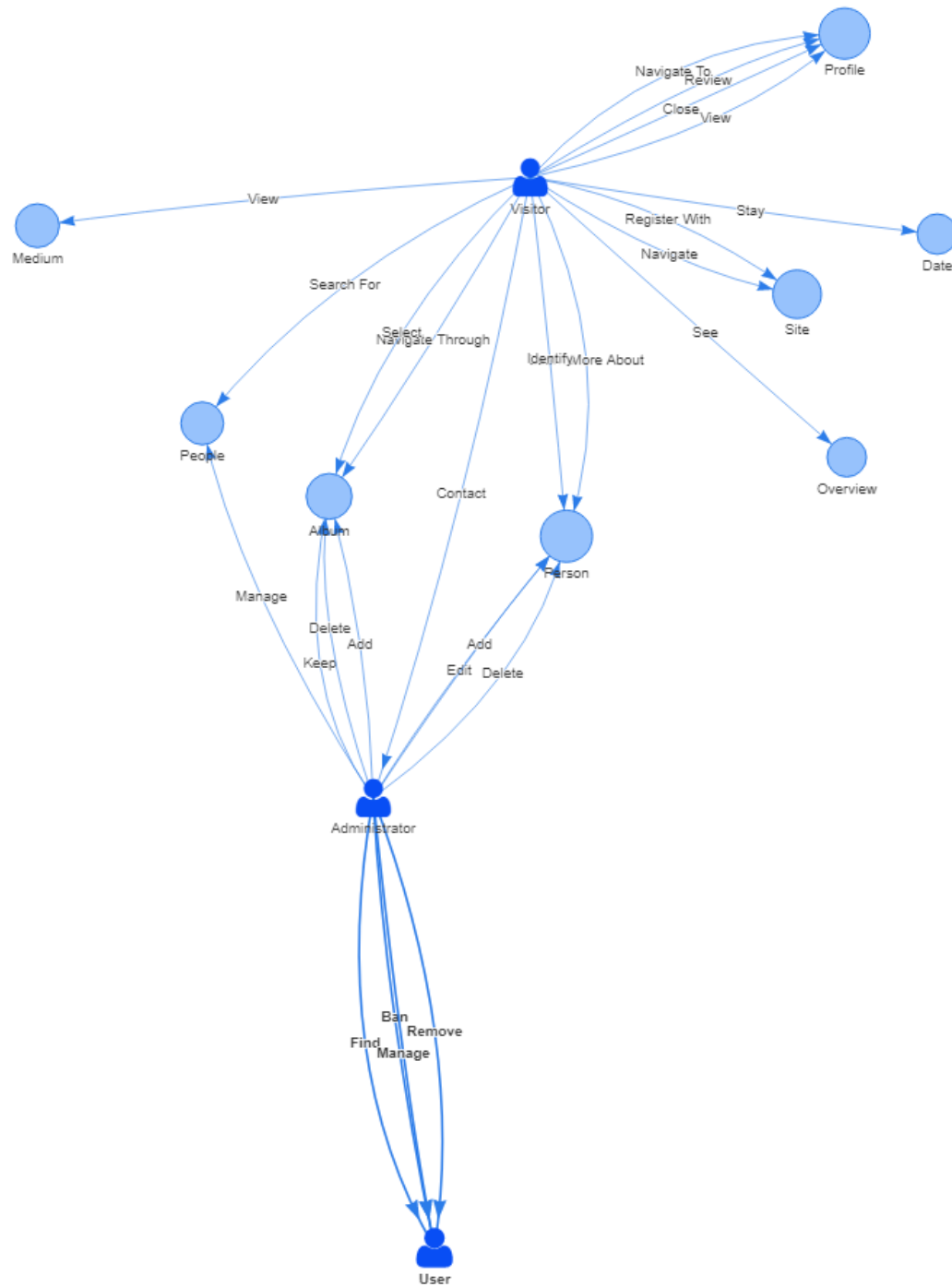


FIGURE 3.3: Lightweight Ontology Example

They present a layered architecture for ontology learning, containing 6 different subtasks. This model was baptised as the ‘Ontology Learning Layer Cake’, and it can be seen in Figure 3.4. It consists of the following ontology learning subtasks:

- **Terms (term extraction):** Terms are linguistic realizations of domain-specific concepts, and are therefore central to further more complex tasks.
- **Synonyms:** Addresses the acquisition of semantic term variants in and between languages.
- **Concepts:** Extraction of an intensional definition of the concept, in combination with a set of linguistic realizations.
- **Concept hierarchies (taxonomies):** Extraction of ‘is-a’ relationships between concepts.
- **Relations:** Extraction of connections between concepts.
- **Rules (axioms):** Extraction of rules that define which facts can be stated from the ontology.

The descriptions of the subtasks above are paraphrased from the work of Buitelaar et al. The paper also provides references that present various methods that address certain aspects of the Ontology Learning Layer Cake.

A more recent paper on the state-of-the-art in the field of ontology learning from text was published in 2012 by Wong et al. (Wong, Liu, and Bennamoun, 2012) They provide an excellent overview of the current techniques in ontology learning in general, such as statistics-based techniques, linguistics-based techniques and logic-based techniques. Furthermore, they present some outstanding challenges in the field, such as the extensibility of lightweight ontologies to full-fledged formal ontologies. Furthermore, they provide an overview of the most popular ontology learning systems, such as ‘Text-to-onto’ (Cimiano and Völker, 2005), ‘OntoLearn’ (Missikoff, Navigli, and Velardi, 2002), and ‘OntoGain’ (Drymonas, Zervanou, and Petrakis, 2010). Finally, they provide a short literature review of previous overview papers on ontology learning from text (with an exception to the paper from Buitelaar et al.), making this paper the most extensive overview of the research field. One of the most important findings is that the performance of term extraction has stabilized, with a score generally above 90%. Next to that, the paper states that larger samples generally improve the performance of term extraction. A final important lesson from the paper is that the majority of the ontologies that are automatically learned, are lightweight ontologies, meaning that the ontologies do not contain axioms. This can be explained, as the ‘rules’ ontology learning subtask is the least addressed research area in ontology learning. (Buitelaar, Cimiano, and Magnini, 2005)

3.6 Ontology Visualization

One of the applications of a software product ontology, is to allow practitioners to explore a large set of requirements in a short period of time. However, ontologies

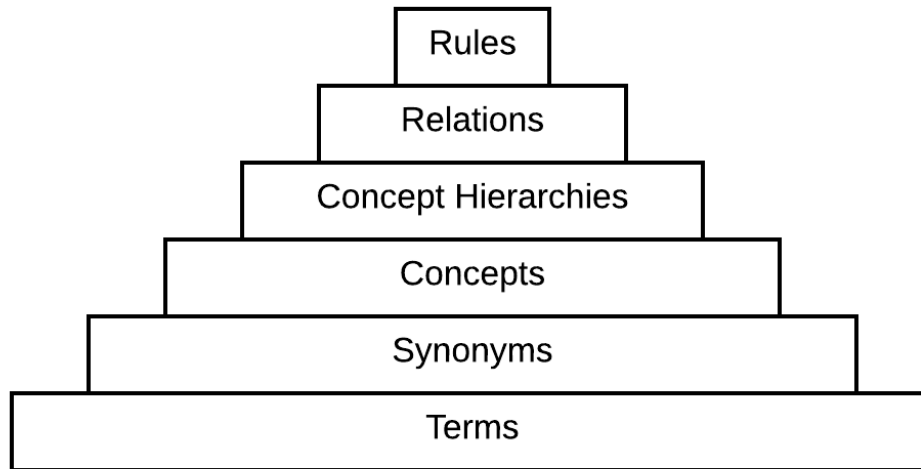


FIGURE 3.4: Ontology Learning Layer Cake (Buitelaar, Cimiano, and Magnini, 2005)

quickly become too large to be explored, a phenomenon known as cognitive overload. In order to prevent this from happening, a proper ontology visualization technique is necessary. This section outlines the research that has been done on ontology visualization.

Katifori et al. have provided a survey on ontology visualization methods in 2007. (Katifori et al., 2007) One of their main findings is that there is no optimal solution for each use case when it comes to ontology visualization, and so the chosen method or tool should provide the user with a choice of multiple solutions. Katifori et al. mention Protégé as an ontology management tool that supports several visualization methods. Another important conclusion is that ontology visualization should be coupled with effective search tools or querying mechanisms, as it becomes difficult to locate a specific instance once the ontology grows very large.

In 2011, a review was published by Sivakumar et al., on the ontology visualization tools in Protégé. (Sivakumar, Arivoli, and Sri, 2011) The authors state that Protégé is one of the most popular ontology visualization tools, and classify the different types of Protégé ontology visualization tools into the following four different categories:

- **Indented List:** Represents the taxonomy of the ontology as a file system explorer-tree view.
- **Node-link and Tree:** Represents the ontology as a set of interconnected nodes.
- **Zoomable:** Extends the node-link and tree visualization by hiding lower levels of the hierarchy, revealing them once the user zooms in on a parent node.
- **Focus+context:** Extends node-link and tree, by showing the node of interest in the center of the view. The other nodes are displayed around the center node.

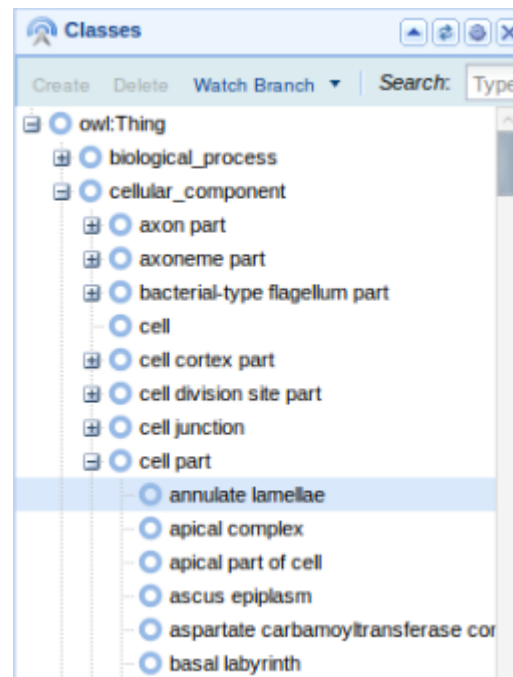


FIGURE 3.5: Indented List Visualization

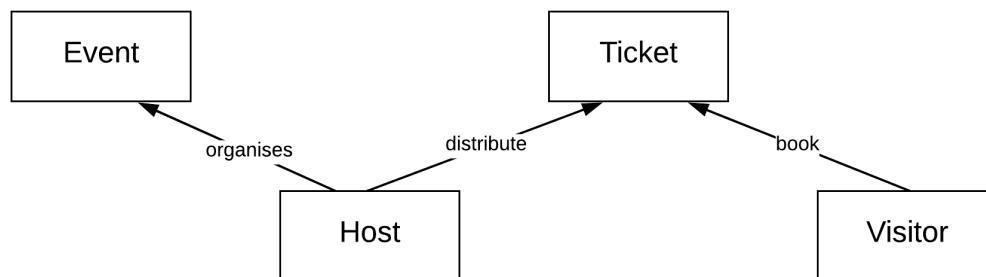


FIGURE 3.6: Node-link and Tree Visualization

The characteristics of the different visualization tools are categorized in this work. The main conclusion is that there is no visualization tool that is the most appropriate for all applications. An example for the indented list and node-link and tree visualizations are displayed in Figures 3.5 and 3.6. An example for the node-link and tree, zoomable and focus+context is displayed in Figure 3.7.

Another roadmap was published in 2014, by Dudás et al. (Dudáš, Zamazal, and Svátek, 2014) They analyzed the current ontology visualization tools, and created a recommender tool, in order to find the perfect visualization tool for the desired application. One particular new tool of interest is the Visual Notation for OWL Ontologies (VOWL) (Lohmann et al., 2016), a very user-centered visualization tool, that also comes with a Protégé plugin. An example of the VOWL visualization can be seen in figure 8. Lohmann et al. provide an evaluation of VOWL, concluding that it

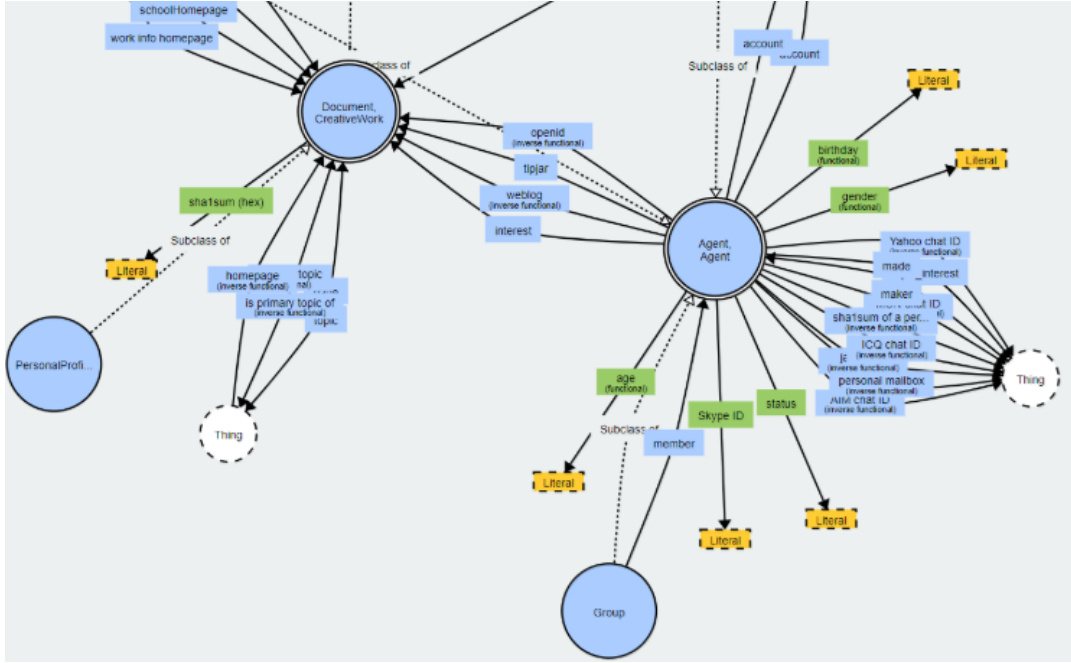


FIGURE 3.7: Graph Visualization in OWL

produces comparatively intuitive and comprehensible ontology visualizations. Another advantage of OWL is that it also comes with a web-based visualization solution (WebOWL), and an ontology querying solution (QueryOWL). (Lohmann et al., 2016)

The set of OWL tools and plugins is open source, under the MIT licence.

3.7 Ontology Storage

The majority of ontology management tools already provide functionality for ontology storage. The storage mechanisms can either be native (based on the file system), or database storage. The decision between a native or a database storage is a trade-off: Native storages usually have a reduced load time, whereas database storages can provide query optimization features. (Lu et al., 2007) Lu et al. propose a system for ontology storage, search and reasoning, called 'SOR' (Scalable Ontology Repository). Zhou et al. propose Minerva, which is a scalable OWL ontology storage and inference system. (Zhou et al., 2006) Both SOR and Minerva can be used to store an ontology. Reasoning can then be performed on this ontology, as both methods support the SPARQL query language. Furthermore, the storages support practical ontology management through ontology navigation.

In this research project, the ontology will be stored through Protégé, using either its native solution, or one of the storage solutions that are provided through plugins. There are many storage plugins for Protégé, for example, del Mar Roldan-Garcia et al. describe a plugin to store an OWL ontology in Protégé in a relational database.

(Mar Roldan-Garcia and Aldana-Montes, 2008) Furthermore, there are other plugins that interface the storage to a specific type of relational database, such as an Oracle database.

3.8 Ontology Mapping

Later in this research project, we want to compare small sub-ontologies, that relate to a single software artifact instance, to each other. This is a procedure called 'ontology mapping'. A plethora of method for ontology mapping already exist, and Choi et al. provide an excellent overview of the state-of-the-art. (Choi, Song, and Han, 2006)

Choi et al. divide the types of ontology mapping methods into three different categories:

- **Ontology mapping between an integrated global ontology and local ontologies**
Methods that support the ontology integration process. Results in mappings between a global ontology, and smaller, local ontologies.
- **Ontology mapping between local ontologies**
More flexible methods that provide mapping functionalities for highly dynamic environments.
- **Ontology mapping (matching) in ontology merging and alignment**
Methods that take several smaller ontologies, and deliver a single, large ontology.

As we are looking to compare smaller, local ontologies in this research, we need a method of the second category. Two of these methods, the GLUE method (Doan et al., 2002) and the QOM method (Ehrig and Staab, 2004), can yield the desired output: A similarity score for two ontologies.

The amount of sub-ontologies that we have to compare in this research, increases exponentially as the amount of software artifact instances grows. Therefore, an efficient method with low computing time is necessary. The QOM method provides this functionality: It is a relatively simple comparison algorithm, that has a lower run-time complexity due to a dynamic programming approach. (Choi, Song, and Han, 2006) Even though the approach favors efficiency, experiments show that the loss in quality is marginal. (Ehrig and Staab, 2004)

3.9 Literature Review Conclusions

One of the main conclusions that can be drawn from the literature review is that **the ontological approach to software traceability that we envision is a unique one.** Previous attempts to use ontologies for software traceability, such as the approach by Zhang et al. (Zhang et al., 2006), differ significantly from our approach, because of the type of ontology that is used. Zhang et al. use a top-level ontology, whereas

we use a domain ontology. Furthermore, Zhang et al. only address code and documentation artifacts, whereas our approach is applicable to all artifacts.

Another conclusion that can be drawn from the literature review, is that **none of the automated approaches to software traceability have come close to ubiquitous traceability** so far. This should not come as a surprise, as the researches that have set this goal for software traceability believe that it should be achieved around 2035. (Gotel et al., [2012a](#)) Whilst our proposed solution might not achieve ubiquitous software traceability anytime soon, we do hypothesise that it has the potential to eventually achieve it.

A final interesting conclusion that can be drawn from the literature review, is that **practitioners do not seem to attach any value software traceability**. It is our hypothesis that this is mainly caused by a lack of integration in popular software development management toolkits. When traceability will become available through highly used platforms, they might be more inclined to make use of its features. Therefore, it is our goal to eventually integrate our work in a popular software development management toolkit such as GitLab.

Chapter 4

Linguistic Tooling for Software Development: A Vision

In this chapter, a vision is presented for software development using integrated linguistic tooling. This vision addresses software traceability, but also displays the other applications of linguistic tooling for software development. In summary, we envision a future where natural language processing techniques improve the overall quality of software development, and makes the life of developers and stakeholders easier.

4.1 Problem Statement

Software product development is a complex task that involves the creation and maintenance of a multitude of software artifacts. Software developers gain an increasing amount of domain knowledge through the development process of the software product. Furthermore, they become increasingly aware of implementation details as the project goes along.

The majority of the knowledge that is required by software developers is often only stored in the brains of the software developers. The artifacts that are created in the software development process are often disconnected, which means that a lot of domain knowledge is required to extract information from these artifacts. A clear example of this disconnectedness is the relation between requirements and test cases. A test case usually covers a section of the code, and asserts whether its behaving in the expected manner. When a developer wants to check if a section of the code behaves as expected, the developer can check this easily by finding the appropriate test. But when a product manager wants to see whether a requirement is satisfied by the system and displays the expected behaviour, a more thorough search action is required: The product manager has to consult the developer with the appropriate domain knowledge, to find the test cases that belong to a certain requirement, as the two artifacts are disconnected.

It becomes clear that this places a major burden on the developers to provide knowledge on the domain knowledge encapsulated in software artifacts. This poses a threat to an organisation, as the developer might leave at a certain point in time.

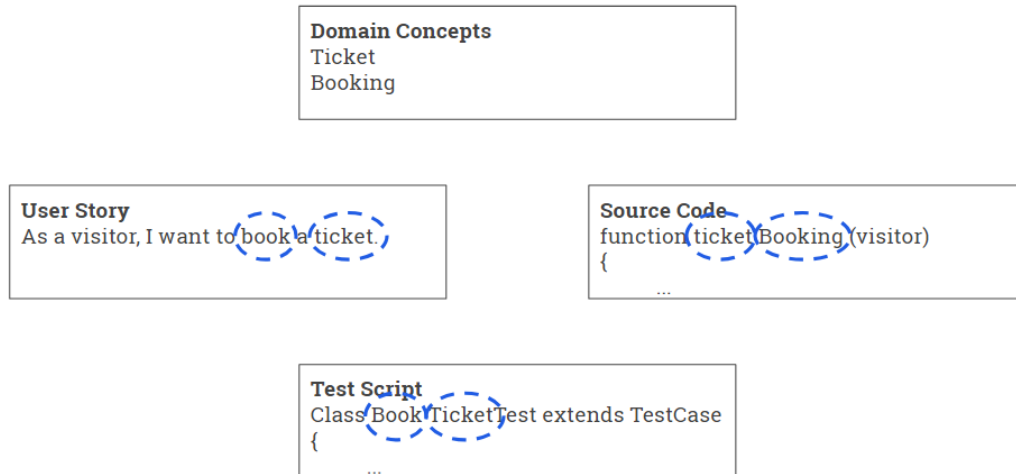


FIGURE 4.1: Domain concepts, and their occurrence in Software Artifacts

Hiring a new developer will be a costly task, as the new developer will need to become acquainted with the software product and its implementation.

In the following section, we present a vision on integrated software development tooling, that has the potential to solve this problem.

4.2 Vision

Each software artifact that is produced in the software development process, contains domain knowledge on a specific part of the software product or its development process. These artifacts might be disconnected, but we can potentially relate them to each other by extracting and mapping domain concepts from them.

An example can be seen in Figure 4.1. Each software artifact that is displayed in this figure contains the domain concepts 'ticket' and 'booking'. In this case, we can use the shared domain concepts as evidence, to state that the artifacts are related. The extraction and relation of domain concepts from software artifacts is a process that can be automated using natural language processing techniques. If we can automatically relate software artifacts in this way, we can solve the problem described in the first section of this chapter: The product manager can use the links between software artifacts, to find the test scripts that belong to a certain requirement.

