RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Université de Toulouse

Doctorat de l'Université de Toulouse

préparé à l'Université Toulouse - Jean Jaurès

UOOR: une approche orientée-objets pour l'Ingénierie des Exigences

---

Thèse présentée et soutenue, le 18 décembre 2024 par

Mariya NAUMCHEVA

École doctorale
EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse

Spécialité
Informatique et Télécommunications

Unité de recherche
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Sophie MARCAILLOU - EBERSOLD et Bertrand MEYER

Composition du jury
Mme Hélène WAESELYNCK, Présidente, CNRS Occitanie Ouest
Mme Rabéa AMEUR-BOULIFA, Rapporteure, Télécom Paris
M. Yves LEDRU, Rapporteur, Université Grenoble Alpes
Mme Catherine DUBOIS, Examinatrice, ENSIIE
M. Jean-Michel BRUEL, Examinateur, Université Toulouse - Jean Jaurès
Mme Nelly BENCOMO, Examinatrice, Durham University
Mme Sophie MARCAILLOU-EBERSOLD, Directrice de thèse, Université Toulouse - Jean Jaurès
M. Bertrand MEYER, Co-directeur de thèse, Constructor Institute of Technology

Résumé

Dans la pratique industrielle, les exigences sont un élément indispensable de tout projet logiciel sérieux. Dans l'étude universitaire du génie logiciel, les exigences sont l'un des sujets les plus étudiés. Pourtant, les exigences telles qu'elles sont pratiquées dans l'industrie n'utilisent que très peu les concepts proposés dans la littérature sur les exigences. La présente thèse part d'une hypothèse sur les *causes* de cette situation et propose une *solution* pour y remédier:

- La cause identifiée est que *changement* est le principal facteur affectant l'application pratique des techniques d'exigences, même les mieux élaborées. L'encre du document d'exigences est à peine sèche que l'environnement du système et les points de vue des parties prenantes sur le système commencent à évoluer. Les méthodes d'évaluation des besoins qui partent du principe que les besoins peuvent être définis une fois pour toutes pour guider le développement sont vouées à l'échec.

- La solution proposée est une *méthode* de gestion des exigences, appelée UOOR, qui unifie de nombreux concepts d'exigences connus et quelques nouveaux concepts, dans un cadre entièrement conçu pour gérer des changements continus tout au long du cycle de vie du projet.

La méthode UOOR (l'acronyme signifie Unified Object-Oriented Requirements) englobe les techniques d'exigences couramment utilisées, à savoir les scénarios, et les intègre dans le processus de développement logiciel continu. Cette thèse introduit la notion de traçabilité transparente des exigences, qui repose sur la propagation des liens de traçabilité, eux-mêmes basés sur des propriétés formelles des relations entre les artefacts du projet. Comme preuve de concept, la thèse présente un outil de traçabilité, à intégrer dans un IDE général, qui permet de relier les exigences à d'autres artefacts du projet logiciel, d'afficher des notifications de changement dans les exigences et de tracer ces changements dans les éléments de projet connexes.

L'approche UOOR n'est pas seulement une proposition théorique mais a été conçue pour une utilisation pratique et a été appliquée à des applications significatives, y compris une étude de cas complète, détaillée dans un livre à venir.

Abstract

In industrial practice, requirements are an indispensable element of any serious software project. In the academic study of software engineering, requirements are one of the heavily researched subjects. And yet requirements as practiced in industry makes shockingly sparse use of the concepts propounded in the requirements literature. The present thesis starts from an assumption about the *causes* for this situation, and proposes a *remedy* to redress it:

- The posited explanation is that *change* is the major factor affecting the practical application of even the best-intentioned requirements techniques. Hardly is the ink dry on the requirements document that the system's environment and the stakeholders' views about the system start evolving. Requirement methods that assume that requirements can be set once and for all to guide the development are doomed.
- The proposed remedy is a requirements *method*, called UOOR, which unifies many known requirements concepts and a few new ones in a framework entirely devised to accommodate and support seamless change throughout the project lifecycle.