Next to tracing solutions and software artifact navigation, the relations enable various types of analysis on the software product and its development process. For example, when a new requirement is specified, its impact can be forecasted, by extracting domain concepts from the new requirement, and analysing where they collide with existing software artifacts. This provides the software product manager with valuable information, that can be used as an input to the development process.

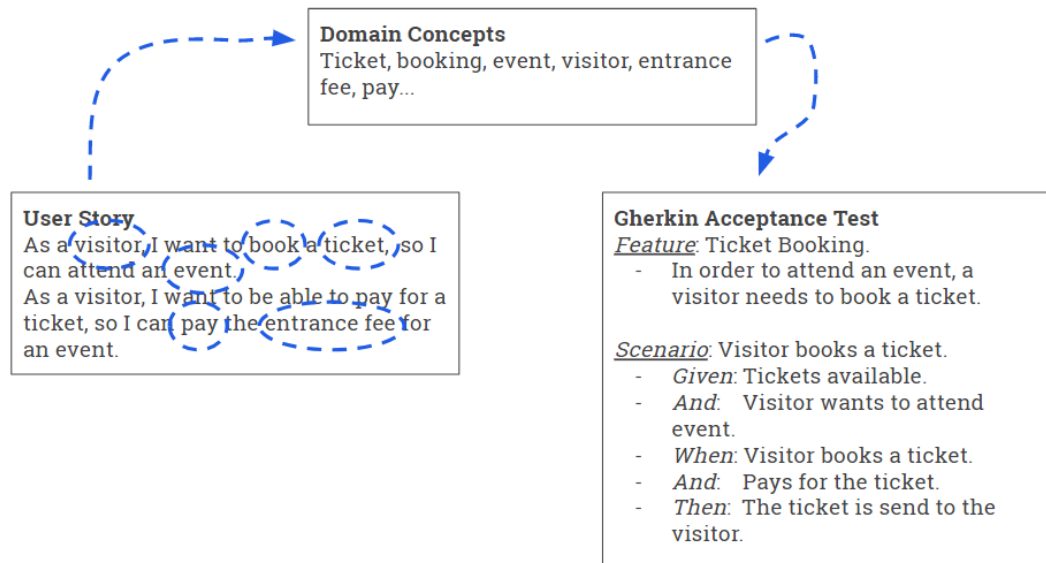


FIGURE 4.2: User Stories to Gherkin

A final usage scenario of the relations is the generation of software artifacts. In modern day software development, a plethora of templates is used for software artifacts. Two clear examples of these templates are the 'User Story' requirement template and the 'Gherkin' acceptance tests template, used in agile software development and behaviour driven development respectively. Because of their precise syntax, we can find linguistic mappings between the two template, allowing for the generation of one artifact from the other. By extracting domain concepts from a User Story, and applying them to our linguistic mapping, we can automatically generate Gherkin acceptance tests. This approach has already been demonstrated by Lucassen et al., and an example can be seen in Figure 4.2. (Lucassen et al., 2017)

Traditionally, trace links are mainly used for navigation purposes, in most cases navigation from a requirement to other artifacts, to track the implementation state of a requirement. (Bouillon, Mäder, and Philippow, 2013) By applying natural language processing and ontology techniques, other uses for software traceability are enabled, as we know more about the semantics of the trace link. This shows the true benefit of the envisioned solution.

As can be seen in Figure 4.2, a lot more domain knowledge is needed for the generation of software artifacts, in comparison with the establishing of simple relations between software artifacts.

4.3 Needs

In order to achieve our vision of linguistic tooling for software development, the following is needed:

- Digital integration of software artifacts.

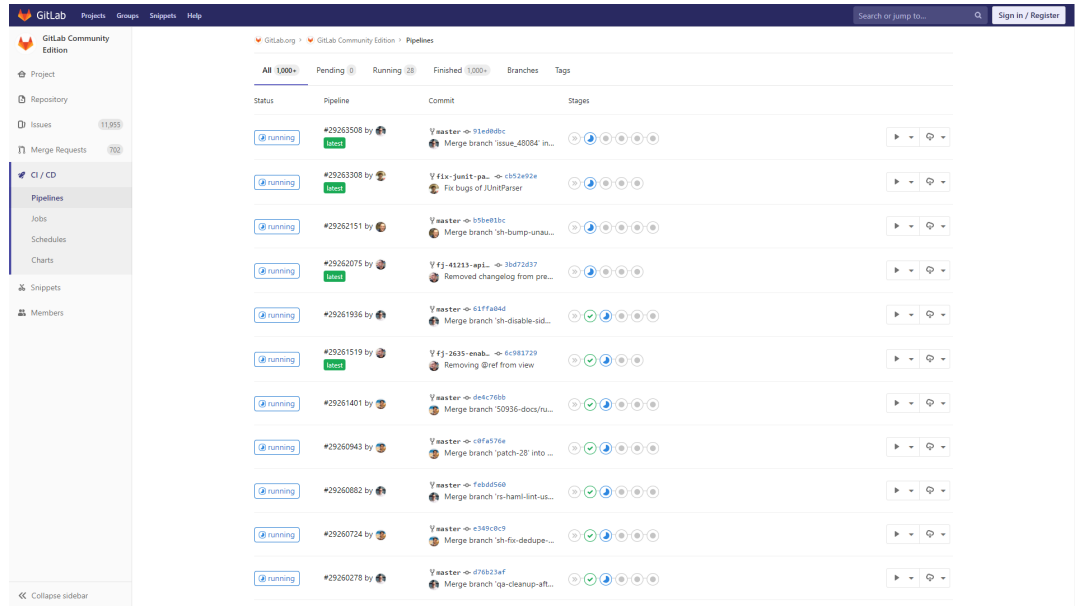


FIGURE 4.3: GitLab

- Methods to extract domain concepts from each type of software artifact.
- A method to relate software artifacts using domain concepts.
- Methods to generate software artifacts using domain concepts.

First of all, the software artifacts need to be digitally integrated, in order to make natural language processing possible. It is very common in current software development that an architecture only exists on the whiteboard and in the brains of the development team, but in order to perform an analysis, it will need to be integrated in the software development toolkit of the development team.

An example of such a toolkit, is the open-source collaborative revision management platform GitLab. GitLab already integrates a plethora of software artifacts, and we focus on integrating our research with GitLab, as will be explained in a later chapter.

Next, we need methods to extract domain concepts from software artifacts, so we can automatically extract the domain knowledge entailed in a certain software artifact. Lucassen et al. provide a great contribution to this, by extracting conceptual models from user stories. (Lucassen et al., 2016a) Even though conceptual models from user stories are a sufficient starting point, we need to be able to learn domain concepts from each software artifact, in order to achieve our vision.

The next step is to create a method to relate software artifacts using domain concepts. This is the main topic and contribution of this master thesis, where we use domain concepts extracted from software artifacts and a automatically generated domain ontology to achieve software traceability between different software artifacts. This is only possible when the software artifacts are fully digitally integrated, and the domain concept extraction is of a sufficient quality.

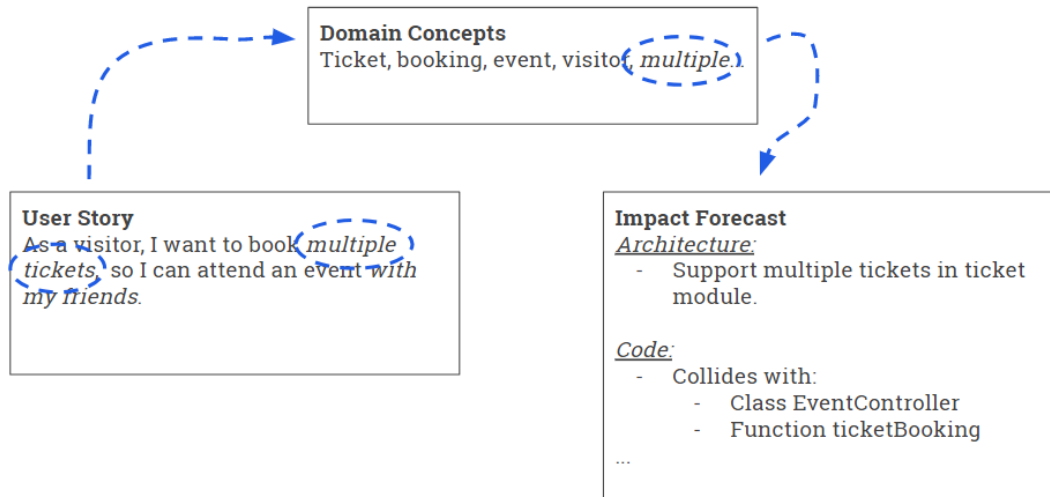


FIGURE 4.4: Automatic Impact Forecasting

Finally, when the extraction of domain knowledge from software artifacts is advanced enough, and when we can establish trace links between a wide variety of software artifacts, we can create methods to generate various software artifacts in an automated manner. A first step to achieving this ambitious goal would be to provide suggestions as a starting point for a software artifact, such as module suggestions in architecture models, or the generation of a basic Gherkin acceptance test, so that the practitioner can create better software artifacts.

Another type of software artifact generation is the generation of software artifacts that are not used that often, such as an impact forecast for a new requirement. Such an artifact is often difficult to create, and its creation process is often time consuming. By using the trace links that have been generated, we can automatically provide practitioners with a brief overview of where the new requirement collides with the existing software product. An example can be seen in Figure 4.4. Another example can be seen in Figure 4.5, where we can see how a new feature relates to the different software artifacts in the software system. In this case, the practitioner can easily navigate to the areas of the software system that are influenced by the new feature.

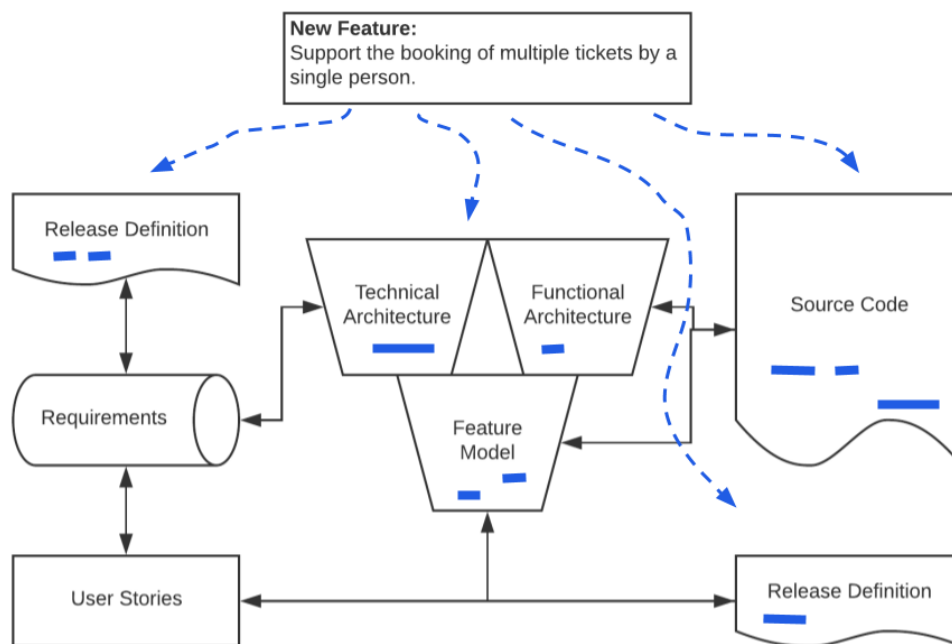


FIGURE 4.5: Impact Forecasting in multiple Artifacts

Chapter 5

Ontological Traceability for Software Theory

In this chapter, a theory for ontological traceability for software is proposed. The first section describes the Product Domain Ontology, which provides the foundation of ontological traceability for software. The second section describes how the Product Domain Ontology can be used to enable software traceability, in the form of a method.

5.1 Product Domain Ontology

We have adopted the definition of Grüber for an ontology at an earlier stage in this research, defining an ontology as an explicit specification of a conceptualization. (Grüber, 1995) In this section, we envision the Product Domain Ontology (PDO), as an explicit specification for a certain software product. Notice that the PDO is a *domain ontology*, meaning that the PDO is different for each software product. We propose the following definition for the Product Domain Ontology:

*"A **Product Domain Ontology** is a **domain ontology**, that explicitly specifies the conceptualization of a certain **software product**."*

It is our vision that the PDO can eventually become the single source of truth for a software product, functioning as a bridge between the different artifacts related to the software product and its development process.

An example PDO can be seen in in Figure 5.1. This example PDO is a lightweight ontology, meaning that it only includes concepts, concept taxonomies and relations. It differs from a heavyweight ontology in the fact that it does not contain axioms, which aide the semantic interpretation of concepts and relations. This PDO was automatically learned from user story requirements, using the visual narrator tool, which is described by Lucassen et al. (Lucassen et al., 2016a)

In this research, we only address a domain ontology on the software product. In future research, the potential use of a top-level *software product ontology* and a *product task ontology* should be investigated. The rationale behind these other software product ontologies, is that some domain knowledge might be better suited to a top-level

or task ontology. An example of such domain knowledge is the method or function execution sequence of a piece of source code, which would be hard to model in a domain ontology.

5.1.1 Software Artifacts: Building Blocks for the PDO

The development of a software product often entails the creation of a plethora of software artifacts. Each software artifact that is created in the process teaches us something about the software product: It poses a unique view on (a part of) the software product. We can categorize software artifacts into the following four different categories:

- **Specification Artifacts**

Artifacts that describe the software product and how it should be build, such as *requirements*, *architecture* and *documentation*. These software artifacts are the richest source of information for the PDO, as they specify how the software product provides solutions for the domain it functions in. These artifacts are therefore the most important artifacts to consider, when in the process of learning the PDO.

- **Implementation Artifacts**

Artifacts that contain the actual implementation of the software product, such as *code*, *installers*, and *CI/CD tools*. These artifacts are often quite difficult for a linguistic analysis, as they are written in a specific (programming) language, that is meant to be interpreted by a computer, rather than a human. Still, some linguistic elements reside in these artifacts, meaning that a form of linguistic traceability is still possible.

- **Validation Artifacts** Artifacts that validate the workings of the software product, such as *test scripts*. Validation artifacts check whether the implementation of the software product covers the specification of the software products. These artifacts usually do not add knowledge to the PDO, and therefore learning on these artifacts is of lesser importance then the learning of specification artifacts. In the rare case that these artifacts add knowledge to the PDO, then this is likely a result of an incomplete specification of the software product.

Notice that in some cases, such as in test-driven development, the validation artifacts are actually specification artifacts. The two are closely related, however, even in examples such as test-driven development, there are validation artifacts that are not specification artifacts, such as deployment tests.

- **Auxiliary Artifacts** Other software artifacts such as a *user manual* or a *marketing brochure*, that are designed to aide a stakeholder with a specific task. These artifacts often pose a different perspective on the system, as they are often aimed at a different, more specific type of stakeholder than the other artifacts. Therefore, these artifacts have great potential for enriching the PDO, with more domain knowledge.

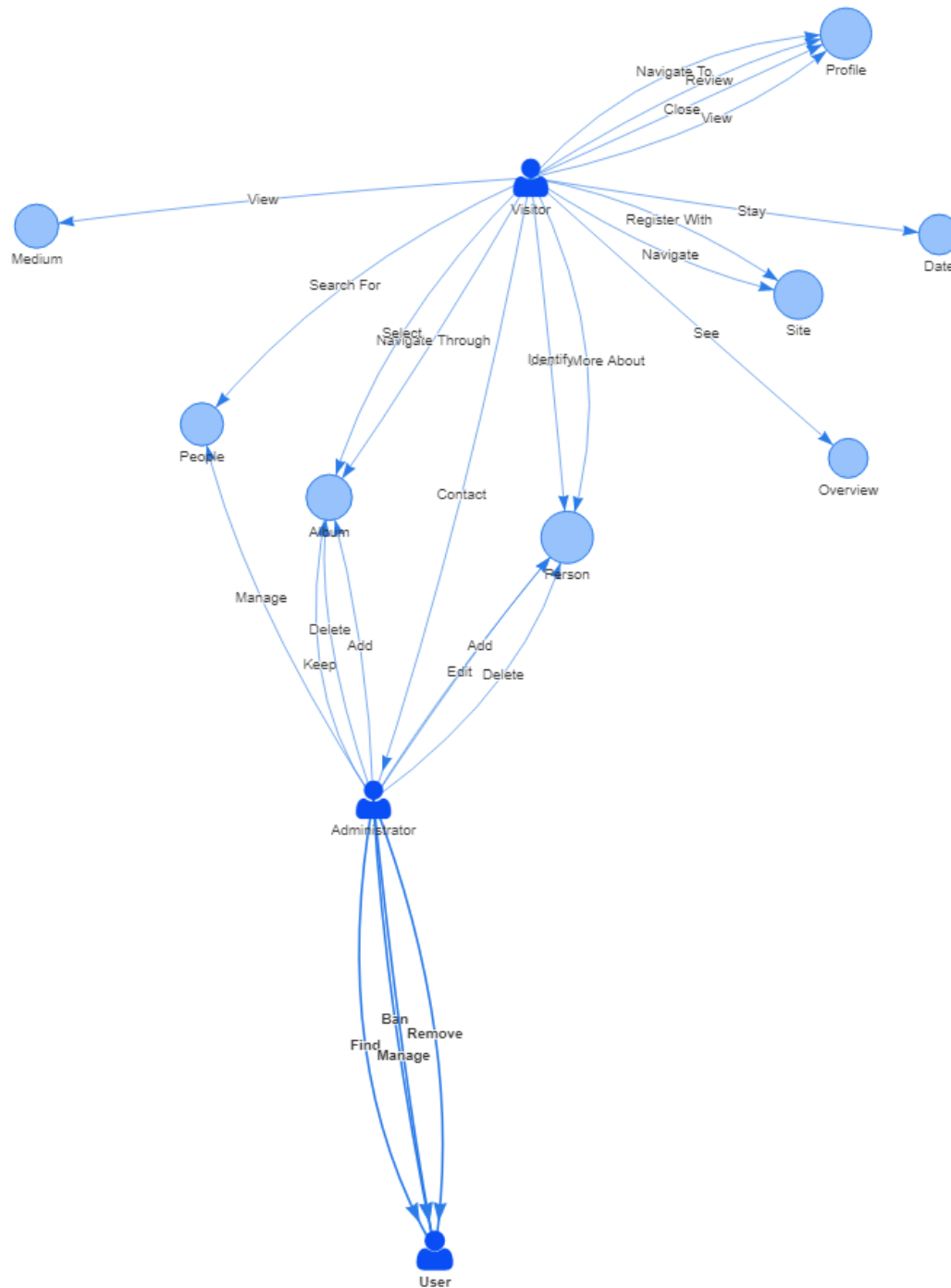


FIGURE 5.1: Product Domain Ontology Example

The PDO is in essence a specification artifact by itself, as it describes the software product and how interacts with its domain. However, it resides as a higher abstraction level than many other specification artifacts, as it contains the combined domain knowledge of all the related software artifacts. This is one of the assumptions we make in this research:

Assumption 1 *A complete Product Domain Ontology can be learned from a complete set of software artifacts.*

This assumption is necessary as we hope that ontological traceability for software will eventually be able to achieve the goal of ubiquitous software traceability. In order to achieve this, it is vital that a complete PDO can be generated, as long as the software artifacts are complete. When this is not the case, ubiquitous traceability can not be guaranteed.

5.1.2 Formalizing Software Artifacts

In this section, a couple of definitions for software artifacts are proposed. First of all, we define A as the complete set of software artifacts of a software system, in a specific domain:

Definition 1

A_S : The complete set of software artifacts of a software system S , in a specific domain.

$a \in A$: Software artifact instance a is a single instance of a software artifact.

Example, the complete set of software artifacts in a software system:

$$A = \{a_1, a_2, \dots, a_n\}$$

Each software artifact instance has a type. Some examples of software artifacts types are a *User Story*, a *PHP Class*, or a *Feature Test*. First, we define T as the complete set of software artifact types:

Definition 2

T : The complete set of types of software artifacts.

P : The complete set of linguistic phrases, associated with a certain software artifact.

Example, the complete set of software artifact types:

$$T = \{\text{User Story, PHP Class, } \dots, \text{Feature Tests}\}$$

Finally, we define the following mapping, that displays the fact that each software artifact instance has a type:

Definition 3

$$Type : A \rightarrow T$$

Example, the type of a software artifact instance:

$a_1 = \text{"As an administrator, I want to create events, so I can manage events in the system."}$

$$Type(a_1) = \text{User Story}$$

In this research, we are interested in the domain knowledge that is entailed in the different software artifacts. In order to extract domain knowledge from software artifacts, we initially extract linguistic phrases from the software artifact. A phrase is defined as a group of words, often carrying a special idiomatic meaning, that functions as a single unit within a grammatical hierarchy.¹ We are interested in the phrases that contain domain knowledge about the software product. We define a mapping that maps the software artifacts A to the set of linguistic phrases P , and provide an example with a user story software artifact. The semi-structured form of user stories aides the extraction process. The process is more difficult from artifacts such as code, which contains less linguistic information.

Definition 4

$$Phrases : A \rightarrow P$$

Example, the phrases resulting from a single user story:

$a_1 = \text{"As an administrator, I want to create events, so I can add events to the system."}$

$$Phrases(a_1) = \{\text{"administrator"}, \text{"create events"}, \dots, \text{"add events to the system"}\}$$

5.1.3 Creating the PDO

The PDO that is used in this research, is a domain ontology learned from user story requirements. This process has been proposed by Lucassen et al., who display a method for the creation of conceptual models from user story requirements. (Lucassen et al., 2016a) These conceptual models can be seen as lightweight ontologies, as they contain concepts, taxonomies and relations, but no axiomas. We aim

¹<https://en.wikipedia.org/wiki/Phrase>

to demonstrate automated software traceability, using the ontologies created from user story requirements as a basis. We assume that the resulting lightweight ontology is sufficient for the creation of a proof-of-concept for ontological traceability for software.