The UOOR method (the acronym stands for Unified Object-Oriented Requirements) encompasses the commonly used requirements techniques, namely, scenarios, and integrates them into the seamless software development process. The thesis introduces the notion of seamless requirements traceability, which relies on propagation of traceability links, themselves based on formal properties of relations between project artifacts. As a proof of concept the thesis presents a Traceability tool, to be integrated into a general-purpose IDE, that provides the ability to link requirements to other software project artifacts, to display notifications of change in requirements and to trace those changes to the related project elements.

The UOOR approach is not just a theoretical proposal but has been designed for practical use and has been applied to significant applications, including a full case study detailed in a forthcoming book.

Acknowledgements

Completing this PhD thesis has been a long and challenging journey, which would not have been possible without the support, guidance and encouragement of many people.

First and foremost, I would like to express my deep gratitude to my supervisor, Sophie Ebersold, for her unwavering dedication, insightful feedback and constant support. Thank you for devoting so much of your time to reviewing and discussing my work and for standing by me in all circumstances.

I am equally thankful to my co-supervisor, Bertrand Meyer, for teaching me the principles of rigorous research and the art of time management. Your high standards, constructive critique and guiding supervision have shaped my academic journey. Thank you for pushing me to strive for excellence.

I am deeply grateful to Jean-Michel Bruel for the countless discussions, insightful feedback and leadership in the work on the companion book. Your influence has been invaluable to my growth as a researcher.

I would like to thank Zakaria Hackm for his work on the implementation of the Traceability tool, Ilya Shimchik for participating in the Roborace requirements elicitation activities, Sophie Ebersold for organizing the study at the University of Toulouse, Thuy Nguyen for discussions on Roborace requirements, Imen Sayar for work on requirements concepts and Florian Galinier for discussions on the SIRCOD tool. Your contributions have significantly enriched this research.

A special thanks to Andrey Sadovykh and Nikolay Shilov for their genuine interest in my work, their thoughtful questions, and their consistent feedback and support. Your encouragement has been a great source of motivation.

I would like to thank the members of my thesis jury for dedicating their time to read and evaluate my work. Your feedback has been invaluable in refining this research.

Finally, I owe an immeasurable debt of gratitude to my family for their unconditional love and support. I am especially grateful to my husband, Sasha, for his patience, understanding, and unwavering belief in my success. This achievement would not have been possible without you by my side.

Contents

I Introduction to the problem 1

1 Introduction 3
1.1 The challenges of requirements specification 3
1.2 Purpose of the thesis 4
1.3 Overview of the contributions 4
1.4 Structure 5

2 Requirements engineering terminology 7
2.1 Studied approaches 7
2.2 What are requirements? 9
2.3 Requirements vs specifications 11
2.4 Requirements dimensions 12
2.4.1 Goals 12
2.4.2 Environment 13
2.4.3 System 15
2.4.4 Project 17
2.5 The role of scenarios 18
2.6 Conclusion 19

3 Related work 21
3.1 Qualities of "good" requirements 21
3.1.1 Verifiability 21
3.1.2 Unambiguity 22
3.1.3 Traceability 22
3.2 Characteristics of a usable approach to requirements 22
3.2.1 Explicit methodology 22
3.2.2 Amount of training required 23
3.2.3 Tool support 23
3.2.4 Requirement artifacts reusability 23
3.2.5 Seamlessness 23
3.3 Overview of the studied approaches 24
3.4 Natural language-based approaches 25
3.5 Scenarios 26
3.5.1 Use cases 27
3.5.2 User stories 29
3.5.3 Use case 2.0 29

3.6 UML and SysML 30
3.7 Scenario-driven approaches 31
3.7.1 Restricted Use Case Modeling 31
3.7.2 Use Case Maps 33
3.7.3 Object-Oriented Analysis and Design 33
3.8 OO-Method 34
3.9 Goal-oriented requirements engineering 35
3.10 Test-driven software development 35
3.11 Behavior-driven software development 36
3.12 Contract-based and seamless approaches 37
3.12.1 ACL/VF framework 37
3.12.2 Multirequirements 38
3.12.3 SIRCOD 38
3.12.4 Seamless Object-Oriented Requirements 39
3.12.5 PEGS 40
3.13 Conclusion 41

II The unified solution 43

4 Unified Object-Oriented approach to Requirements 45
4.1 Introduction 45
4.1.1 Targeted projects 45
4.1.2 Fundamental idea and notation 46
4.1.3 Running example 47
4.2 Theoretical and technical background 47
4.2.1 Seamless software development 47
4.2.2 Contract-based requirements 47
4.2.3 Specification drivers 49
4.2.4 Autoproof 49
4.3 Key elements of the UOOR approach 50
4.4 Eliciting and documenting requirements 52
4.5 Modeling components of the system and its environment 52
4.6 Producing OO functional specification 56
4.7 Producing an OO behavioral specification 58
4.8 From requirements to code 61
4.8.1 Refinement 61
4.8.2 Traceability links 62
4.9 System verification and requirements reuse 62
4.9.1 System verification 63
4.9.2 Requirement artifacts reusability 63
4.10 Conclusion 65

5 Project elements traceability 67
5.1 Seamless requirements traceability 67
5.2 UOOR traceability information model 68
5.3 Project elements 69

5.3.1 Natural language requirements 70
5.3.2 OO requirements 71
5.3.3 Tests 71
5.3.4 Implementation 72
5.4 Relations between project elements 72
5.4.1 Repeats 73
5.4.2 Complements 74
5.4.3 Constrains and is constrained by 74
5.4.4 Refines and generalizes 75
5.4.5 Implements and specifies 76
5.4.6 Contains and is a part of 76
5.4.7 Tests and is tested by 77
5.4.8 Validates and is validated by 77
5.4.9 Refers to 77
5.5 Relations propagation 77
5.6 Conclusion 79

6 UOOR traceability tool 81
6.1 Introduction 81
6.2 Existing Tools and Technologies 82
6.3 Traceability tool interface 85
6.4 Link management 85
6.5 Tracking changes 87
6.6 Tool limitations and evaluation 88
6.7 Conclusion 88

7 Evaluation of the UOOR approach 91
7.1 Roborace case study 91
7.1.1 Case study design 91
7.1.2 Roborace project overview 92
7.1.3 Roborace use cases 93
7.1.4 Modeling components of the system and its environment 94
7.1.5 Producing OO functional specification 97
7.1.6 Producing OO behavioral specification 99
7.1.7 Lessons learnt from the case study 102
7.1.8 Limitations 103
7.1.9 Conclusion 103
7.2 UOOR user study 105
7.2.1 Study design 105
7.2.2 Study results 106
7.2.3 Limitations and threats to validity 108
7.2.4 Discussion 108
7.2.5 Conclusion 109
7.3 Conclusion 109

III Conclusion 111

8 Conclusion 113
8.1 Summary of contributions 113
8.2 Evaluation of the UOOR approach 114
8.3 Limitations 115
8.4 Perspectives 115

List of Figures

4.1 Three dimensions of UOOR requirements. 50
4.2 Overview of the UOOR approach. 51
4.3 Autoproof output for the incorrect implementation of the class BOOK. 63

5.1 Agile traceability information model. 69
5.2 UOOR traceability information model, ... 69
5.3 Key project elements and relations. 78

6.1 Referencing requirements document in EIS from a class. 83
6.2 Referencing requirements document in EIS from a class feature. 83
6.3 Requirement in SIRCOD. 84
6.4 Documentation view. 84
6.5 Traceability tool interface. 85
6.6 Option Selection 86
6.7 Class selector 86
6.8 Feature selector. 86
6.9 Selecting Bookmark. 87
6.10 Links in the code. 87
6.11 Tracking changes. 88