Assumption 2 *A Product Domain Ontology that has been learned from user story requirements is sufficient for a proof-of-concept for ontological traceability for software.*

This assumption entails that while a Product Domain Ontology learned from user story requirements is probably not rich enough to provide results that can be used by practitioners, it can provide results that allow us to evaluate the theory we propose. This is a result of the fact that we can see what the proof-of-concept does, in the cases that the domain knowledge is fully available in the Product Domain Ontology. If the proof-of-concept fails to create a certain trace link, that we would expect, we can see if this is the result of missing domain knowledge from the Product Domain Ontology.

Most ontologies describe roles, concepts, attributes and relations. In this formalization, we only address concepts and relations, as roles and attributes are useful for artifacts specified in natural language, but not so much for artifacts that are not specified in natural language (such as code).

We start of by defining C and R as the sets of concepts and relations respectively:

Definition 5

C : The set of concepts of the domain of the software system.

R : The set of relations of the domain of the software system.

Example, concepts and relations in a software system:

$$C = \{\text{Administrator, Event, } \dots, \text{System}\}$$

$$R = \{\text{create, add, } \dots, \text{delete}\}$$

Using the concepts and relations, we can define the PDO of the software system S as the subset graph, resulting from the application of the relations (R) to the Cartesian product of the concepts (C), with labels:

Definition 6

$$PDO_S : \langle C, R \rangle, \text{ where } R \subseteq C \times C, \text{ with labels}$$

The creation of the PDO is achieved through a process called ontology learning, which has been elaborated on in chapter 3. We define ontology learning as the process of extracting concepts and relations from linguistic phrases, that in turn have been extracted from software artifacts.

Definition 7

$$OntologyLearning : P \rightarrow C \times R$$

Using this definition, we can also define the PDO as the result of applying ontology learning to a set of software artifacts:

Definition 8

$$OntologyLearning(A_S) = PDO_S$$

For the process of ontology learning, we use a procedure described by Lucassen et al. (Lucassen et al., 2016a) The identification of concepts and relations from user stories in the work of Lucassen et al. is achieved by applying 23 heuristic rules, in order to find patterns in a given language. These patterns can then be used to extract a lightweight ontology from the user stories.

As the definition of ontology learning suggests, ontology learning needs to be performed from all available software artifacts, in order to achieve a complete PDO.

The usage of a lightweight ontology generated from user stories is a good starting point, but the ontology learning process will need to be extended in the future, by incorporating other software artifacts in the learning process. User stories only provide a single perspective on the software product, and they often abstract from implementation details. The addition of learning from other software artifacts, such as *architecture* or *test scripts* can therefore enrich the PDO, as they are different types of software artifacts, and provide other perspectives to the software product.

5.2 PDO Traceability Method

In this section, we propose a method for the creation of trace links between various software artifacts, using the PDO. The theory behind this method is elaborated in the first section, after which, each part of the method is explained in depth.

The method consists of the following 5 steps: Artifact instance selection, linguistic term extraction, sub-ontology creation, ontology matching, and trace link creation. The basic rationale of the method is that we compare software artifact instances, and use the PDO to find evidence that two artifact instances are related. When we find enough evidence, the method creates a trace link between the artifact

instances. Each step will be elaborated on in the following sections. The complete method can be seen in Figure 5.2.

5.2.1 Trace Link Creation Theory

We start of by defining the hypothetical complete set of trace links in a software system as L :

Definition 9

L : The set of trace links between software artifacts in a software system.

In order to establish these trace links, we compare the sub-ontologies of the two software artifacts, to find evidence of their relatedness. We define the sub-ontology of a software artifact, as the union of the concepts and relations in the PDO, that are related to the artifact. The sub-ontology is not created through the process of ontology learning, but through the process of mapping the linguistic phrases of the artifact instance to the PDO, in order to find relating concepts and relations. This process is elaborated in the next section on the PDO traceability method.

Definition 10

$SubOntology : \langle C, R \rangle$, where $R \subseteq C \times C$,

and $SubOntology(a) \subseteq PDO$

and $a \in A$

A trace link establishes a relation between two software artifacts. We can therefore define a trace link as a subset of the Cartesian product of all software artifacts. Furthermore, a candidate trace link between two artifacts is only a trace link, if and only if the intersection of its sub-ontologies is greater than a predefined threshold.

Definition 11

$TraceLink \subseteq A \times A$

$SubOntology_1 \cap SubOntology_2 : \{ \langle C, R \rangle | c \in C_1 \wedge c \in C_2,$

$r \in R_1 \wedge r \in R_2 \}$

$(a_1, a_2) \in L \iff |\{SubOntology(a_1) \cap SubOntology(a_2)\}| \geq threshold$

Example, a trace link between two software artifacts: A *User Story* and a *Feature Test*.

$a_1 = \text{"As an administrator, I want to create events, so I can manage events in the system."}$

$a_2 = \text{"CreateEventTest extends TestCase"}$

$\text{TraceLink} = (a_1, a_2)$

In the next sections, the PDO traceability method will be proposed. This method is an implementation of the theory above on the creation of trace links, using the PDO.

5.2.2 PDO Traceability Method Overview

The general rationale of the PDO Traceability Method is to find evidence for the relatedness of software artifact instances, using the PDO. The PDO Traceability Method consists of the following 5 main phases:

- **Artifact Instance Selection**
Two artifact instances are selected as a candidate trace link, these two artifact instances will be compared in the current iteration of the method.
- **Linguistic Term Extraction**
Using linguistic patterns, that are unique for each software artifact type, linguistic terms are extracted from the two artifact instances.
- **Sub-ontology Creation**
Using the linguistic terms, a sub-ontology is created, by adding a concept or relation from the PDO to the sub-ontology, if it resembles one of its linguistic terms.
- **Ontology Matching**
The two sub-ontologies are compared to each other, resulting in a similarity score for the two artifact instances.
- **Trace Link Creation**
If the similarity score for the two artifact instances is over a predefined threshold, the two artifact instances are related, and a bidirectional trace link is created between the two artifact instances.

In the following sections of this chapter, we will elaborate each of the 5 phases in the PDO Traceability Method. A process-deliverable diagram for the method can be seen in Figure 5.2.

5.2.3 Artifact Instance Selection

Whilst we are often talking about traceability between software artifacts, the actual trace links connect the instances of these artifacts. For example, a single user story is an example of an instance of the *requirements* software artifact, and a single function is an example of an instance of the *code* software artifact.

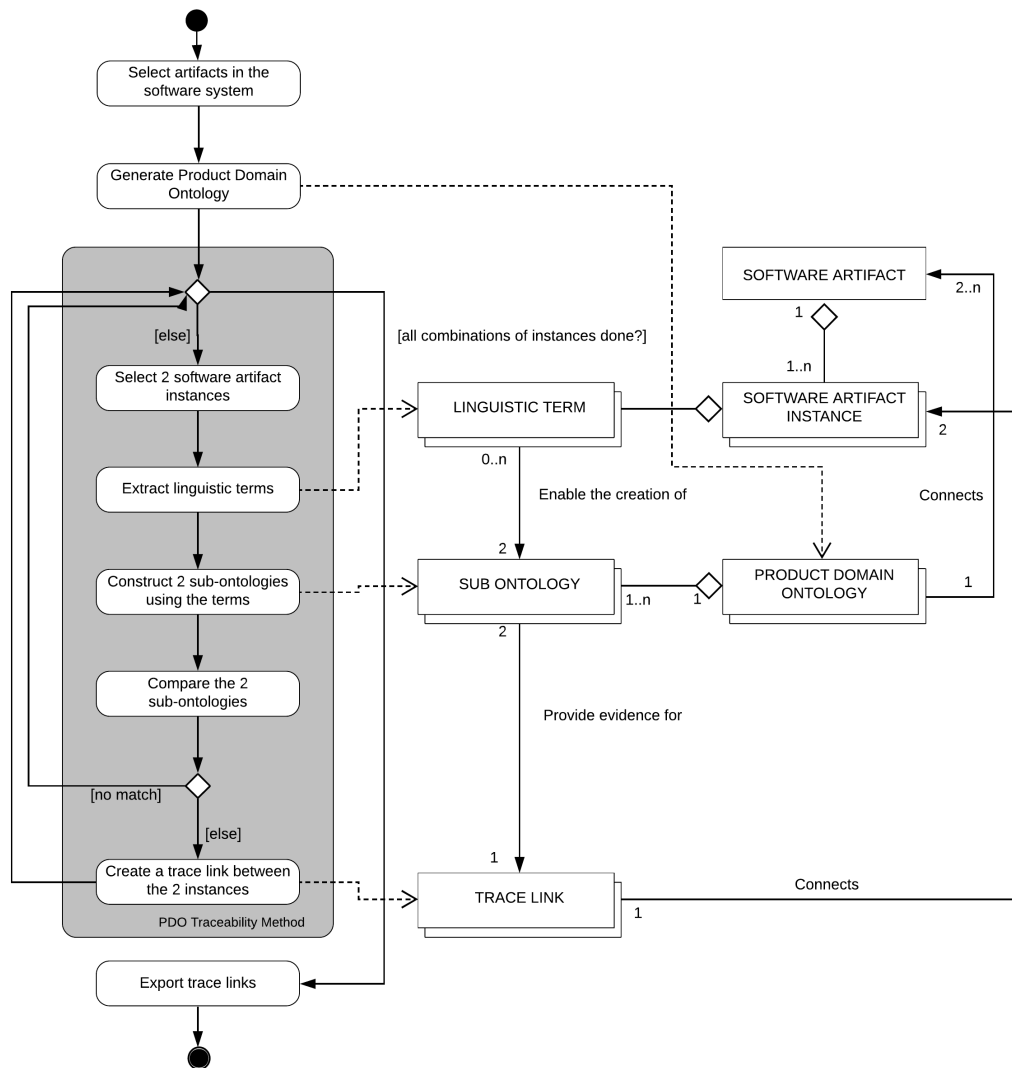


FIGURE 5.2: PDD for the PDO Traceability Method

Before we can even start with traceability, we need to identify which artifact types and instances for which we want to generate trace links. This will vary for per software product, as there are a couple of decisions to be made:

- **Types of Software Artifacts**

The first decision that needs to be made is which types of software artifacts need to be traced in the software product. Not every software product contains the same software artifacts, and the trace links need to be fit-for-purpose. (Cleland-Huang et al., 2014) Furthermore, the practitioner needs to decide which artifacts should be traced in the software product, based on which information is of interest.

- **Software Artifact Instances**

Different software products use different software artifact instances. For example, in this research, we use a software product that uses *user stories* as its requirements specification language. However, this is not the case for each software product. For some artifacts, such as *code*, we could also decide to establish trace links to multiple types of artifact instances. In the case of code, this could mean that we establish trace links to both classes and individual functions (or methods), even though the functions are embodied in the classes.

In the proof-of-concept, the types of software artifacts and their instances have already been selected, for the software product that will be used. The software artifact types are: Requirements (user stories), code (model, view and controller classes with their respective functions) and test scripts (feature tests).

The first step of the algorithm is to select 2 different software artifact instances. All the possible combinations of software artifacts will need to be analyzed. The only exception to this is when the inverse of the candidate trace link has already been analyzed: Because of the bidirectionality of the trace links, if the inverse of a trace link has already been analyzed, the candidate trace link will not have to be analyzed again. A candidate trace link will be analyzed if the following holds:

Definition 12

$$isAnalyzed : A \times A \rightarrow B, \text{ where } B = \{0, 1\}$$

$$isCandidate : A \times A \rightarrow B, \text{ where } B = \{0, 1\}$$

$$isCandidate : \neg isAnalyzed((a_1, a_2)) \wedge \neg isAnalyzed((a_2, a_1))$$

The analyzed function maps a trace link tuple to the boolean domain, resulting in a *true* result if the trace link tuple has already been analyzed. A trace link is a candidate trace link if neither the trace link or its inverse have been analyzed.

These artifact instances will then be compared by the algorithm, to see if there is enough evidence for a trace link. This process is repeated until each possible combination of software artifact instances has been checked by the algorithm.

5.2.4 Linguistic Term Extraction

When 2 artifact instances have been selected, the algorithm extracts linguistic terms from the artifact instance. In this step, we construct 2 distinct lists of terms that appear in the artifact instances. However, due to the lexical differences, a slightly different approach is necessary for each artifact. A pattern needs to be identified for each software artifact type, in order to extract all the meaningful linguistic terms.

Definition 13

LT : The complete set of linguistic terms of a software artifact instance.

$$Terms : A \rightarrow LT$$

Example, linguistic terms in a user story:

$a_1 = \{\text{"As an administrator, I want to create events, so I can add events to the system."}\}$

$$Terms(a_1) = \{\text{administrator, create, events, add, system}\}$$

As can be seen above, in a user story requirement, we can leave out the artifact syntax constructs, such as "As a" and "I want", because they do not provide additional information that can be beneficial to the comparison of the artifact instances. The same holds for a function instance in the source code artifact, where we can leave out the "function" or "class" terms, as it does not add information for the comparison of the artifacts.

It is important to notice that we are not performing an ontology learning step here. In the process of ontology learning, phrases are extracted from the software artifacts, after which heuristics are applied to find meaningful concepts and relations, in order to construct an ontology. The linguistic term extraction step in the PDO Traceability Method, as well as the sub-ontology creation step described in the next section, resemble ontology learning, but differ quite a bit from it. The most important distinction between the two processes is that in the creation of sub-ontologies, no heuristics are used. The linguistic terms that have been extracted from the two artifact instances under analysis are mapped to the existing PDO, in order to construct a sub-ontology, as will be explained in the next section. This process is significantly less complex than the difficult task of ontology learning. This is also the reason we are talking about *linguistic terms* in this step, in contrast to *phrases* in the ontology learning step.

5.2.5 Sub-ontology Creation

After the 2 lists of linguistic terms have been obtained, they will be related to the PDO, with the goal of constructing 2 sub-ontologies, one for each of the artifact instances. For each term, the PDO is analyzed, in order to find a concept or relation that resembles the linguistic term. The word 'resembles' is used, because we need to do more than just compare words. For example, the words *book* and *booking* are different, but could resemble the same relation in the PDO. In this step, stemming is also performed on the linguistic terms. And to further add to the complexity of the procedure, the word *reserving* could also relate to this same relation in the PDO. For the comparison of the linguistic terms, a semantic similarity score is used.

Definition 14

$$\text{Similarity} : lt_i \rightarrow c_i, r_i$$

$$\text{where } lt_i \in LT \text{ and } c_i \in C \text{ and } r_i \in R,$$

$$C, R \in PDO$$

As can be seen in the definition above, we compare the linguistic term under analysis to the concepts and relations in the PDO, to find a resembling concept or relation. If we find a concept that resembles the term, we add this concept to the sub-ontology. If we find a relation that resembles the term, we add the relation, its origin concept and its destination concept to the sub-ontology, as the relation will not make sense without them.

Definition 15

$$\text{inSubOntology} : C, R \rightarrow B, \text{ where } B \in \{0, 1\}$$

We repeat this process for each linguistic term related to the artifact instance, and for both the artifact instances. At the end of this procedure, we have constructed 2 sub-ontologies, that each represent the artifact instance that was used to create the sub-ontology.

Statis vs. Run-time Sub-ontology Creation

The current step of the method, in which the sub-ontologies are created, can potentially be executed beforehand, which saves computing time. When an artifact enters the UTA for the first time, the UTA creates a sub-ontology, and references it to the artifact. When sub-ontologies are statically created, the UTA only has to perform a lookup operation at run-time, rather than creating the complete sub-ontology. As

the creation of the sub-ontology is an expensive operation, this saves a lot of computation time, at the expense of storage space.

However, when creating sub-ontologies beforehand, we need to handle the case in which an artifact changes, or the PDO changes. When this happens, the UTA needs to check the affected sub-ontologies, to make sure they are still correct.

5.2.6 Ontology Matching

We check the similarity of the artifact instances by comparing the constructed sub-ontologies, in a procedure called *ontology matching*. If the similarity of the sub-ontologies of the artifact instances is over a certain threshold, a trace link is constructed, as the artifacts are apparently related to each other. We can state that they are related, because they share a part of the PDO, meaning that they contain matching domain knowledge.

A lot of research has been done already in this field, and we have decided to select the QOM method described by Ehrig et al. as our ontology matching method, as it is one of the few methods that provides a score for the similarity of the ontologies, rather than just a mapping. (Ehrig and Staab, 2004) This method will result in a similarity score for the sub-ontologies. If this similarity score is over a certain threshold, we can proceed to the final step of the method, the creation of a trace link.

5.2.7 Trace Link Creation

When enough evidence for the relatedness of the artifact instances can be found in the ontology matching procedure, a trace link is created between the artifact instances. This trace link is then saved as a unordered 2-tuple containing references to both artifact instances. The tuple is unordered, as the trace link is bidirectional, meaning the the inverse tuple refers to the same trace link.

This marks the end of the analysis of the 2 artifact instances, after which, the method proceeds to the next set of candidate artifact instances. The method terminates when there are no more candidate trace links to analyze. At this point in the method, all identified trace links are exported. In the most straightforward case, the trace links are exported as a list of 2-tuples, containing references to the artifact instances. However, the exporting module can be adapted to suit different platforms, that can use the trace links. To summarize the complete method:

Definition 16

$$\text{GenerateTraceLinks} : A, PDO \rightarrow TL,$$

where A is a set of software artifacts, PDO is the Product Domain Ontology, and TL is the resulting set of trace links.

A complete example of the PDO Traceability Method, from start to finish, can be found in Figure 5.3.

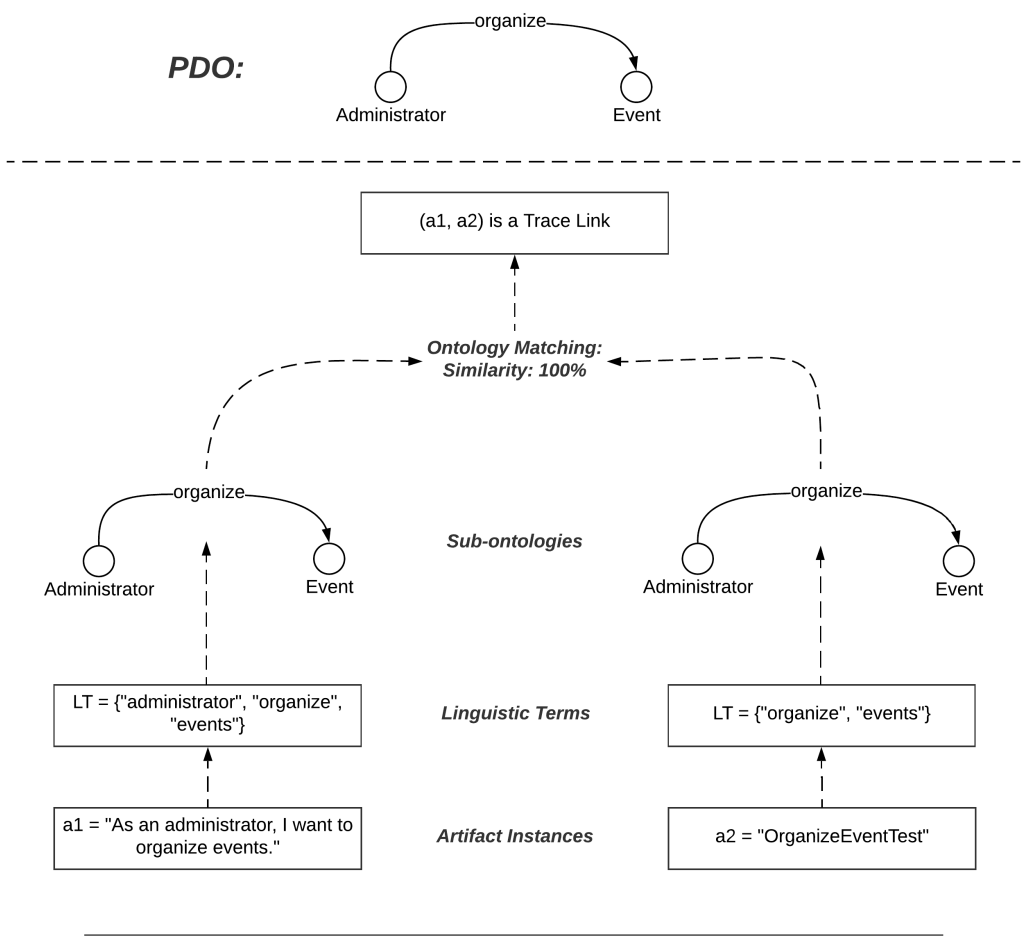


FIGURE 5.3: PDO Traceability Method Example

5.2.8 Implementation

This chapter only covers the formalization of the PDO Traceability Method. The actual implementation of the method in the proof-of-concept can be found in chapter 7, where the algorithms are discussed more in-depth. This is done because the implementation of the algorithms may change over time, while the formalization of the method will remain as documented in this chapter.

Chapter 6

Trace Link Integration in GitLab

One hurdle in the adoption of software traceability in practice, is the fact that scientific prototypes often are not adopted by practitioners. The researchers in the Grimm project group at Utrecht University aim to overcome this difficult problem by embedding our research into software traceability in the open-source collaborative revision management system GitLab.

Due to its open-source nature, we can freely contribute to the platform, and provide many practitioners over the world with the benefits of software traceability. In return, we have the potential to obtain large data sets of trace links, that can be used for analysis. This will help the developments in the field of software traceability, as the lack of gold-standard data sets is a recognized problem in the field. (Antoniol et al., [2017](#))

The Grimm research group at Utrecht University has already had contact with some employees at GitLab, and they are exited about our contributions. We are currently in the process of exploring the possibilities of a close collaboration in our research efforts.

This chapter describes some of the most valuable software traceability features that could potentially be implemented in GitLab. This chapter gives an impression of what could be achieved when successfully implementing software traceability.

6.1 Ontology Based Artifact Navigation

The first and foremost part of the theory that should be implemented in GitLab, is the accessibility of the PDO from the GitLab platform. If a GitLab project makes use of user story requirements in the issue tracker, these user stories can be exported through the GitLab API, and then be analyzed by the series of Grimm tools: AQUA and the Visual Narrator.

AQUA is a framework, combined with a tool, that assesses the quality of user story requirements.(Lucassen et al., [2016b](#)) By implementing its functionality in GitLab, practitioners can create higher quality user stories. The Visual Narrator tool can

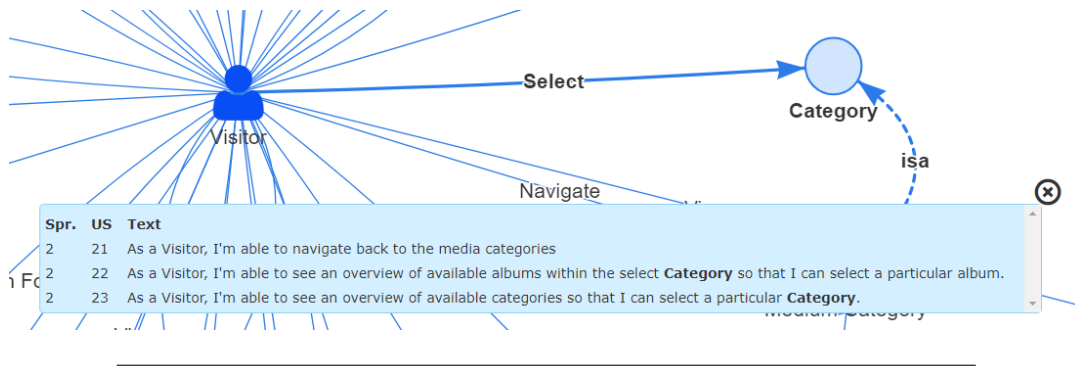


FIGURE 6.1: Ontology Navigation Example

use these high quality user stories, to create an ontology for the software product, using natural language processing techniques. (Lucassen et al., 2016a)

The Visual Narrator will return an ontology to GitLab, which can then be visualized using a visualization package, such as WebVOWL. This allows the practitioners to see the PDO for their software product, and it is the quickest way to provide the benefits of our research to practitioners.

Being able to access the ontology from GitLab, allows stakeholders to have discussions on the software product, using a single model for a reference, instead of many complicated requirement documents. But on top of that, having access to the ontology in GitLab enables *ontology based artifact navigation*. This means that a practitioner can browse through software artifacts, using the ontology as a navigation system, rather than a file tree. For example, if a product manager wants to see the user stories related to a certain concept in the ontology, the product manager only has to click on the concept node in the ontology. And if a developer wants to quickly find where a certain feature is implemented, the developer only has to click on the relation in the ontology, and will be redirected to the correct code file.

The features described above can result in significant time advantages for practitioners, and will make searching for artifacts a lot easier. Furthermore, if this feature is implemented in GitLab, a lot of sets of user stories become available to the Visual Narrator, which provides opportunities for the improvement of the tool, as more data becomes available.

An example of this feature can be seen in Figure 6.1, which has been taken from the Interactive Narrator tool¹, developed in the Grimm Project. The example that is visible in the figure, is when the product manager clicks on the *Category* concept in the PDO, and sees the related user stories.

¹<https://interactivenarrator.science.uu.nl/demo>

6.2 PDO Based Artifact Suggestions

Another application of trace links in software development, is to use the concepts and relations from the PDO, and the trace links, to suggest terms while implementing the software product. To elaborate this feature, an example of code suggestions is used, but suggestions can be given to any type of software artifact. This is more of a function related to the IDE of the developer, but GitLab plays a big part in this as well, as they control the revision of the code and other artifacts. In the example in Figure 6.2, we can see that a software developer is in the process of creating a function. By analyzing the code as the developer types it, we can provide suggestions for the software developer. In this case, we see that the developer is building a function with the *create* term in it, as well as a second term that starts with the letter *E*. If we have the relation (Role) - Create - Event in the PDO, we could suggest the term *Event* to the software developer.



The image shows a snippet of code in a text editor. The code is 'public function createE' followed by a cursor. A suggestion box is open, showing 'createEvent' with a small icon to its left. Below the code, there is a comment '/**'.

FIGURE 6.2: PDO Concept Suggestions

This feature has many potential benefits, which are listed below:

- **Improved Programming Workflow**

First of all, it makes the life of the programmer easier, as they do not have to type the entire functions: They can just accept the suggestions.

- **Artifact Consistency**

Furthermore, it results in more consistent software artifacts, as the code is now in line with the PDO, because the software developers have used the same terms. Because the software artifacts are now more similar in terms of linguistics, this will result in better trace links as well.

- **Improved Developer Focus**

Finally, it actively reminds the software developer of the requirements, while implementing them. This helps to keep the software developer focused on the task at hand.

The functionality described above, using the PDO to suggest terms, could be enhanced even further with the use of trace links from GitLab. For example, if the software developer in the example is assigned to a specific user story in the issue tracker, we can provide only a limited amount of terms for the suggestion, based on the sub-ontology related to the user story. This will result in more accurate results, and faster results, as the search space is smaller.

6.3 User Story Testing

One of the most promising features that is enabled by software traceability, is the combination of information from different software artifacts, that would otherwise be disconnected. A clear example of this, is the combination of requirements (user stories), code, and different types of test cases, in combination with the trace links between the two artifacts. This allows us to see whether a user story is sufficiently tested, what the code quality of the implementation of a certain user story is, if the UI of the user story adheres to the style guide, or if the user story might even be failing in the test cases, and so on.

An example on how to integrate this in to GitLab, is visualized in Figure 6.3. Here, we can see a user story, that resides in the issue tracker of GitLab. Other than the regular view you would get on this page, additional information is shown: Different types of tests are visible, and the green color indicates that the test are passing. In this specific example, 4 different types of tests are visualized:

- **Feature Tests** - *Is the feature described in the user story implemented?*
- **Code Quality** - *Is the code that implements the user story of high quality?*
- **Build Tests** - *Does the user story pass the production build CI/CD tests?*
- **UI Tests** - *Does the UI that enables the user story function as expected?*

This information is very valuable to the product manager, and it is currently unavailable, as the requirements, code, and test cases are disconnected in most software projects. By implementing these features in GitLab, traceability will become increasingly more valued by practitioners.

The tests displayed in the example in Figure 6.3 are only the beginning however, as GitLab continues to develop their testing pipelines, more and more types of software artifacts will become available for trace links.

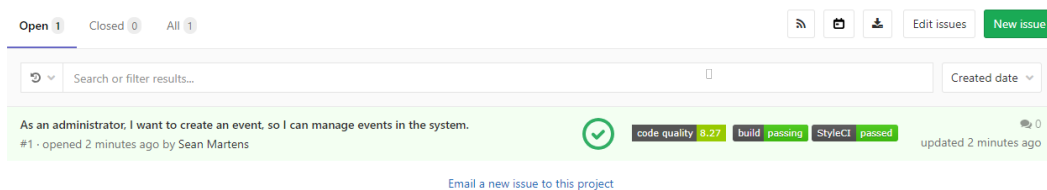


FIGURE 6.3: Test Information for User Stories

6.4 Requirement Impact Analysis

One final feature that we will describe in this chapter, is the requirement impact analysis. In the current situation in software development, when a new requirement arises, its impact can only be estimated by domain experts on the software product, that have a lot of implementation knowledge. This means that the decisions on

whether the requirement can be implemented in time, are therefore done based on opinions, rather than data.

When a new requirement is entered in the system, the UTA will immediately analyse the new requirement, and generate trace links to relating software artifacts. If the new requirement conflicts with other requirements, or current implementations, the impact of the requirement will be very big. Because we analyzed the new requirement on the fly, we can make an assessment of its impact, and give direct feedback to the product manager, even before the issue is submitted.

An example can be seen in Figure 7.9. Here, a new requirement is being added to GitLab. But, as you can see, the button with *possible conflicts* has been added to the top of the issue. When the user clicks on this button, the user can see conflicting requirements, architecture, or any other type of relevant artifact instance. This allows the product manager to respond, and adapt the user story accordingly, or schedule a meeting to discuss the right course of action.

This is the most futuristic and advanced feature described in this chapter, but remember that it is our goal to eventually realize ubiquitous traceability (in 2035), so these type of features will be available in the future.

The screenshot shows the GitLab web interface for creating a new issue. The top navigation bar includes the GitLab logo, 'Projects', 'Groups', 'More', and a search bar. The breadcrumb trail indicates the user is in 'GitLab.org > GitLab Community Edition > Issues'. The main heading is 'New Issue', followed by a yellow 'Possible Conflict' button and the text 'Degree of conflict: 87%'. The form fields include a 'Title' dropdown set to 'Feature proposal' with the value 'Support Multiple Tickets', and a 'Description' field with a 'Write' tab selected. The description text reads: 'User Story: As a visitor, I want to buy multiple tickets, so I can order more tickets for friends.' Below the description, there is a note 'Markdown and quick actions are supported' and an 'Attach a file' button. A checkbox option is present: 'This issue is confidential and should only be visible to team members with at least Reporter access.' At the bottom, there are two buttons: 'Submit issue' (green) and 'Cancel' (white).

FIGURE 6.4: New Requirement Conflict

Chapter 7

Proof-of-concept Implementation

This section describes the proof-of-concept for ontological traceability for software, that has been created to validate the proposed theory. Even though its purpose is only to validate the theory for ontological traceability for software, the tool has been designed to be extendable, so that future research can continue to contribute to it.

The proof-of-concept for ontological traceability for software has been baptized as the *Ubiquitous Traceability Artisan*, or *UTA* for short. The name displays the ultimate goal of the tool: To eventually deliver ubiquitous software traceability. The proof-of-concept described in this research is only a first step towards this ambitious goal, but hopefully, the tool will eventually achieve the goal.

7.1 Technology

UTA is a web-based application, that has been created with the popular, open-source PHP framework Laravel. The Laravel project is hosted on GitHub, under the MIT licence. Laravel overtook its competitors in 2015, and has been the most popular PHP framework ever since.¹

Using a PHP framework like Laravel allows us to easily create a web application, but at the same time, it also allows us to expose the main functionality of the application through an API. The framework follows the popular model-view-controller architectural pattern, and provides useful features for the creation and maintenance of relational databases.

The Laravel framework allows us to quickly build prototypes for the UTA tool, but each feature that is contributed can be exposed through the API, so the functionality can be integrated with other platforms.

For the future of UTA, we aim to bring its features to many practitioners, by integrating them with GitLab. GitLab is a version control system, with advanced features for issue tracking, continuous delivery and continuous integration. GitLab is the market leader in continuous integration tooling², and is rapidly acquiring new customers. One of the key advantages of GitLab is that it is also an open-source project, developed by a large community under the MIT licence.

¹<https://www.sitepoint.com/best-php-framework-2015-sitepoint-survey-results/>

²<https://about.gitlab.com/2017/09/27/gitlab-leader-continuous-integration-forrester-wave/>

GitLab already integrates many different software artifacts, making it an ideal candidate to integrate with. However, GitLab integration and large practitioner experiments are beyond the scope of this research. This research focuses on developing the key features to validate the proposed theory, in the form of a proof-of-concept. These features can be integrated with GitLab at a later point in time.

7.2 Architecture

This section elaborates the architecture of the UTA. The architecture described in this section is the envisioned eventual architecture of the UTA, rather than the exact architecture of the proof-of-concept that is delivered with this thesis.

7.2.1 Functional Architecture

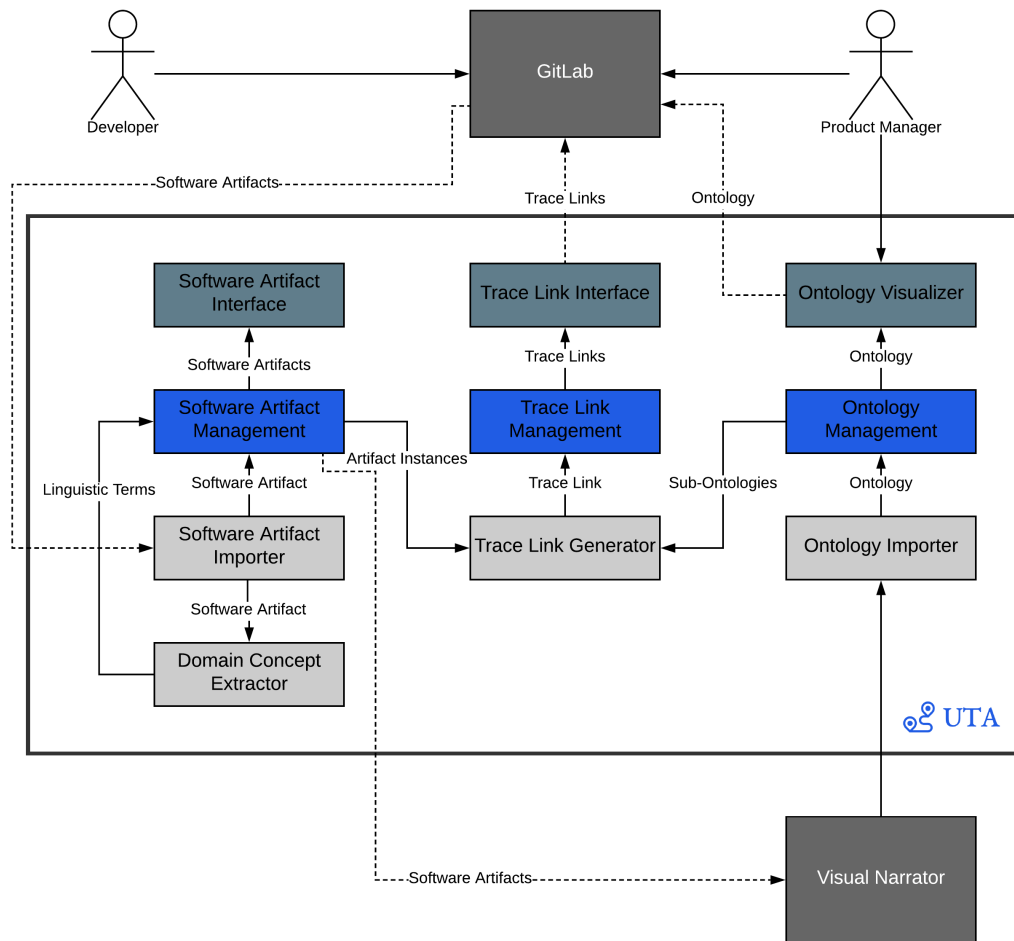


FIGURE 7.1: UTA Functional Architecture Model

The functional architecture model, that can be observed in Figure 7.1, describes the system from a usage perspective. In the center of the architecture, the UTA and

its modules can be identified. Furthermore, two external applications can be seen in the architecture: GitLab and Visual Narrator. These applications provide key functionalities to the UTA, and are therefore relevant in the architecture.

The UTA contains three strategic modules, depicted in blue in the architecture:

- Software Artifact Management
- Trace Link Management
- Ontology Management

The Software Artifact Management module allows the users to be in control of the software artifacts in the system. In order to analyze the software artifacts, the system first needs to know where to find them. The user can add software artifacts in two different ways: By hand or through an API connection. The rationale behind these features is that some software development companies already have all their artifacts contained in a (online) version control system, that allows for an API connection, but some companies still work offline, and the system should support both. By connecting to a version control system like GitLab, we can automatically import a majority of the software artifacts in an automated manner, meaning the system requires little effort from practitioners.

The Software Artifact Management module can be seen in a more in-depth view in Figure 7.2. As can be seen, the 'facade' pattern is applied, as the types of artifacts that need to be parsed vary a lot. This makes sure that whatever artifact enters in the system, the artifact parser can handle it (if a facade is created for it). The imported artifact, as well as its linguistic terms are stored in a database.

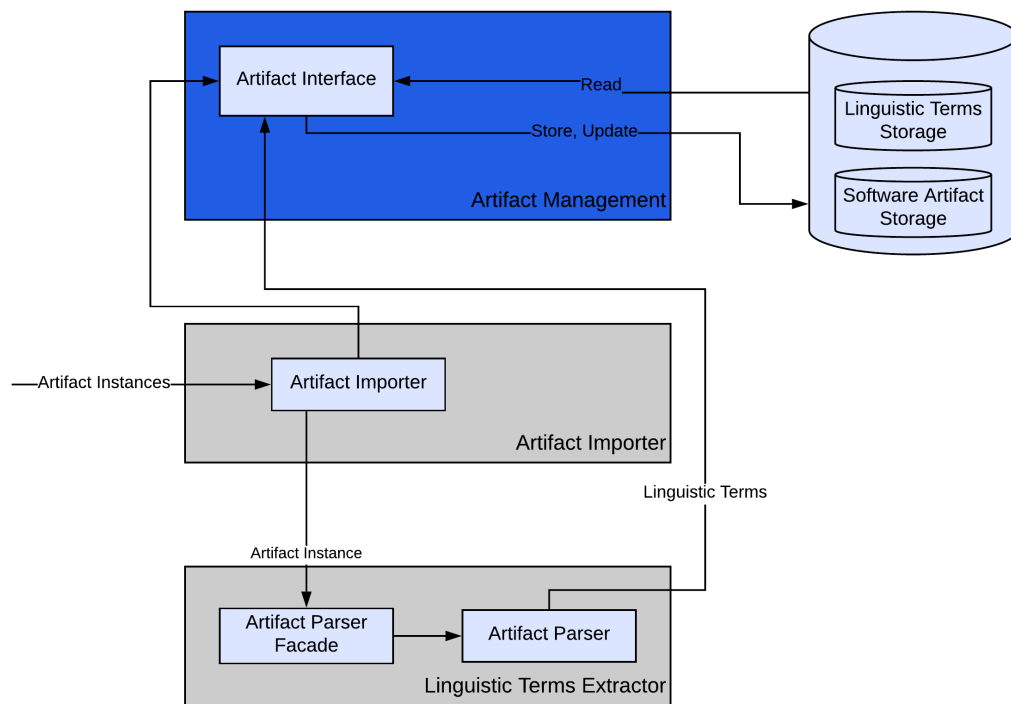


FIGURE 7.2: In depth: Software Artifact Management

The Trace Link Management module handles the storage and maintenance of trace links, that have been created by the system. It allows users to add missing trace links, or delete redundant trace links by hand, in case the system has made an error. Furthermore, this module allows the user to control what happens with the trace links after the generation. The users can get an overview of the trace links in the system, or export them to other systems (like GitLab), to benefit from the advantages of the trace links. How the trace links can be used in systems like GitLab to benefit practitioners is described in a later chapter.

The Trace Link Management module and its surrounding modules can be seen more in-depth in Figure 7.3. The Trace Link Generator receives two sub-ontologies for a set of two software artifacts, and creates a trace link, if the ontologies are similar. This trace link is stored in the Trace Link Storage, and can be updated by users through the Trace Link Interface, which also handles the exporting of trace links, to other applications.

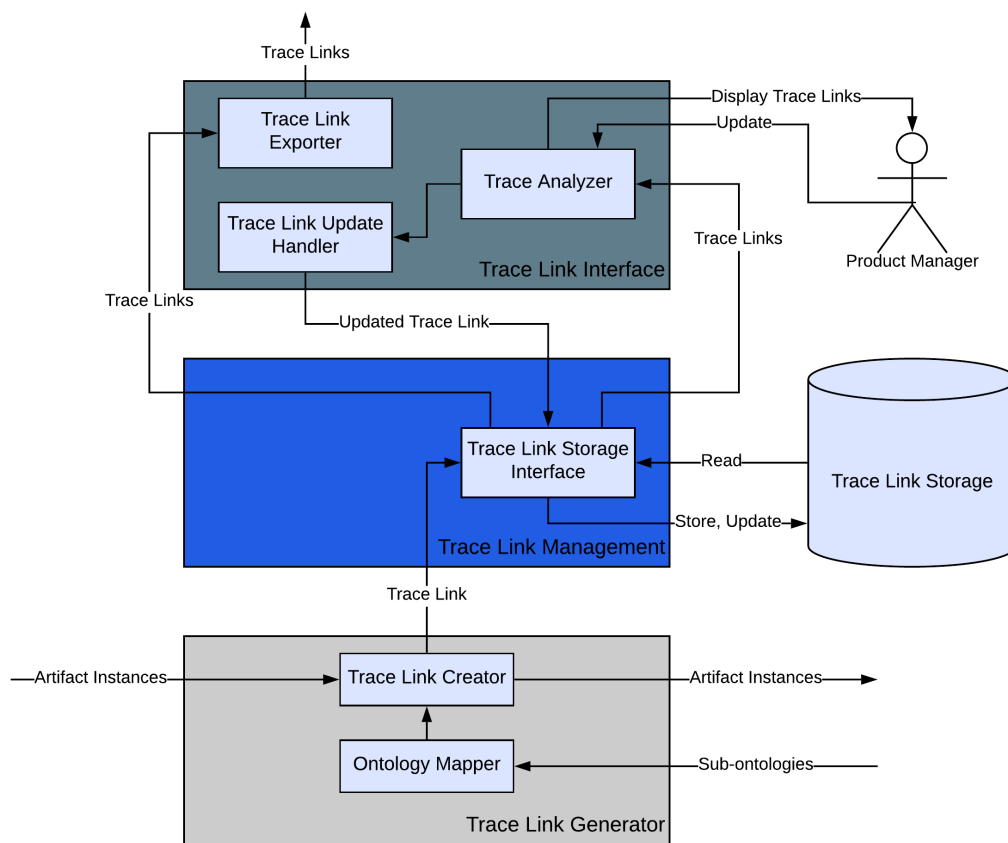


FIGURE 7.3: In depth: Trace Link Management

The third strategic module is that Ontology Management module. This module stores and updates the PDO, that has been generated by the Visual Narrator tool. The module allows the user to view the ontology, and enrich the ontology, in case the automated generation made an error.

The Ontology Management Module can be seen more in-depth in Figure 7.4. The ontology is received from the Visual Narrator tool, and parsed and stored in the Ontology Storage. The ontology can be enriched by users through the ontology visualizer. The Ontology Management module is also responsible for the generation of sub-ontologies from artifact instances. It takes a artifact instance and its related linguistic terms, and returns a sub-ontology to the requesting module.

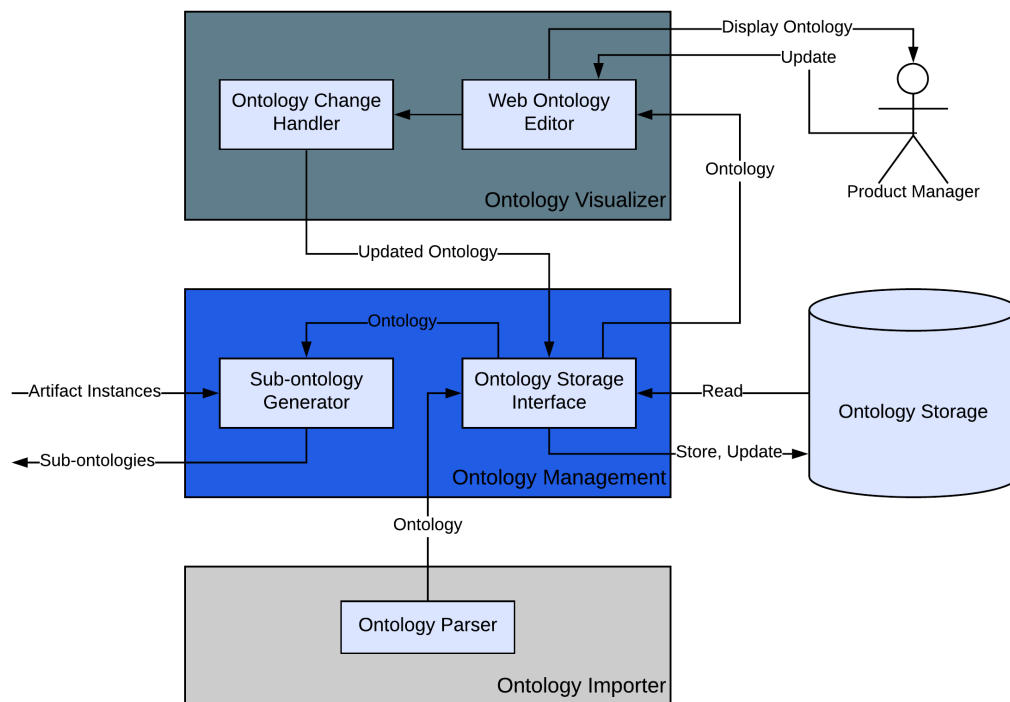


FIGURE 7.4: In depth: Ontology Management

7.3 Linguistic Term Extraction

Before the system can start comparing software artifacts, in order to find trace links, the linguistic terms need to be extracted from the artifact instances. These linguistic terms contain the domain knowledge entailed in the software artifact. This section describes the linguistic term extraction procedure in the UTA, that analyses an artifact instance, extracts the linguistic terms, and saves both the artifact instance and the associated linguistic terms in the database.

Unfortunately, a plethora of software artifacts is available, and they differ a lot in form, which makes linguistic term extraction a complicated task. The linguistic term extraction procedure is aided by a set of templates: Each type of software artifact has its own template, which is used for the extraction procedure. The diagram in Figure 7.5 depicts the architecture of the linguistic term extraction module in the UTA.

Once the artifact instance has been imported, the module extracts its type, and sends the type to the template selector. The template selector finds the appropriate template for the artifact type, and sends it to the parser. The artifact instance will be parsed using the template provided, after which the linguistic terms are exported to the requesting module.

The remainder of this section will display the templates used for the artifacts that can be used in the proof-of-concept for the UTA.

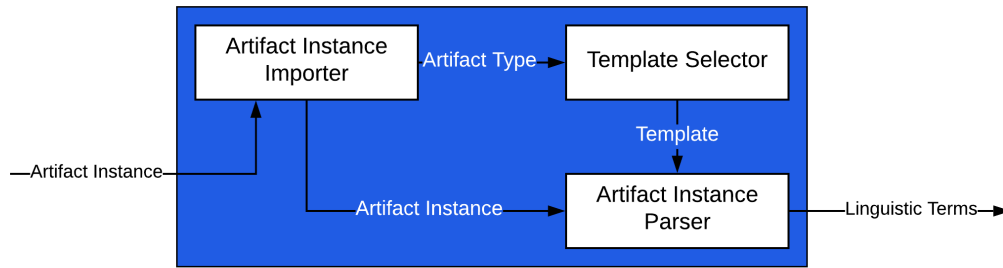


FIGURE 7.5: Linguistic Term Extraction Procedure

7.3.1 User Story

As the ontology used in the proof-of-concept is an ontology generated from user story requirements, it is only natural that the proof-of-concept accepts user stories. The UTA accepts user stories that follow the Connextra template, which is used by 70% of the practitioners. (Lucassen et al., 2016a)

Connextra template: "As a (**type of user**), I want (**goal**), [so that (**some reason**)]."

The semi-structured nature of user stories aides the extraction procedure. The UTA follows the following template when parsing user stories:

Template 1

*"As a ({**linguistic terms**}), I want ({**linguistic terms**}), [so that ({**linguistic terms**})]."*

After extracting the linguistic terms, stop word removal is performed, to filter out words such as 'a', 'for', and 'the'.

7.3.2 Code

In contrast to user stories, code is a more complex software artifact to define templates for. The biggest problem with this software artifact, is the fact that it is not written in natural language, which makes it hard to find and extract linguistic terms.

For the extraction of linguistic terms from code in the proof-of-concept, we accept three types of the code software artifact: Models, views, and controllers. This decision was made, because the *model-view-controller(MVC)* pattern is widely used in practice, and was also used in the data set that was available for the proof-of-concept.

For the proof-of-concept, we extract linguistic terms from the headers of the model and controller code classes, as well as from the headers of the functions that

reside in these classes. For the extraction of linguistic terms from the views, we make use of the translation files used in the view software artifact.

Model

The model is the central point of the pattern, that manages the data, logic, and rules of the application.³ The linguistic terms are extracted from model classes according to the following template, extracing from both the class header and the function headers:

Template 2

class {linguistic terms} extends Model

{public, protected, private} function {linguistic terms} ({linguistic terms})

View

The views in the MVC pattern can be any output that represents the information in the application. Due to its varying nature, and due to the large amount of markup necessary to create views, linguistic term extraction is a complicated task for views. However, the views do have one property in common: They often use translation files to represent content in the views. By extracting the translation identifiers from a view class, we can lookup the linguistic terms in the translation files. The UTA does this according to the following template:

Template 3

__('translation_identifier')

'translation_identifier' => '{linguistic terms}'

Controller

The controller component in the MVC pattern accepts input from the user, and converts it to commands for the model or the views.⁴ The linguistic terms are extracted from controller classes according to the following template, extracing both from the class header and the function headers:

³<https://web.archive.org/web/20120729161926/http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>

⁴<https://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design>

Template 4

```
class {linguistic terms}Controller extends Controller

{public, protected, private} function {linguistic terms} ({linguistic terms})
```

7.3.3 Test Case

Test cases could be seen as a variation of the code software artifact, as it is still a script file. However, test cases do follow their own template, which highly resembles other code classes. The template is depicted below. This template hold for both *unit tests* and *feature tests*.

Template 5

```
class {linguistic terms}Test extends TestCase

{public, protected, private} function {linguistic terms} ({linguistic terms})
```

7.3.4 Linguistic Term Splitting

Often, a post-processing step is necessary after extracting the linguistic terms from the software artifacts. Many linguistic terms, especially those coming from code classes, are still grouped together in a single term, while these are in fact separate linguistic terms. An example of this is CamelCase, which is a convention that is often applied in code. In CamelCase, spaces are removed, and the letter after the removed space becomes a capital letter. The linguistic term splitting convention used in the proof-of-concept can be seen in the example below:

$$\{\text{CamelCase}\} \rightarrow \{\text{Camel}, \text{Case}\}$$

Furthermore, in test cases, it is often the convention to split the individual words describing the test case, with an underscore, rather than using CamelCase. Therefore, we use the following convention for splitting linguistic terms in test cases:

$$\{\text{test_case_description}\} \rightarrow \{\text{test}, \text{case}, \text{description}\}$$
7.3.5 Linguistic Term Extraction Pseudocode

The following algorithm specifies the linguistic term extraction procedure, as implemented in the UTA.

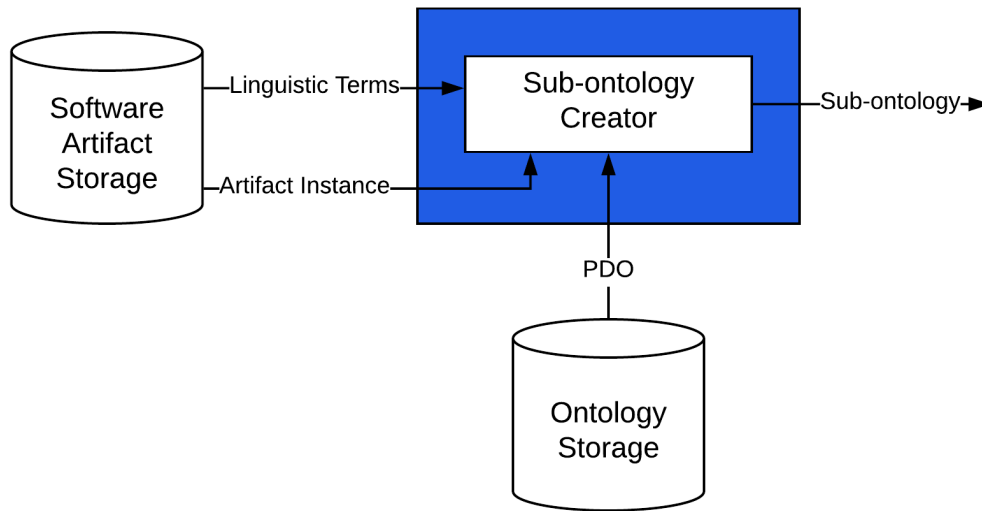


FIGURE 7.6: Sub-ontology Creation Architecture

Algorithm 1: Linguistic Term Extraction**Data:** The artifact instance and its type**Result:** A set of linguistic terms related to the artifact

- 1 Define the empty set of linguistic terms: lt ;
- 2 Request the template using the artifact type;
- 3 Apply template to the artifact instance to receive the $rawTerms$;
- 4 **for all** $rawTerms$ **do**
- 5 split the raw term if needed;
- 6 add the term(s) to lt ;
- 7 Filter lt for keywords;
- 8 **return** lt

7.4 Sub-ontology Creation

At this point in the process that is being described, the UTA has a set of software artifact instances, and their corresponding linguistic terms. The linguistic terms associated with a software artifact instance will be used to generate a sub-ontology for the artifact instance, which is necessary for the generation of trace links. The procedure to go from linguistic terms to a sub-ontology is described in this section. The section of the functional architecture of the tool that deals with the sub-ontology creation is depicted in Figure 7.6.

The most logic method to create a sub-ontology for a specific software artifact instance, would be to apply ontology learning techniques to the artifact instance.

However, this would mean that an ontology learning method would have to be created for each software artifact that we would like to trace, which is a lot of work. Therefore, a different approach is used, using the linguistic terms that have been generated for the artifact instances.

The method uses the PDO as a reference ontology, and selects which parts of the ontology are related to the artifact instance, using the linguistic terms. Therefore, if a concept or relation does not appear in the PDO, then it will not be in the sub-ontology. The sub-ontology is therefore a subset of the PDO.

Definition 17

$$SubOntology : A \rightarrow Ontology$$

$$SubOntology(a_i) \subseteq PDO, \text{ where } a_i \in A$$

7.4.1 Sub-ontology Creation Procedure

The following algorithm specifies the sub-ontology creation procedure, as implemented in the UTA.

Algorithm 2: Sub-ontology Creation

Data: A set of linguistic terms for the artifact instance, and the PDO

Result: A sub-ontology that represents the artifact instance

```

1 Define the empty sub-ontology: so;
2 for all concepts in the PDO do
3   for all linguistic terms do
4     if match found then
5       add current concept to so;
6     continue with next concept;
7 for all relations in the PDO do
8   for all linguistic terms do
9     if match found then
10      if domain concept in so and range concept in so then
11        add current relation to so;
12      continue with next relation;
13 return so

```

The algorithm starts by checking all concepts in the PDO, to see if there is a linguistic term associated with the artifact instance, that resembles the concept. The matching procedure is done using the string similarity method proposed by Oliver.

(Oliver, 1993) If a match is found, the concept under analysis is added to the sub-ontology.

When all concepts are checked, the algorithm analyses the relations in the PDO. Once again, the algorithm tries to match the relation to a linguistic term related to the artifact instance. However, if there is a match, the relation is only added to the sub-ontology if both the domain concept and the range concept of the relation are already in the sub-ontology. Otherwise, the relation will not be added to the sub-ontology.

After all concepts and relations have been checked, the algorithm returns the sub-ontology that has been created.

7.5 Trace Link Generation

The final step of the method is to generate the trace links, using the generated sub-ontologies. The following algorithm describes how the UTA generates a trace link.

Algorithm 3: Trace Link Generation

Data: Two sub-ontologies, for the two artifact instances under analysis

Result: A score that can be used to determine whether a trace link exists between the artifact instances

```

1 Define the amount of matching concepts: matchingConcepts;
2 for all concepts in sub-ontology1 do
3   for all concepts in sub-ontology2 do
4     if match found then
5       increment matchingConcepts;
6 Define the amount of matching relations: matchingRelations;
7 for all relations in sub-ontology1 do
8   for all relations in sub-ontology2 do
9     if match found then
10      increment matchingRelations;
11 Calculate similarity1: (matchingConcepts + matchingRelations) /
    totalElements(sub-ontology1);
12 Calculate similarity2: (matchingConcepts + matchingRelations) /
    totalElements(sub-ontology2);
13 return max (similarity1, similarity2);
```

The algorithm analyses the two sub-ontologies, and counts the amount of concepts and relations that they have in common. A similarity score is then calculated, by dividing the amount of matching elements in the sub-ontologies by the total number of elements, in both sub-ontologies. The largest result is returned at the end of the algorithm.

The reason for the double similarity score, is because the size of the sub-ontologies could be very different. For example, the first sub-ontology could belong to a large artifact instance, where the second sub-ontology could belong to a small artifact instance. The first similarity score would yield a relatively low similarity score in this case, because of the division by a greater number. In contrast, the second similarity score could be very large, because of the division by a smaller number.

The similarity score can display the existence of a trace link from the first to the second sub-ontology, and the other way around. We return the largest of the two similarity scores, because if that similarity score is high enough to establish the trace link, the inverse trace link is also established, because of the bidirectionality of a trace link.

7.6 Connecting the Modules

In the previous sections, several modules of the UTA have been described. We now have everything in place to implement the PDO Traceability Method in the UTA. The pseudocode for the overall implementation of the PDO Traceability Method can be seen in the following code section. As can be seen, the implementation in the UTA aligns with the theory that has been discussed in chapter 5.

Algorithm 4: PDO Traceability Method

Data: A set of software artifact instances, a PDO, and a trace link threshold

Result: A set of trace links between the artifact instances provided

```

1 Define the empty set of trace links: traceLinks;
2 for all artifact instances as source do
3   for all artifact instances as target do
4     Extract the linguistic terms from the source and the target artifact
       (Algorithm 1);
5     Create sub-ontologies for the source and target artifacts, using the
       linguistic terms (Algorithm 2);
6     Receive a similarityScore for the artifact instances, using the
       sub-ontologies (Algorithm 3);
7     if similarityScore > threshold then
8       add trace link (source, target) to traceLinks;
9 Store the traceLinks in the trace link database;
```

The algorithm makes use of the three previously defined algorithms, and executes them for each possible combination of trace links. If the received similarity score is over the threshold, a trace link is created.

After the completion of the algorithm, the trace links that have been created are stored in the trace link database, adhering to the architecture.

Notice that in the current implementation, described in the algorithm above, we are doing double work. If the potential trace link between artifact instances a_1 and a_2 is already analyzed by the algorithm, the inverse trace link (a_1, a_2) will not have to be analyzed, due to the bidirectional property of the trace links. This can easily be fixed by creating a list of artifact instances, and removing an artifact instance if it has been completely checked as source to all other artifact instances. This way, if an artifact instance is completely explored, it will no longer occur as a target for other artifact instances. This has been implemented in the UTA, but for the sake of simplicity, it has been eliminated from the pseudocode above.

7.7 UTA Screenshots

This section contains a series of screen shots from the UTA application.

Trace Links

ID	Origin Type	Target Type	Origin Description	Target Description
1	User Story	Feature Test	As an administrator, I want to create events, so that I can manage events in the system.	CreateEventTest
2	User Story	Feature Test	As an administrator, I want to update events, so that I can change its information.	UpdateEventTest
3	User Story	Feature Test	As an administrator, I want to update events, so that I can change its information.	UpdateUserTest
4	User Story	Feature Test	As an administrator, I want to delete events, so that I can cancel events.	DeleteEventTest
5	User Story	Feature Test	As an administrator, I want to create a group, so that I can manage groups.	CreateGroupTest
6	User Story	Feature Test	As an administrator, I want to update a group, so that I can update who is in the group.	UpdateGroupTest
7	User Story	Feature Test	As an administrator, I want to update a group, so that I can update who is in the group.	UpdateUserTest
8	User Story	Feature Test	As an administrator, I want to delete a group, so that I can remove redundant groups.	DeleteGroupTest
9	User Story	Feature Test	As an administrator, I want to change data of members, so that I can update their data in case of a change.	UpdateUserTest
10	User Story	Feature Test	As an administrator, I want to set the welcome text after registration, so that I can update the confirmation email.	UpdateUserTest
11	User Story	Feature Test	As an administrator, I want to set a date range on the calendar, so that I can see events over time.	UpdateUserTest
12	User Story	Feature Test	As an administrator, I want to set a date range on a report, so that I can generate a report on a period of time.	UpdateUserTest
13	User Story	Model	As a member, I want to receive emails from my trainer, so I'm informed of upcoming events.	EventModel
14	User Story	Model	As an administrator, I want to create events, so that I can manage events in the system.	EventModel

FIGURE 7.7: Trace Links Identified in the UTA

Classes


















ID	Name	View
1	EventModel	
2	GroupModel	
3	UserModel	
4	EventController	
5	GroupController	
6	ReportController	
7	UserController	
8	CreateEventTest	
9	CreateGroupTest	
10	CreateUserApiTest	
11	CreateUserTest	
12	DeleteEventTest	
13	DeleteGroupTest	
14	DeleteUserTest	
15	UpdateEventTest	
16	UpdateGroupTest	
17	UpdateUserTest	

FIGURE 7.8: Artifact Overview (Classes) in the UTA

Method #1: isRepeatableEvent

Belongs to class: EventModel

Domain Concepts

ID	Name
1	Event
2	is
3	Repeatable
4	Event

FIGURE 7.9: Artifact Domain Concepts (Method) in the UTA

Chapter 8

Validation: Experimental Results

In the previous chapters, the theory for Ontological Traceability for Software has been proposed, and how it has been implemented in the UTA. In this section, we present the results from the tool, by comparing the generated trace links to a data set, that has been constructed with the help of practitioners.

8.1 UTA Evaluation: Experiment Overview

In order to evaluate the UTA, an experiment has been set up, that allows us to draw conclusions on the performance of the proposed vision, and the automated support of the vision in the tool. As sub-question SQ6 suggests, we want to be able to say something about the performance of the UTA in comparison with humans. Therefore, this experiment will be a comparison between trace links, generated by domain experts and the UTA, on a single data set of software artifacts.

The experiment was conducted in cooperation with the Bytestack company. Bytestack has provided us with a data set, containing a fully complete software product, and a plethora of software artifacts. Bytestack uses user stories to formulate their requirements, which can be used by the UTA to generate an ontology for the software product under analysis.

The experiment is a split analysis by domain experts and the UTA, where both create a set of trace links for the software system under analysis. These trace links are then compared, to analyse the performance of the UTA. For this experiment, only the user stories and feature tests will be analyzed, as the analysis of potential trace links requires the analysis of all possible combinations of artifact instances. This is possible for the UTA, but less feasible for the domain experts at Bytestack. Traceability between user stories and feature tests is considered the most valuable type of trace link at Bytestack, as the two are disconnected by nature, while a trace link between them can provide many insights. Therefore, user stories and feature tests were chosen as the artifacts under analysis in this first evaluation of the UTA. An overview of the experiment process can be seen in Figure 8.1.

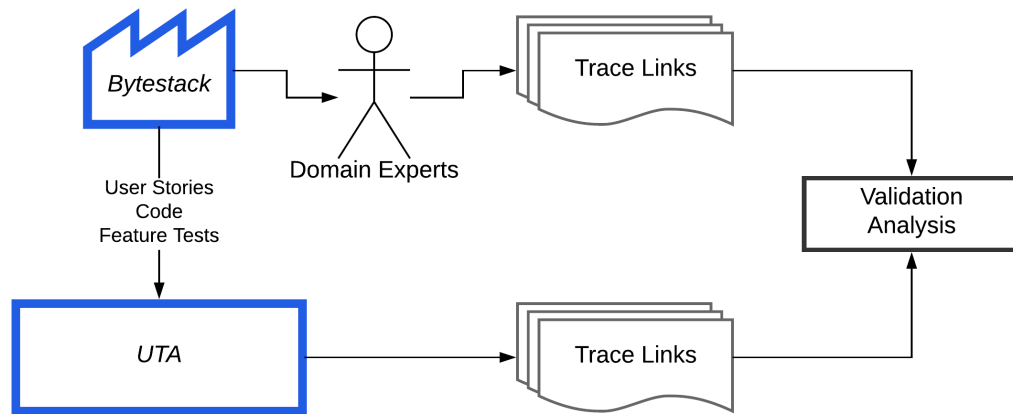


FIGURE 8.1: Experiment Setup

8.1.1 Experiment: Second Iteration

Initially, the experiment was focused on the generation of trace links between user story requirements and feature tests, as the acquisition of manual trace links generated by practitioners is a time-consuming task. However, after the initial iteration of the experiment, it was decided that results on different types of trace links could significantly enhance the quality of the results. Therefore, a second iteration of the experiment has been conducted, where we examine trace links between user stories and code. To be more specific, we are analyzing code files of the *model* and the *controller* type.

This allows us to be more specific on the precision and recall of the proof-of-concept, and it allows us to research whether a trace link between two artifact instances exists, if they both have a trace link to a third artifact instance.

8.1.2 Data Set: The UAS Software Product

The software product that the trace links are generated for, is the User Administration System (UAS), that has been provided by Bytestack. The UAS allows an association to keep track of its members, the events they participate in, and how many dues they have to pay. It is mostly used by local sporting associations. The application is a web-based application, built using PHP and JavaScript. Bytestack offers a variety of software products, but the UAS is chosen because of its relatively small size. Because it is a relatively small software product, the amount of artifact instances is also relatively small. This allows for a more complete examination of the software product, because the manual creation of a data set by domain experts becomes increasingly unfeasible as the amount of software artifacts increases. The following software artifacts are available for the UAS software product:

- **Requirements** - User Stories (44)
- **Code** - Models (3) and Controllers(4)

- **Test Cases - Feature Tests (10)**

In this initial evaluation of the UTA, only the user stories and the feature tests will be analyzed, as this already requires significant efforts from the domain experts at Bytestack.

Next to the limited amount of artifact instances, the user stories for this software product are the most complete set available at Bytestack. Because of the limited amount of features, there are user stories for each implemented feature in the software product. This allows us to create a complete PDO on the software product, as the Visual Narrator tool has a complete oversight of all requirements.

8.1.3 PDO for the UAS

The first step that the UTA undertakes in the process of generating trace links, is the creation of a PDO. For this process, the UTA makes use of the Visual Narrator tool, described by Lucassen et al. (Lucassen et al., 2016a) The PDO that was generated by loading the user stories of the UAS system into the UTA, can be seen in Figure 8.2.

8.2 UTA Evaluation: Trace Links

In this section, the results of the trace link experiment are presented, giving us data on the performance of the UTA, in comparison with expert practitioners.

8.2.1 Trace Links Identified by Domain Experts

The UAS is developed by two main developers at Bytestack. They are completely responsible for the entire system, and can be considered domain experts on the software product, and its workings within an association. These two domain experts created a set of trace links between the user stories and feature tests of the software product. These trace links can be observed in Table 8.1. Notice that the user stories are shortened in the table, in order to fit the table in this document. The complete user stories can be found in Appendix A.

The domain experts were provided with a handout of the user stories of the UAS, and a list of feature tests. For each feature test, the domain experts filled in one or more user stories, that they would establish a trace link with. The reason for this inverse assignment is that the amount of user stories exceeds the amount of feature tests, meaning that there are a number of user stories that are not tested. By asking the domain experts to identify the user stories related to the feature tests, the same trace links are established, and it saves the domain experts time, as the redundant user stories are not analyzed. In the second iteration of the experiment, a list of the models and controllers was provided, in the same fashion as the feature tests in the first iteration of the experiment.

After the domain experts established the trace links individually, a meeting was scheduled to discuss the differences between the established trace links. In this meeting, the domain experts came up with a general set of 10 trace links, that they are completely correct according to them. These are the strongest trace links in the software product, and will be compared with the trace links generated by the UTA.

In the second iteration of the experiment, the same practitioners created trace links between user stories, models, and controllers in the same data set, which resulted in another set of trace links. These trace links can be seen in Appendix D, as the set is too large to fit in the current chapter.

User Story	Feature Test
#4: As an admin, I want to create events.	"CreateEventTest"
#7: As an admin, I want to delete events.	"DeleteEventTest"
#6: As an admin, I want to update events.	"UpdateEventTest"
#12: As an admin, I want to create a group.	"CreateGroupTest"
#16: As an admin, I want to delete a group.	"DeleteGroupTest"
#15: As an admin, I want to update a group.	"UpdateGroupTest"
#26: As an admin, I want to assign costs to the first hour.	"UpdateSettingsTest"
#27: As an admin, I want to assign costs to extra hours.	"UpdateSettingsTest"
#29: As an admin, I want to set the welcome text.	"UpdateSettingsTest"
#1: As a member, I want to register in the system.	"CreateUserTest"

TABLE 8.1: Trace Links Identified by Domain Experts

8.2.2 Trace Links Identified by UTA

The same set of user stories and feature tests are analyzed by the UTA. The UTA uses the PDO that can be seen in Figure 8.2. The trace links that the UTA generated can be seen in Table E.3. Once again, the user stories have been shortened. The complete set of user stories can be seen in Appendix A. For the generation procedure, a threshold of 0.9 has been used, for the similarity score of the sub-ontologies related to the artifact instances.

User Story	Feature Test
#4: As an admin, I want to create events.	"CreateEventTest"
#6: As an admin, I want to update events.	"UpdateUserTest"
#6: As an admin, I want to update events.	"UpdateEventTest"
#7: As an admin, I want to delete events.	"DeleteEventTest"
#12: As an admin, I want to create a group.	"CreateGroupTest"
#15: As an admin, I want to update a group.	"UpdateGroupTest"
#15: As an admin, I want to update a group.	"UpdateUserTest"
#16: As an admin, I want to delete a group.	"DeleteGroupTest"
#18: As an admin, I want to change data of members.	"UpdateUserTest"
#29: As an admin, I want to set the welcome text.	"UpdateUserTest"
#32: As an admin, I want to set a range on the calendar.	"UpdateUserTest"
#39: As an admin, I want to set a range on a report.	"UpdateUserTest"

TABLE 8.2: Trace Links Identified by the UTA

8.2.3 Trace Link Comparison: User Stories to Feature Tests

Now that we have obtained the two sets of trace links, we can compare them. In the creation procedure of the trace links, there are a total of 440 candidate trace links, as there are 44 user stories, and 10 feature tests. The domain experts at Bytestack created a total of 10 trace links, and the UTA created a total of 12 trace links. The following 6 trace links have been identified by both the domain experts at Bytestack and the UTA:

User Story	Feature Test
#4: As an admin, I want to create events.	"CreateEventTest"
#6: As an admin, I want to update events.	"UpdateEventTest"
#7: As an admin, I want to delete events.	"DeleteEventTest"
#12: As an admin, I want to create a group.	"CreateGroupTest"
#15: As an admin, I want to update a group.	"UpdateGroupTest"
#16: As an admin, I want to delete a group.	"DeleteGroupTest"

TABLE 8.3: Common Trace Links: Feature Tests

Next to the 6 true positives, the UTA resulted in 6 false positives and 4 false negatives. On a complete set of 440 possible trace links, this results in an accuracy of 0.977. Unfortunately, the data set is very skewed, and in this case, accuracy is not a representative measurement. The precision and recall provide a better measurement. The precision of the UTA on this data set is only 0.5, while the recall is 0.6. The accompanying confusion matrix can be seen in Table 8.4, and the measurements can be seen in Table 8.5.

The numbers in the tables above do not look promising for the performance of the UTA, but there are logical explanations as to why the current version of the UTA

		Predicted (UTA)	
		0	1
Actual (Experts)	0	424	6
	1	4	6

TABLE 8.4: UTA - Trace Link Confusion Matrix: Feature Tests

Measurement	Value
<i>Precision</i>	0.5
<i>Recall</i>	0.6
<i>F₁ score</i>	0.55

TABLE 8.5: UTA - Trace Link Measurements: Feature Tests

yields these results. We elaborate on this in the discussion section, however, the main reason for the false trace links, is the missing of domain knowledge from the PDO. This is a result of the fact that the PDO is created from a single source (user stories), which do not provide enough domain knowledge.

8.2.4 Trace Link Comparison: User Stories to Models

In the second iteration of the experiment, one of the types of trace links we analyzed are the trace links from user stories to models. The full results can be seen in Appendices D and E.

The common trace links, identified by both the practitioners and the UTA, can be seen in the table below. The table is accompanied by the confusion matrix and the measurements for trace links from user stories to models.

User Story	Model
#3: As a member, I want to receive e-mails from my trainer.	"EventModel"
#4: As an admin, I want to create events.	"EventModel"
#5: As an admin, I want to read events.	"EventModel"
#6: As an admin, I want to update events.	"EventModel"
#7: As an admin, I want to delete events.	"EventModel"
#9: As an admin, I want to add members to an event.	"EventModel"
#10: As an admin, I want to select if an event is billable.	"EventModel"
#11: As an admin, I want to set extra costs to events.	"EventModel"
#12: As an admin, I want create a group.	"GroupModel"
#13: As an admin, I want to add users to a group.	"GroupModel"
#14: As an admin, I want to read a group.	"GroupModel"
#15: As an admin, I want to update a group.	"GroupModel"
#16: As an admin, I want to delete a group.	"GroupModel"

TABLE 8.6: Common Trace Links: Models

		Predicted (UTA)	
		0	1
Actual (Experts)	0	105	13
	1	1	13

TABLE 8.7: UTA - Trace Link Confusion Matrix: Models

Measurement	Value
<i>Precision</i>	0.5
<i>Recall</i>	0.93
<i>F₁ score</i>	0.65

TABLE 8.8: UTA - Trace Link Measurements: Models

8.2.5 Trace Link Comparison: User Stories to Controllers

The final experiment that was conducted, was the experiment for trace links from user stories to controllers. The common trace links that have been identified by both the practitioners can be seen in the table below. The confusion matrix and measurements can be seen below the table.

User Story	Feature Test
#3: As an member, I want to receive e-mails from my trainer.	"EventController"
#4: As an admin, I want to create events.	"EventController"
#5: As an admin, I want to read events.	"EventController"
#6: As an admin, I want to update events.	"EventController"
#7: As an admin, I want to delete events.	"EventController"
#9: As an admin, I want to add members to events.	"EventController"
#10: As an admin, I want to select if an event is billable.	"EventController"
#11: As an admin, I want to set extra costs to an event.	"EventController"
#17: As an admin, I want to set the type of an event.	"EventController"
#12: As an admin, I want to create a group.	"GroupController"
#13: As an admin, I want to add users to a group.	"GroupController"
#14: As an admin, I want to read a group.	"GroupController"
#15: As an admin, I want to update a group.	"GroupController"
#16: As an admin, I want to delete a group.	"GroupController"

TABLE 8.9: Common Trace Links: Controllers

		Predicted (UTA)	
		0	1
Actual (Experts)	0	147	13
	1	2	14

TABLE 8.10: UTA - Trace Link Confusion Matrix: Controllers

Measurement	Value
<i>Precision</i>	0.52
<i>Recall</i>	0.88
<i>F₁ score</i>	0.65

TABLE 8.11: UTA - Trace Link Measurements: Controller

8.2.6 UTA Performance: Combined Results

Three different types of trace links have been analyzed in this research, all trace links originate in the user story. The trace links have three different target software artifacts: Feature tests, models, and controllers. All three artifact types have been submitted to the same experiment, where trace links created by practitioners are compared to trace links created by the UTA. The combined results of the three experiments can be seen in the confusion matrix and measurements table below.

The combined results are calculated using the micro-average of the results of the three individual trace links types.

		Predicted (UTA)	
		0	1
Actual (Experts)	0	676	32
	1	7	33

TABLE 8.12: UTA - Trace Link Confusion Matrix: Combined

Measurement	Value
<i>Precision</i>	0.51
<i>Recall</i>	0.83
<i>F₁ score</i>	0.63

TABLE 8.13: UTA - Trace Link Measurements: Combined

8.3 UTA Performance Discussion

The proof-of-concept for Ontological Traceability for Software yielded a lot of false positives and false negatives, as can be seen in the results overview in Table 8.14. In this section, we will take a closer look at some examples of these false positives and false negatives, to gain a better understanding in the decision procedure of the UTA.

The first two section will use trace links between user stories and feature tests as general point of reference for the discussion, where the final section will discuss the complete performance of the UTA.

	1: US to FT		2: US to M		3: US to C		Combined	
Horizontal: Actual Vertical: Predicted	0	1	0	1	0	1	0	1
0	424	6	105	13	147	13	676	32
1	4	6	1	13	2	14	7	33
Precision	0.5		0.5		0.52		0.51	
Recall	0.6		0.93		0.88		0.83	
F-Score	0.55		0.65		0.65		0.63	

TABLE 8.14: Results Overview

8.3.1 Missing Domain Knowledge

One obvious problem in the trace link generation process of the UTA, is missing domain knowledge. One clear example of this is the trace links related to the application settings: The domain experts created trace links between user stories #26, #27, #29, and the feature test "UpdateSettingsTest". (See Tables 8.1 and E.3) Apparently, assigning costs and setting the welcome text, are all handled in the application settings. However, in the user stories, the word *settings* never appears. Because the word *settings* isn't present in the user story, the sub-ontology generation module doesn't receive the term, and it cannot add this word to the sub-ontology. And even if the linguistic term *setting* was present, it doesn't appear in the PDO (see Figure 8.2), meaning that it will never appear in the sub-ontology.

In the example above, it becomes clear that domain information is missing from the PDO and the linguistic term extraction module. The domain experts do have access to the knowledge that *costs* and the *welcome text* are apparently application settings. This results from the fact that all the domain knowledge is extracted from the user stories, which do not contain this information. In order to improve the UTA, the PDO should be learned from multiple software artifacts, in order to become more complete. On top of that, in the linguistic term extraction procedure, we must look further than just the terms in the artifact itself. Maybe we can combine the artifact terms with terms from another source, that could for example tell us that *costs* and the *welcome message* are in fact *settings*.

The problem described above can also be solved by adding a user story, which contains the domain knowledge, that costs and a welcome message are settings. Therefore, extensively guiding the practitioners in the creation of a complete set of user stories can also be a solution to the problem of missing domain knowledge. Lucassen et al. already provide a basis for user story quality with the AQUASA tool, which can be extended in the future, to check all the quality metrics of the QUS framework. (Lucassen et al., 2016b)

8.3.2 Over-optimistic Sub-ontology Generation

The sub-ontology generation module is currently over-optimistic, leading to small edge case exceptions. The input that the sub-ontology generation module receives

are the plain linguistic terms related to the artifact instance, without any additional context. This results in some problems in certain cases. For example, if the sub-ontology generation receives the linguistic terms *administrator*, *update*, and *Event*, it will add the relation *administrator - update - event* to the sub-ontology, even though the linguistic term *update* might relate to a completely different concept. This is a result of the fact that sub-ontology generation is done through linguistic term mapping, rather than ontology learning.

Creating sub-ontologies using ontology learning is not a feasible solution, as it results in a lot of artifact specific work. Ontology mapping also becomes significantly harder in this case, as concepts and relations might not be exactly the same in terms, even though they represent the same domain knowledge.

8.3.3 General Remarks

The UTA performs very good in recognizing trace links between artifact instances that represent parts of the *CRUD* (*create, read, update, delete*) features. This is because the artifact instances relating to these kinds of features are often very structured, and use a smaller, common vocabulary. The UTA works well with these types of artifact instances, because the vocabulary often resembles the terms in the PDO.

In contrast, when the artifact instances become less structured, the UTA performs less good. More domain knowledge is needed to analyze complex artifacts, and the vocabulary might not resemble the terms in the PDO.

One of the most important findings is that the performance of the UTA is generally the same over the three different investigated types of trace links. This indicates that the performance is not influenced by the type of artifact, but is stable over the investigated artifact types. However, we need to be cautious with this statement, as only three different types of trace links are currently evaluated. Further research and evaluations of the UTA are necessary to confirm these findings.

Furthermore, it is interesting that the vast majority of the errors produced by the UTA, are false positives. This is a positive point, as the UTA apparently does not create trace links that make no sense at all, but is rather optimistic sometimes when establishing trace links. This is a result of the limited amount of domain knowledge available in the artifacts: If two artifacts reference one or two of the same concepts, they are usually related according to the tool, because the intersection of the two sub-ontologies usually approaches 1 in the case of artifact instances with little domain knowledge. This means that even with the high threshold of 0.9 currently used by the UTA, the trace link is still created. While the two artifacts might discuss related topics, this relatedness is not strong enough for a trace links. The only way to prevent these errors in the future, is by extracting more domain knowledge from software artifacts.

Finally, the results are also dependent on the orthogonality of the identified entities in the PDO. In this case, the PDO did not have a lot of similar concepts or

relations. But it is likely, that when a lot of similar terms appear in the PDO, the UTA will provide less favourable results.

Chapter 9

Discussion

In this thesis, we performed a theory building research project on the use of techniques from the field of ontology, for the means of software traceability. Due to the large amount of manual work that goes into the establishment and maintenance of trace links, software traceability is not often implemented in practice, even though research has shown that there are many benefits for software development companies, when implementing it. It becomes obvious that the key to increasing the adoption of software traceability, is by reducing the efforts needed to establish it.

Many research has been conducted on the field of software traceability in the past two decades. The grand challenge for the research field is to achieve *ubiquitous traceability*, which means that traceability is always there, without having to think about getting it there. This goal should be feasible around 2035, according to Gotel et al. (Gotel et al., [2012a](#)) While it might be too ambitious to work towards such a goal in only 2018, this research hopes to provide a starting point, and a general approach to software traceability, that will eventually be able to achieve ubiquitous traceability.

In this research project, we have proposed a vision and theory for Ontological Traceability for Software, as well as provided a design on how it could be used in a popular version control system such as GitLab. Furthermore, a proof-of-concept for the theory has been constructed, baptized the Ubiquitous Traceability Artisan, or UTA. We started of this research project by formulating a main research question for the project:

RQ: “How can bidirectional trace links between software artifacts be realized using an ontology?”

In this section, we answer the sub-questions that have been formulated in Chapter 2, after which we proceed to attempt to answer the main research question stated above. We draw a series of conclusions from the research project, explain some of the validity threats, and finally, we provide some pointers and ideas for future research projects on the subject of Ontological Traceability for Software.

9.1 Sub Research Questions

The first phase of this research contained a thorough literature study, with the goal to gain an extensive understanding in the fields of software traceability and ontology, and how they can aide software production. This phase was guided by the following three sub-questions:

SQ1: "Which trace links can possibly be found between software artifacts and an ontology?"

SQ2: "Which trace links between software artifacts are of the greatest interest by practitioners?"

SQ3: "What is the optimal name, visualization, and storage mechanism for an ontology that serves software traceability?"

These research questions have been answered in the theoretical foundation chapter, and will be addressed in the first subsection of this chapter, section 8.1.1.

The next phase in the research project, was the design of the theory for Ontological Traceability for Software, and implementation of this theory in a proof-of-concept, the UTA. This process was guided by the following two research questions:

SQ4: "How can a trace link between an ontology and a set of software artifacts be realized?"

SQ5: "How can a trace link between a software artifact and an ontology be realized?"

These questions have been answered by proposing and implementing the theory for Ontological Traceability for Software, which has been done in chapters 5 and 7. We will discuss these questions in section 8.1.2.

The following phase in the research project was to identify how trace links can be of help in a collaborative revision tool, where the open-source platform GitLab was chosen as the best option to integrate with. This phase was guided by the following research question:

SQ6: "How can trace links provide benefits to practitioners in a collaborative revision tool such as GitLab?"

This research question is answered by the design for the GitLab integration, and will be addressed in section 8.1.3.

The final phase of this research project is to assess the performance of the implemented proof-of-concept, in comparison with human performance. This phase was guided by the following research question:

SQ7: "How well does the proof-of-concept for ontological traceability for software perform in comparison with humans?"

This research question is answered by the evaluation experiment, and will be discussed in section 8.1.4.

9.1.1 Discussion: Software Traceability in Literature

It was our first objective to identify the possibilities of trace links, in combination with certain software artifacts and the ontology. The results from this explorative literature study are positive, as we can relate each type of software artifact to the ontology, as long as there is a linguistic mapping between the two. Each software artifact contains some linguistic element, and can therefore be related to the ontology. For some artifacts, this might be harder than for others, but beforehand, there are no software artifact types that we have to exclude.

The second sub-question is related to practitioner interest for trace links. Unfortunately, the main conclusion from the literature study, is that in its current state, software traceability is not valued very highly in companies. This is a result of the high costs associated with the establishment and maintenance of software artifact trace links. Most company implementations are driven by research experiments, and even though research shows many benefits that companies can receive from trace links, they are not inclined to implement it, due to the high costs.

However, when trace links are implemented in a company or software project, the trace links between requirements and code, and requirements and test cases (which are also a type of code), are often perceived as valuable by practitioners. It allows product management to identify where a certain requirement is implemented, and how it performs in test cases. This kind of information is normally not available, due to the disconnectedness of requirements and code. Furthermore, these trace links allow developers to better estimate the amount of work that needs to be done for the implementation of a new requirement. These findings were of great importance for the further execution of the research project, as we decided to focus on the generation of trace links between requirements and different types of code.

As a final step in the literature review, we looked into the field of ontology, with a focus on visualization and storage mechanisms. Furthermore, we attempted to come up with a proper name for the ontology that serves software traceability. There are a plethora of visualization and storage mechanisms, most of which integrate with the ontology editor Protégé. For the proof-of-concept, ontology reading performance was of great importance, as many linguistic terms have to be compared to the different elements in the ontology. Therefore, we decided to save to ontology in a local relational database, with an import interface, so it can be retrieved from other tools such as Protégé, or the Visual Narrator. As a visualization package, WebVOLW appeared to have the highest potential: It is open-source, new, and under active development. In the future, WebVOLW can be used to visually edit the ontology, which provides the practitioners with an opportunity to improve the manually generated ontology, using a large touch screen. This aligns with the vision proposed by Van der Werf et al., who propose the *Software Architecture Video Wall*, to aide collaborative

decision making. (Werf et al., 2017) This vision naturally extends towards collaborative ontology enhancement.

The main conclusion that can be drawn from the literature study, is that there is a gap in the literature on the establishment of software traceability, using ontology techniques. Previous attempts that pose some similarity, such as the work of Zhang et al., only use a top-level ontology to aide software traceability. (Zhang et al., 2006) The approach using domain ontologies is a unique one, worth exploring further, due to its potential to eventually become a ubiquitous solution to software traceability.

9.1.2 Discussion: Ontological Traceability for Software Theory

The next goal of the research project, is to establish a method to generate trace links between software artifacts, using an ontology. We elaborated our vision for the use of natural language processing techniques in software development in chapter 4. Our hypothesis that software traceability can be automated with the use of natural language processing techniques, is based on the fact that software artifacts often have a lot of linguistic terms in common. Furthermore, software artifacts that concern the same portion of domain knowledge, often appear to be related. In existing linguistic or information retrieval methods, a mapping between two software artifacts is often set up. An upside to this approach, is that the trace link generation procedure is tailored to the software artifact, yielding proper results. However, it requires a lot of manual labor, as new methods need to be established for each new software artifact. Therefore, we proposed a new approach, where we relate software artifacts to a central point of truth: The ontology. We then use the mappings between the software artifact and the ontology to find evidence for the relatedness of two software artifact instances. The benefit of this approach is that when we want to relate a new type of software artifact to other existing software artifacts, we only have to create a mapping between the software artifact and the ontology, rather than establishing a relation with each other possible artifact. This significantly reduces the complexity of trace link generation.

This is a theory building research project, so after establishing our vision, we present the theory for the *Product Domain Ontology (PDO)*, and the theory for the *PDO Traceability Method* in chapter 5. One important definition we propose in this chapter is the definition for the PDO:

*"A **Product Domain Ontology** is a domain ontology, that covers the conceptualization of a certain software product."*

We provide a formal definition of what the PDO entails, and we provide an extensive overview of how it is created. The definition of the PDO is important, as it is the central piece of the accompanying method for the generation of trace links, which is

the second theory we propose in this chapter.

The PDO Traceability Method, is a method that uses a domain ontology to automatically establish trace links between software artifacts. The method is the key contribution to the literature of this research project. Each phase of the PDO Traceability Method is elaborated in the chapter, and accurate formal definitions of its components are provided.

One of the weak points in the theory is that it heavily relies on the following two assumptions:

Assumption 1: *A complete PDO can be generated applying natural language processing techniques to software artifacts.*

Assumption 2: *We can generate ubiquitous trace links by extracting all domain knowledge from artifact instances, and relating them to the PDO.*

If these assumptions appear to be valid over the coming decades, then the PDO Traceability Method has the potential to provide ubiquitous software traceability. Whether this is the case will remain to be seen, as it is too early to make sensible comments on this. By providing the two theories, we have answered SQ4 and SQ5, on how bidirectional trace links can be realized, using a domain ontology.

9.1.3 Discussion: GitLab Integration

Ontological Traceability for Software can enable many different types of features, when it ultimately becomes ubiquitous in the future. We have described a series of examples of these features, and how they can be implemented in the open-source collaborative revision manager GitLab. We have chosen GitLab, because of its open-source nature, and because of the wide variety of software artifacts that has already been integrated in the product.

The features described in chapter 6, are unique features that are not available in GitLab, or any comparable software product at this point in time. These features are all enabled by software traceability, and can be used to explain its benefit to practitioners.

There are three general types of features that are enabled by Ontological Traceability for Software, are the following:

- **Trace Link Navigation Features**

Features that aide a stakeholder in the software development process to locate a specific artifact instance, by allowing the stakeholder to click on related artifact instances from a certain artifact instance, in order to search for a certain software artifact.

Example: Ontology Based Artifact Navigation

- **Trace Link Information Features**

Features that combine information from various artifact instances, that are related by a trace link. These artifact instances are disconnected, and the combination of their information can provide interesting insights, that were not possible before.

Example: User Story Testing

- **Artifact Generation Features** Features that generate a new artifact, based on information received from a trace link. This artifact can either be an existing type of artifact, (such as a feature test, generated from a user story) or a completely new type of artifact. (such as an impact forecast, generated from trace links from the requirement artifact)

Example: Impact Forecast Generation

Because of the open-source nature of GitLab, the members of the Grimm project at Utrecht University can contribute these features to the GitLab platform themselves, once the research is ready.

9.1.4 Discussion: Proof-of-concept Performance

As this is a theory building research, the majority of this thesis consists of the theories surrounding Ontological Traceability for Software. However, the final research question, SQ7, concerns the implementation and evaluation of a proof-of-concept for the theories proposed in the earlier chapters. Chapter 7 discusses the implementation of the proof-of-concept, while chapter 8 analyses the performance of the proof-of-concept.

The PDO Traceability Method is implemented in the Ubiquitous Traceability Artisan (UTA), and supports the generation of trace links between the following software artifact types:

- **Requirements** - *User Stories*
- **Code** - *Models, Views and Controllers*
- **Test Cases** - *Feature Tests and UI Tests*

To achieve this, the UTA makes use of the ontology generation from user stories, provided by the Visual Narrator tool. (Lucassen et al., 2016a)

For the analysis of the performance of the UTA, an experiment was set up, in which we compare trace links created by the UTA with trace links created by practitioners. For this experiment, a small software product was used, provided by the Bytestack company. Two domain experts concerning the software product under analysis have manually created trace links, which we compare to the trace links the tool generates. For this experiment, only trace links between user stories and feature tests are considered, as the manual creation of these trace links involves a lot of effort.

Even though the performance of the proof-of-concept was not as high as we had hoped, the final numbers are in line with our expectations. The *precision* of the tool

was 0.5, while the *recall* of the tool was 0.6. (The *accuracy* metric did not provide useful feedback, due to a heavily unbalanced data set) The main reason for the lack in performance, is that the PDO is currently generated from a single software artifact: The user stories. While user stories provide a good starting point for the generation of a PDO, they do not contain all domain knowledge of the software product. For example, implementation details are often hidden in user stories, as they often involve non-technical stakeholders. This domain knowledge is important when establishing trace links to implementation artifacts, such as code.

A positive side that can be seen from the results, is that the tool can recognize the trace links, that are obviously linguistically related. When two artifacts share a lot of domain concepts, the UTA establishes a trace link, meaning that the UTA is able to recognize trace links, as long as there is enough linguistic evidence.

A final remark on the performance of the UTA, is that we must remind ourselves that the target for fully ubiquitous traceability is set for 2035, so it is highly improbable that we would have achieved a higher precision and recall in this stadium of the research project. This research proposes a theory for the usage of ontologies in the field of software traceability, and the proof-of-concept shows that this is a good approach to keep researching in the future.

9.2 Main Research Question

In this section, we answer the main research question of this research project, which was formulated as follows:

RQ: "How can bidirectional trace links between software artifacts be realized using an ontology?"

In order to answer this question, a theory for Ontological Traceability for Software has been designed, and evaluated with professors at Utrecht University. After the method was validated, the method has been formalized in this thesis, and it was implemented in the *Ubiquitous Traceability Artisan (UTA)*. As final part of the research project, we have evaluated its performance in an experiment, where we compare the resulting trace links with trace links created by practitioners.

The ontological approach to software traceability was chosen, because it potentially has a series of benefits, which we will discuss here.

It significantly reduces the amount of work that needs to be done per artifact type.

Because the approach uses an ontology as a central point of truth, and establishes traceability by relating artifact instances to this ontology, the trace links no longer rely on individual analysis of a pair of artifact types. For each artifact, the only trace

link we need to establish, is the trace link between the artifact type and the ontology. In previous solutions, each artifact type would have to be related to each other artifact type, if we wanted to have a trace link between these two artifact types. This significantly reduces the amount of work needed to establish software traceability, as the previous solutions needed a specific module for each type of trace link. If n this the amount of artifact types we want to establish trace links for, then the amount of modules needed with the previous solution is n^2 . With the ontological solution, we only need to relate the artifact types to the ontology, so the amount of modules needed to establish the trace link is only n .

The amount of work needed to incorporate new types of software artifact is limited.

In previous solutions, when a new artifact type is added to the traceability solution, trace links need to be established for each existing artifact type. With an increasing amount of artifact types, this takes an increasing amount of time. With the ontological solution, we only have to relate the new artifact type to the ontology, and we can establish trace links with all the already existing trace links.

Due to its foundations in natural language processing, features that go beyond simple traceability become available.

Some of the features that have been discussed before, such as the generation of certain software artifacts, would not be possible with existing software traceability solutions. The generation of artifacts becomes possible because of its background in natural language processing, and the focus on linguistic patterns in software artifacts. In the future, these features can provide practitioners with a lot more benefits than regular software traceability, so this is also an advantage of the ontological approach.

All features concerned, the main advantage of the ontological approach to software traceability, is that it has the potential to ultimately become *ubiquitous*. There are many more advances needed in the fields of natural language processing, but once all relevant domain knowledge can be extracted from software artifacts, the method can provide ubiquitous trace links. Naturally, this only works when the necessary domain knowledge is in the artifacts, and not in the heads of the practitioners, which is yet another problem that needs solving.

As a final point of discussion, we can argue whether the established trace links are truly bidirectional. In this research, we did not find a counter example, where the inverse trace links was in fact not a trace link, but this might be true in some edge cases. The method establishes a trace link if the similarity score is over a certain threshold. This similarity score is not directional. If in future experiments, we encounter this problem of directional trace links, the procedure will need to be adapted. The occurrence of this problem however is very unlikely, as the similarity score is based of the

intersection of sub-ontologies, which will be the empty set in the case of a directional trace links.

9.2.1 Establishing Trace Links Using an Ontology

In this research, we established the theory of Ontological Traceability for Software, in two parts: Creating a *Product Domain Ontology(PDO)*, and using this ontology to establish trace links. It was important for the research project to split the theory into the two theories they are now. In this way, we have defined a traceability method, that works with every software product domain ontology, rather than only our own PDO. There are software companies that create an ontology for their software product, and they can use the traceability method proposed in this research, without using the PDO. However, the theory for the creation of an ontology from software artifacts is equally as important, because we want the traceability to be ubiquitous, and this is not the case if it requires the manual creation of an ontology for the software product.

The creation of trace links is done with the PDO Traceability Method, that is implemented in the proof-of-concept. This method answers how trace links can be established between software artifacts, using an ontology:

1. We start of with a set of software artifact instances that we want to create trace links for, and a domain ontology that represents the software product that is associated with the software artifacts.
2. The next step is to extract all domain knowledge from the artifacts that will be linked. For each different type of software artifact, we need a template for this. The template will allow a tool to convert an artifact instance to a set of linguistic terms, related to the artifact instance.
3. This is the step where the ontology comes in to play. We combine the linguistic terms with the ontology, in order to construct a sub-ontology. This can be seen as a domain ontology, but just for the artifact instance that we are analyzing. This domain ontology is a subset of the ontology used to construct it.
4. Once we have constructed the sub-ontologies, we will compare them, and see how alike they are. If two sub-ontologies are very alike, this means they contain the same domain knowledge, and therefore, they have a high probability to be related. This step yields a similarity score for the two sub-ontologies. This similarity score can also be considered as a relatedness score for the artifact instances.
5. In the final step, we check if the similarity score of the two artifact instances is high enough, in which case, we state that the ontology provides enough evidence to state that the artifact instances are related. When this is the case, we establish a trace link between the two artifact instances.

6. The above steps are repeated for each possible combination of artifact instances, to explore every possible trace link. The only exception to this, are the inverse trace links of already explored artifacts: Trace links are bidirectional, so if the trace link (a_1, a_2) already exists, the candidate trace link (a_2, a_1) does not need to be explored.

Many optimizations are still needed to the various modules of the traceability method described in this thesis, but the global process is solid, and has the potential to achieve ubiquitous traceability in the future. It is therefore that we baptized the proof-of-concept the *Ubiquitous Traceability Artisan(UTA)*, with the ubiquitous part more as a future vision than an accurate description of the current capabilities of the tool. It is only a proof-of-concept, for all.

With the insights we gained through the process of executing this research project, we proceed to draw a series of conclusions from this research project in the next section.

9.3 Conclusions

Now that the main findings of this research project are discussed, we can draw some conclusions from this research project in this section.

9.3.1 Ontological Traceability for Software

In this research, we investigated whether techniques from the field natural language processing and ontology can aide in the automatic creation and maintenance of software artifact trace links. We did so by building a theory for Ontological Traceability for Software, which we validated in a proof-of-concept.

The first major conclusion that we can draw, is that the *PDO Traceability Method* appeared to be an effective method for the creation of trace links, using an ontology and a set of linguistic terms extracted from software artifact instances. We can draw this conclusion, because the PDO Traceability Method, implemented in the Ubiquitous Traceability Artisan(UTA), performed very well under the following two circumstances:

- The *linguistic terms*, extracted from the artifact instances, provide a complete representation of the artifact instances. This means that the majority of the domain knowledge embedded in the artifact instances was extracted.
- The section of the *PDO* that relates to the artifact instances, is complete and thorough. This means that the resulting sub-ontologies from the application of the linguistic terms to the PDO results in complete sub-ontologies. The result of this is that the domain knowledge that was captured in the linguistic terms, is not lost when constructing the sub-ontologies.

There are a couple of examples in the experiment, where this was the case. When testing *create*, *read*, *update*, and *delete* functionalities on *model* objects, the user story and the feature test contain the same domain knowledge, as can be seen in the example below:

User Story: "As an administrator, I want to create events, so I can manage events in the system."

Feature Test: "CreateEventTest"

In the example, the two artifact instances contain a lot of the same domain knowledge, which makes it easy for the UTA to establish a trace link. The resulting sub-ontologies can be seen in Figure 9.1.

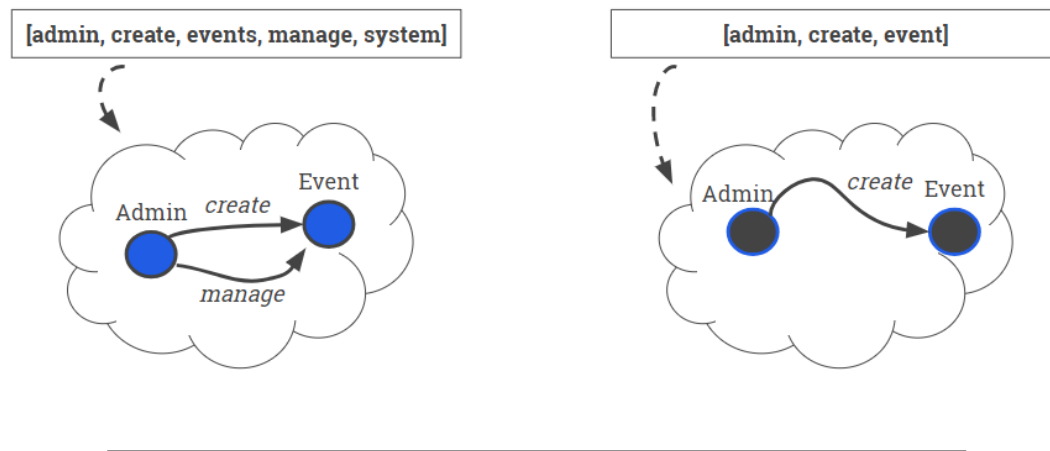


FIGURE 9.1: Example Sub-ontologies

As can be seen in Figure 9.1, the sub-ontology of the feature test, is fully comprised in the sub-ontology for the user story, which means that the UTA will create a trace link for this example.

This works so well, because the domain knowledge is provided in the form of structured language, which is easier to understand for the UTA. As soon as the user stories or feature tests become slightly more complex, the UTA misses quite a few trace links. This is the result of subpar domain concept extraction, and subpar ontology learning: When the domain knowledge is not available to the PDO Traceability Method, it cannot create a trace link.

The drawback is that both linguistic term extraction and ontology learning need to be equally good, in order to achieve desired results. This is the result of the fact, that if a linguistic term is absent, the related concept will not be in the sub-ontology, and if a concept is not extracted into the linguistic terms, it will also not be in the sub-ontology, even though the concept is in the ontology. The only solution, is to improve both *ontology learning* and *domain knowledge extraction*, from various software artifacts.

The main conclusions from this research on Ontological Traceability for Software can be summed up as follows:

- The *PDO Traceability Method* performs as expected, when all the necessary domain knowledge is available to the method. This however, is not the case for many software artifacts.
- The *UTA* has a precision of 0.51 and a recall of 0.83, meaning that its performance is still subpar to human trace links, and not yet close to be ubiquitous. However, considering that this was a proof-of-concepts, these results can be considered promising, especially as the *UTA* delivered very few false negatives.
- In cases where all domain knowledge is available to the *UTA*, it delivers promising results. This suggests that if *ontology learning* and *domain knowledge extracting* from software artifacts is improved, the tool will naturally improve as well.
- Ontological Traceability for Software has the potential to become *ubiquitous traceability* in the future, but at the moment, it is not close to realizing it.

9.3.2 Software Traceability in Practice

During this research project, we also uncovered a lot about software traceability in practice. First of all, when approaching companies, most of them did not know what software traceability entailed. After explaining what software traceability means, they often replied with the following question: "*And why is that useful?*". It usually takes a big effort to convince a practitioner that software traceability is worth their while, which is why we spend a great deal of this research project to describing how our findings can be implemented for practitioners.

We designed the *UTA* to be able to integrate with third party tooling. More specific, we created a couple of designs for the open-source collaborative revision management platform GitLab. GitLab already stores the majority of the software artifacts that are related to a project, including the requirements, code, and test cases. By integrating our findings with the GitLab platform, software traceability would become available to more practitioners, who can then experience its benefits.

With our interactions with practitioners, it became clear that an ubiquitous solution to software traceability, is the only viable solution. The efforts to establish traceability are too high for practitioners, and they do not see the benefits that it can bring them. The main conclusion from this part of the research project, is therefore the following:

- By implementing findings from software traceability research into an open-source platform such as GitLab, we can bring the benefits of software traceability to practitioners. This will allow them to experience the benefits of software traceability, without experiencing the costs that it currently takes to establish it. In return, the software traceability research community will receive a lot of data from practitioners, in the form of software artifacts, generated trace links, and traceability usage. This data can be used to do additional research into software traceability, which would otherwise be unfeasible, due to the efforts needed to get a significantly large data set.

9.4 Validity Threats

In this section, we discuss the validity of the research, its reliability, and the limitations on the research project.

9.4.1 External Validity

The *external validity* refers to the generalizability of the research. The performance results for the UTA were received from a single data set only, and a single experiment with practitioners. This is intentional, as practitioner time is very scarce, and the pairwise comparison of trace links in the experiment takes a lot of time. This is also the reason why only the traceability from user stories to feature tests has been evaluated: They provide the most benefits according to the practitioners that participated.

The UTA needs to be validated on a lot more different data sets, preferably even different kinds of software products, to see if the performance statistics are accurate or not. Also, the validation is done by comparison with trace links created by humans, who can of course make mistakes: The performance statistics might also be affected by human error in this case.

9.4.2 Internal Validity

The *internal validity* refers to the manner in which the research project has been conducted. The implementation of the PDO Traceability Method, in the UTA, has been open-sourced, and is available to inspect on GitLab. This significantly reduces the risk of bugs that remain unseen, as the entire research community can inspect it, and play around with it. The UTA comes with the data set that has been used to conduct the experiments, this includes:

- The *user stories* that describe the system, and are used to construct the PDO, are available in the UTA, when downloaded together with the test set. They are also available in Appendix A.
- The *feature tests* that are used for the experiment, are also available in the UTA, and their names and descriptions can be seen in Appendix B.
- The *code* software artifacts can be used in the UTA, as they have been imported. However, the class files remain hidden, as the company in the experiment does not want to open-source its product at this moment. The artifacts can still be used for analyzing traceability however, as all linguistic terms related to the artifact are already extracted, and available in the test set.

9.4.3 Reliability

The entire research process has been strictly documented in chapter 2, in order to ensure repeatability by other practitioners. The process of creating the theories is rather hard to replicate, as it was a creative process, conducted over several months,

with the combined efforts of all the researchers associated with the Grimm project at Utrecht University. However, the evaluation of the proof-of-concept, and the rationale behind the architecture of the proof-of-concept, is strictly documented in this thesis.

9.4.4 Limitations

The biggest limitation on this research, is the sample size of the evaluation. Unfortunately, creating a data set of trace links to compare with is a difficult and time consuming task, which is why we initially had to limit ourselves to the evaluation of trace links between user stories and feature tests. We explored other avenues, but there is no open data set for software traceability, that satisfies our needs. This is also a big problem in the research field. (Antoniol et al., 2017)

Eventually, we were able to convince the practitioners to participate in a second round experiment, which tripled our sample size. This improved the results dramatically, and displayed the the results found in the traceability between user stories and feature tests, transfer to other artifacts such as code.

We hope that by eventually integrating with GitLab, future evaluations can be done on a data set with a larger sample size. By involving an open-source community, the amount of effort it takes practitioners to create a data set is much lower.

9.5 Future Research

As this was only the first research project into Ontological Traceability for Software, many follow up research projects can be done. In this section, we present some of our ideas for further research, that will help the research community of software traceability to achieve its goal of ubiquitous traceability.

9.5.1 Extensive Evaluation of the UTA

The UTA needs further evaluation. One interesting research project, is to take an open-source software product, that makes use of user story requirements, and attach it to the UTA. The creation of a comparison data set is then still a tedious and long process, but as this research project only focusses on the evaluation of the UTA, it is manageable. This will result in more accurate performance statistics on the current implementation of the UTA. Furthermore, by creating a large test data set that involves an open-source software product, we can create a benchmark data set, to which all our further traceability endeavours can be tested.

9.5.2 Improvement of Ontology Learning in PDO Creation

One of the limiting factors in the current proof-of-concept, is the ontology learning process. This is done by the Visual Narrator tool, which is specifically designed

to extract conceptual models (lightweight ontologies) from user stories. If we can extend the Visual Narrator tool to accept input from multiple software artifact types, we can greatly improve the resulting PDO, as the user stories only pose a single perspective on the system. It could also be interesting to compare ontology creation from a different type of requirement, use cases or epic-stories for example, to see which type of requirement specification is most suited for ontology creation.

9.5.3 Integration of the UTA in GitLab

If the improvements above are investigated, the trace links resulting from the UTA could provide real benefits to practitioners. It will therefore be useful to implement the traceability features described in chapter 6 of this thesis in GitLab. This will bring the benefits of traceability to a larger audience of practitioners, resulting in more data for the research community.

9.5.4 Evaluate Static and Run-time Sub-ontology Creation

At this point in time, the UTA creates the sub-ontologies used for the establishment of trace links at run-time. It might be beneficial for processing speed, to generate the sub-ontology related to an artifact, at the moment that it enters the UTA for the first time. In this case, the sub-ontologies only have to be looked up at run-time, rather than being created. When the PDO becomes larger, the amount of time it takes to create a sub-ontology also becomes larger. Therefore, it could save a lot of processing time, if the sub-ontology is created upfront. However, this comes paired with a need for a larger storage database.

Bibliography

- Antoniol, G. et al. (2001). "Design-code traceability recovery: selecting the basic linkage properties." In: *Science of Computer Programming*, pp. 213–234.
- Antoniol, G. et al. (2002). "Recovering traceability links between code and documentation." In: *IEEE transactions on software engineering*, pp. 970–983.
- Antoniol, G. et al. (2017). "Grand Challenges of Traceability: The Next Ten Years". In: *Proceedings of the Grand Challenges of Traceability 2017*.
- Arkley, P. and S. Riddle (2005). "Overcoming the traceability benefit problem". In: *IEEE*, pp. 385–389.
- Blaauboer, F., K. Sikkel, and M.N. Aydin (2007). "Deciding to adopt requirements traceability in practice". In: *In International Conference on Advanced Information Systems Engineering. Springer, Berlin, Heidelberg*. Pp. 294–308.
- Bontcheva, K. and M. Sabou (2006). "Learning ontologies from software artifacts: Exploring and combining multiple sources." In: *In Workshop on Semantic Web Enabled Software Engineering (SWESE)*.
- Borg, M., P. Runeson, and A. Ardö (2014). "Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability". In: *Empirical Software Engineering*, pp. 1565–1616.
- Bouillon, E., P. Mäder, and I. Philippow (2013). "A survey on usage scenarios for requirements traceability in practice". In: *International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer, Berlin, Heidelberg*. Pp. 158–173.
- Buitelaar, P., P. Cimiano, and B. Magnini (2005). "Ontology learning from text: An overview". In: *Ontology learning from text: Methods, evaluation and applications*. Pp. 3–12.
- Choi, N., I.Y. Song, and H. Han (2006). "A survey on ontology mapping". In: *ACM Sigmod Record*, 35(3). Pp. 34–41.
- Cimiano, P. and J. Völker (2005). "text2onto: A Framework for Ontology Learning and Data-driven Change Discovery". In: *International conference on application of natural language to information systems. Springer, Berlin, Heidelberg*. Pp. 227–238.
- Cleland-Huang, J., O.C. Gotel, and A. Zisman (2012). "Software and systems traceability". In: *Springer, Berlin, Heidelberg*.
- Cleland-Huang, J. et al. (2014). "Software traceability: trends and future directions". In: *In Proceedings of the on Future of Software Engineering. ACM*. Pp. 55–69.

- Doan, A. et al. (2002). "Learning to map between ontologies on the semantic web". In: *Proceedings of the 11th international conference on World Wide Web*. ACM. Pp. 662–673.
- Drymonas, E., K. Zervanou, and E.G. Petrakis (2010). "Unsupervised ontology acquisition from plain texts: the OntoGain system". In: *International Conference on Application of Natural Language to Information Systems*. Springer, Berlin, Heidelberg. Pp. 277–287.
- Dudáš, M., O. Zamazal, and V. Svátek (2014). "Roadmapping and navigating in the ontology visualization landscape". In: *International Conference on Knowledge Engineering and Knowledge Management*, Springer, Cham. Pp. 137–152.
- Egyed, A. and P. Grünbacher (2002). "Automating requirements traceability: Beyond the record replay paradigm." In: *IEEE*, pp. 163–163.
- Ehrig, M. and S. Staab (2004). "QOM—quick ontology mapping". In: *International Semantic Web Conference*. Springer, Berlin, Heidelberg. Pp. 683–697.
- Gotel, O.C. and C.W. Finkelstein (1994). "An analysis of the requirements traceability problem". In: *Proceedings of the First International Conference on Requirements Engineering*. IEEE. Pp. 94–101.
- Gotel, O.C. et al. (2012a). "The grand challenge of traceability (v1. 0)". In: *Software and Systems Traceability*. Springer, London. Pp. 343–409.
- Gotel, O.C. et al. (2012b). "The quest for ubiquity: A roadmap for software and systems traceability research." In: *Requirements Engineering Conference (RE), 2012 20th IEEE International*. IEEE. Pp. 71–80.
- Grüber, T.R. (1995). "Toward principles for the design of ontologies used for knowledge sharing?" In: *International journal of human-computer studies*, pp. 907–928.
- Guarino, N. (1997). "Semantic matching: Formal ontological distinctions for information organization, extraction, and integration". In: *International Summer School on Information Extraction*. Springer, Berlin, Heidelberg. Pp. 139–170.
- (1998). "Formal ontology in information systems". In: *Proceedings of the first international conference (FOIS'98)*. IOS press.
- Katifori, A. et al. (2007). "Ontology visualization methods—a survey". In: *ACM Computing Surveys (CSUR)*.
- Lohmann, S. et al. (2016). "Visualizing ontologies with VOWL". In: *Semantic Web*, 7(4). Pp. 399–419.
- Lu, J. et al. (2007). "SOR: a practical system for ontology storage, reasoning and search". In: *Proceedings of the 33rd international conference on Very large data bases. VLDB Endowment*. Pp. 1402–1405.
- Lucassen, G. et al. (2016a). "Extracting conceptual models from user stories with Visual Narrator". In: *Requirements Engineering Volume 22, Number 3*. Springer. Pp. 339–358.
- Lucassen, G. et al. (2016b). "Improving agile requirements: the quality user story framework and tool". In: *Requirements Engineering*, 21(3), pp. 383–403.

- Lucassen, G. et al. (2016c). "The use and effectiveness of user stories in practice". In: *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, Cham. Pp. 205–222.
- Lucassen, G. et al. (2017). "Behavior-Driven Requirements Traceability via Automated Acceptance Tests". In: *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*. IEEE. Pp. 431–434.
- Lucia, A. De et al. (2007). "Recovering traceability links in software artifact management systems using information retrieval methods." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)*.
- Mar Roldan-Garcia, M. del and J.F. Aldana-Montes (2008). "A Protégé Plugin for Storing OWL ontologies in Relational Databases". In:
- Missikoff, M., R. Navigli, and P. Velardi (2002). "Integrated approach to web ontology learning and engineering." In: *Computer*, 35(11), pp. 60–63.
- Mäder, P. and A. Egyed (2011). "Do software engineers benefit from source code navigation with traceability? — An experiment in software change management." In: *26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. Pp. 444–447.
- Noll, R.P. and M.B. Ribeiro (2007). "Enhancing traceability using ontologies." In: *Proceedings of the 2007 ACM symposium on Applied computing*. ACM. Pp. 1496–1497.
- Oliver, I. (1993). "Programming classics: implementing the world's best algorithms". In: *New York: Prentice Hall*. Pp. 347–347.
- Ramesh, B. (1998). "Factors influencing requirements traceability practice." In: *Communications of the ACM*. Pp. 37–44.
- Ramesh, B. and M. Jarke (2001). "Toward reference models for requirements traceability." In: *IEEE transactions on software engineering*, pp. 58–93.
- Robeer, M. et al. (2016). "Automated extraction of conceptual models from user stories via NLP." In: *2016 IEEE 24th international Requirements engineering conference (RE)*, IEEE. Pp. 196–205.
- Sivakumar, R., P.V. Arivoli, and A.V.V.M. Sri (2011). "Ontology visualization PRO-TÉGÉ tools – a review". In: *International Journal of Advanced Information Technology*. Pp. 1–11.
- Wautelet, Y. et al. (2014). "Unifying and extending user story models". In: *International Conference on Advanced Information Systems Engineering*. Springer, Cham. Pp. 211–225.
- Werf, J.M.E. van der et al. (2017). "Facilitating Collaborative Decision Making with the Software Architecture Video Wall". In: *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference*, IEEE. Pp. 137–140.
- Wieringa, R.J. (2014). "Design science methodology for information systems and software engineering". In: *Springer*.
- Wong, W., W. Liu, and M. Bennamoun (2012). "Ontology learning from text: A look back and into the future". In: *ACM Computing Surveys (CSUR)*.

- Zhang, Y. et al. (2006). "An ontology-based approach for traceability recovery." In: *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006)*, pp. 36–43.
- (2008). "Ontological approach for the semantic recovery of traceability links between software artefacts". In: *IET Software*. Pp. 185–203.
- Zhou, J. et al. (2006). "Minerva: A scalable OWL ontology storage and inference system". In: *Asian Semantic Web Conference*. Springer, Berlin, Heidelberg. Pp. 429–443.

Appendix A

UAS User Stories

#1: As a member, I want to register in the system, so my membership is registered with the association.

#2: As a member, I want to receive a confirmation e-mail, so I know my registration was successful.

#3: As a member, I want to receive e-mails from my trainer, so I'm informed of upcoming events.

#4: As an administrator, I want to create events, so I can manage events in the system.

#5: As an administrator, I want to read events, so I can see information of a specific event.

#6: As an administrator, I want to update events, so I can change its information.

#7: As an administrator, I want to delete events, so I can cancel events.

#8: As an administrator, I want to see an overview of events in a calendar, so I can see upcoming events.

#9: As an administrator, I want to add members to an event, so I can let them participate.

#10: As an administrator, I want to select if an event is billable, so I can let it count for the dues.

#11: As an administrator, I want to set extra costs to an event, so I can bill extra costs.

#12: As an administrator, I want to create a group, so I can manage groups.

#13: As an administrator, I want to add users to a group, so I can manage who is in the group.

#14: As an administrator, I want to read a group, so I can see who is in the group.

#15: As an administrator, I want to update a group, so I can update who is in group.

#16: As an administrator, I want to delete a group, so I can remove redundant groups.

#17: As an administrator, I want set a type to an event, so I can indicate what type of event it is.

#18: As an administrator, I want to change data of members, so I can update their data in case of a change.

#19: As an administrator, I want to generate a report, so I can see the dues to be paid by a member.

- #20: As an administrator, I want to generate a report for trainers, so I can see the hours they worked.
- #21: As a member, I want to know the dues to be paid, so I can pay them.
- #22: As a member, I want to receive an e-mail when new dues have to be paid, so I can pay them.
- #23: As a trainer, I want to see events, so I know which events are upcoming.
- #24: As a trainer, I want to e-mail groups of members, so I can share information with my groups.
- #25: As a trainer, I want to e-mail a member, so I can share information with them.
- #26: As an administrator, I want to assign costs to the first hour of events, so I can manage the prices.
- #27: As an administrator, I want to assign costs to extra hours of events, so I can manage the prices.
- #28: As an administrator, I want to know the costs per member, so I can bill the dues.
- #29: As an administrator, I want to set the welcome text after registration, so I can update the confirmation e-mail.
- #30: As an administrator, I want to send an e-mail with the dues, so my members know the amount of dues to be paid.
- #31: As an administrator, I want to see a home screen, so I get a confirmation that I logged in.
- #32: As an administrator, I want to set a date range on the calendar, so I can see events over time.
- #33: As an administrator, I want to select a group in the calendar, so I can only see events attached to that group.
- #34: As an administrator, I want to create a member, so I can add the member when they didn't register.
- #35: As an administrator, I want to select if a member is a digimember, so I can manage which members are digimembers.
- #36: As an administrator, I want to search members, so I can find a certain member.
- #37: As an administrator, I want to search groups, so I can find a certain group.
- #38: As an administrator, I want to search events, so I can find a certain event.
- #39: As an administrator, I want to set a date range on a report, so I can generate a report on a period of time.
- #40: As an administrator, I want to login, so I can access the system.
- #41: As an administrator, I want to logout, so I can safely leave the system.
- #42: As an administrator, I want to receive a password reset link, so I can reset my password if I forget it.
- #43: As an administrator, I want to remember my login data, so I can stay logged in for a longer time.
- #44: As a user, I want to see my name in the system, so I know I'm logged in with the correct account.

Appendix B

UAS Test Cases

#1: CreateEventTest

Functions:

- testCreateNonRepeatableEvent
- testCreateRepeatableEvent
- testCreateEventWithUsers
- testCreateEventWithEmptyDayAndNotBillable
- testCreateEventWithoutData

#2: DeleteEventTest

Functions:

- testDeleteEvent

#3: UpdateEventTest

Functions:

- testUpdateNonRepeatableEvent
- testUpdateRepeatableEvent
- testUpdateEventWithUsers
- testUpdateeventWithEmptyDayAndNotBillable
- testUpdateEventWithoutData

#4: CreateGroupTest

Functions:

- testCreateGroup
- testCreateGroupWithUser
- testCreateGroupWithoutData

#5: DeleteGroupTest

Functions:

- testDeleteGroup

#6: UpdateGroupTest

Functions:

- testUpdateGroup
- testUpdateGroupAndRemoveAllUsers
- testUpdateGroupAndDontAddExtraUser
- testUpdateGroupAndOntherUser
- testUpdateGroupWithoutData

#7: UpdateSettingsTest*Functions:*

- testUpdateSetting

#8: CreateUserTest*Functions:*

- testCreateUser
- testCreateUserWithEventsAndGroups
- testCreateUserWithoutData

#9: DeleteUserTest*Functions:*

- testDeleteUser

#10: UpdateUserTest*Functions:*

- testUpdateUser
- testUpdateUserWithGroupsAndEvents
- testUpdateUserWithoutData

Appendix C

Scientific Paper

The scientific paper that presents the results of this thesis, can be found in this chapter.

Appendix D

Trace Links Created by Practitioners

D.1 User Stories to Feature Tests

User Story	Feature Test
#4: As an admin, I want to create events.	"CreateEventTest"
#7: As an admin, I want to delete events.	"DeleteEventTest"
#6: As an admin, I want to update events.	"UpdateEventTest"
#12: As an admin, I want to create a group.	"CreateGroupTest"
#16: As an admin, I want to delete a group.	"DeleteGroupTest"
#15: As an admin, I want to update a group.	"UpdateGroupTest"
#26: As an admin, I want to assign costs to the first hour.	"UpdateSettingsTest"
#27: As an admin, I want to assign costs to extra hours.	"UpdateSettingsTest"
#29: As an admin, I want to set the welcome text.	"UpdateSettingsTest"
#1: As a member, I want to register in the system.	"CreateUserTest"

D.2 User Stories to Models

User Story	Model
#3: As a member, I want to receive e-mails from my trainer.	"EventModel"
#4: As an admin, I want to create events.	"EventModel"
#17: As an admin, I want to set a type to an event.	"EventModel"
#5: As an admin, I want to read events.	"EventModel"
#6: As an admin, I want to update events.	"EventModel"
#7: As an admin, I want to delete events.	"EventModel"
#9: As an admin, I want to add members to an event.	"EventModel"
#10: As an admin, I want to select if an event is billable.	"EventModel"
#11: As an admin, I want to set extra costs to events.	"EventModel"
#26: As an admin, I want to assign costs to the first hour.	"SettingModel"
#27: As an admin, I want to assign costs to extra hours.	"SettingModel"
#12: As an admin, I want create a group.	"GroupModel"
#13: As an admin, I want to add users to a group.	"GroupModel"
#14: As an admin, I want to read a group.	"GroupModel"
#15: As an admin, I want to update a group.	"GroupModel"
#16: As an admin, I want to delete a group.	"GroupModel"
#1: As an member, I want to register in the system.	"UserModel"
#18: As an admin, I want to change data of members.	"UserModel"
#35: As an admin, I want to select if a member is a digimember.	"UserModel"

D.3 User Stories to Controllers

User Story	Controller
#3: As a member, I want to receive e-mails from my trainer.	"EventController"
#4: As an admin, I want to create events.	"EventController"
#17: As an admin, I want to set a type to an event.	"EventController"
#5: As an admin, I want to read events.	"EventController"
#6: As an admin, I want to update events.	"EventController"
#7: As an admin, I want to delete events.	"EventController"
#9: As an admin, I want to add members to an event.	"EventController"
#10: As an admin, I want to select if an event is billable.	"EventController"
#11: As an admin, I want to set extra costs to events.	"EventController"
#26: As an admin, I want to assign costs to the first hour.	"SettingController"
#27: As an admin, I want to assign costs to extra hours.	"SettingController"
#12: As an admin, I want create a group.	"GroupController"
#13: As an admin, I want to add users to a group.	"GroupController"
#14: As an admin, I want to read a group.	"GroupController"
#15: As an admin, I want to update a group.	"GroupController"
#16: As an admin, I want to delete a group.	"GroupController"
#18: As an admin, I want to change data of members.	"UserController"

Appendix E

Trace Links Created by UTA

E.1 User Stories to Feature Tests

User Story	Feature Test
#4: As an admin, I want to create events.	"CreateEventTest"
#6: As an admin, I want to update events.	"UpdateUserTest"
#6: As an admin, I want to update events.	"UpdateEventTest"
#7: As an admin, I want to delete events.	"DeleteEventTest"
#12: As an admin, I want to create a group.	"CreateGroupTest"
#15: As an admin, I want to update a group.	"UpdateGroupTest"
#15: As an admin, I want to update a group.	"UpdateUserTest"
#16: As an admin, I want to delete a group.	"DeleteGroupTest"
#18: As an admin, I want to change data of members.	"UpdateUserTest"
#29: As an admin, I want to set the welcome text.	"UpdateUserTest"
#32: As an admin, I want to set a range on the calendar.	"UpdateUserTest"
#39: As an admin, I want to set a range on a report.	"UpdateUserTest"

TABLE E.1: Trace Links Identified by the UTA

E.2 User Stories to Models

User Story	Feature Test
#3: As a member, I want to receive e-mails from my trainer.	"EventModel"
#4: As an admin, I want to create events.	"EventModel"
#5: As an admin, I want to read events.	"EventModel"
#6: As an admin, I want to update events.	"EventModel"
#7: As an admin, I want to delete events.	"EventModel"
#8: As an admin, I want to see an overview of events.	"EventModel"
#9: As an admin, I want to add members to an event.	"EventModel"
#10: As an admin, I want to select if an event is billable.	"EventModel"
#11: As an admin, I want to set extra costs to an event.	"EventModel"
#27: As an admin, I want to assign costs to the first hour.	"EventModel"
#28: As an admin, I want to know the costs per member.	"EventModel"
#30: As an admin, I want to send e-mail with dues.	"EventModel"
#31: As an admin, I want to see a home screen.	"EventModel"
#32: As an admin, I want to set a date range.	"EventModel"
#33: As an admin, I want to select a group in the calendar.	"EventModel"
#38: As an admin, I want to search events.	"EventModel"
#12: As an admin, I want to create a group.	"GroupModel"
#13: As an admin, I want to add users to a group.	"GroupModel"
#14: As an admin, I want to read a group.	"GroupModel"
#15: As an admin, I want to update a group.	"GroupModel"
#16: As an admin, I want to delete a group.	"GroupModel"
#24: As a trainer, I want to e-mail groups.	"GroupModel"
#33: As an admin, I want to select a group in the calendar.	"GroupModel"
#37: As an admin, I want to search groups.	"GroupModel"

TABLE E.2: Trace Links Identified by the UTA

E.3 User Stories to Controllers

User Story	Feature Test
#3: As an member, I want to receive e-mails from my trainer.	"EventController"
#4: As an admin, I want to create events.	"EventController"
#5: As an admin, I want to read events.	"EventController"
#6: As an admin, I want to update events.	"EventController"
#7: As an admin, I want to delete events.	"EventController"
#8: As an admin, I want to see an overview of events.	"EventController"
#9: As an admin, I want to add members to events.	"EventController"
#10: As an admin, I want to select if an event is billable.	"EventController"
#11: As an admin, I want to set extra costs to an event.	"EventController"
#17: As an admin, I want to set the type of an event.	"EventController"
#23: As a trainer, I want to see events.	"EventController"
#26: As an admin, I want to assign costs to the first hour.	"EventController"
#27: As an admin, I want to set extra costs.	"EventController"
#32: As an admin, I want to set a date range.	"EventController"
#38: As an admin, I want to search events.	"EventController"
#12: As an admin, I want to create a group.	"GroupController"
#13: As an admin, I want to add users to a group.	"GroupController"
#14: As an admin, I want to read a group.	"GroupController"
#15: As an admin, I want to update a group.	"GroupController"
#16: As an admin, I want to delete a group.	"GroupController"
#24: As an admin, I want to e-mail groups of members.	"GroupController"
#33: As an admin, I want to select a group in the calendar.	"GroupController"
#37: As an admin, I want to search groups.	"GroupController"
#19: As an admin, I want to generate a report.	"ReportController"
#20: As an admin, I want to generate a report for trainers.	"ReportController"
#39: As an admin, I want to set a date range on a report.	"ReportController"

TABLE E.3: Trace Links Identified by the UTA