7.1 Feedback on the use of UOOR. 106

List of Tables

3.1 Description of the *"Borrow a book"* use case. 28
3.2 Description of the use case slice *"Overdue checkout"*. 29
3.3 Summary of the reviewed approaches to requirements. 42

5.1 Project element types. 73
5.2 Relations between project elements and their properties. 80

7.1 A detailed description of the *"Race without obstacles"* use case. 93
7.2 Summary of responses to the question What are the reasons to use UOOR? 107
7.3 Summary of responses to the question What difficulties have you faced when applying UOOR? 107
7.4 Summary of responses to the question How applying UOOR helped to improve UML specification? 108
7.5 Summary of responses to the question "What could make the UOOR approach more usable?" 108

Acronyms

ACL Another Contract Language. 37, 38

ASM Abstract State Machines. 33

BABOK Business Analysis Body of Knowledge. 8, 12

BDD Behavior-Driven Development. 36, 41, 42

CSTR Constraint. 80

DOORS Dynamic Object-Oriented Requirements System. 25, 68, 82

EARS Easy Approach to Requirements Syntax. 25

ECU Electronic Control Unit. 92

EIS Eiffel Information System. ix, 62, 82, 83, 85

ELEM Project Element. 80

FR Functional Requirement. 80

GNSS Global Navigation Satellite System. 92

GORE Goal-Oriented Requirements Engineering. 42

GPS Global Positioning System. 92

IDE Integrated Development Environment. 38, 46, 63, 82, 83, 88, 114, 115

IM Implementation Artifact. 80

IMU Inertial Measurement Unit. 92

IREB International Requirements Engineering Board. 8, 12

LMS Library Management System. 25, 47, 53, 75

LOTOS Language of Temporal Ordering Specification. 33

NL Natural Language. 25, 26, 32, 40, 42, 62, 65, 73, 80

NLRQ Natural Language Requirement. 80

OCL Object Constraint Language. 30, 31, 34

OO Object-Oriented. 27, 33, 34, 41, 42, 45, 46, 47, 49, 51, 52, 54, 57, 59, 60, 61, 62, 63, 65, 68, 70, 71, 73, 75, 78, 105, 114, 115

OOAD Object-Oriented Analysis and Design. 24, 31, 33, 42

PEGS Project, Environment, Goals, System. 12, 25, 40, 42, 45, 52, 53

RSML Requirements-Specific Modeling Language. 4, 38

RUCM Restricted Use Case Modeling. 24, 31, 32, 42

SIRCOD The Seamless Intergration of Requirements in Code. ix, 4, 25, 38, 39, 41, 42, 45, 84

SOOR Seamless Object-Oriented Requirements. 4, 25, 39, 40, 41, 42, 45

SOORT Seamless Object-Oriented Requirement Templates. 39, 40

SRS software Requirements Specification. 21

SWEBOK Software Engineering Body of Knowledge. 8

SysML Systems Modeling Language. 24, 30, 31, 42, 82

TDD Test-Driven Software Development. 35, 36, 41, 42

UCM Use Case Maps. 33

UML Unified Modeling Language. xi, 24, 30, 31, 32, 33, 34, 42, 47, 105, 106, 107, 108, 109

UOOR Unified Object-Oriented Approach for Requirements. ix, xi, 4, 5, 45, 46, 50, 51, 52, 58, 61, 62, 63, 64, 65, 69, 75, 81, 88, 89, 91, 101, 102, 103, 105, 106, 107, 108, 109, 113, 114, 115, 116

URI Uniform Resource Identifier. 83

Part I

Introduction to the problem

Chapter 1

Introduction

1.1 The challenges of requirements specification

An increasing number of tasks – from enterprise management and reporting to operating medical devices – tends to be delegated to software. To rely on software, however, we have to be sure that it does exactly what it is supposed to do. The process of description of what the software will do and how it will perform is known as software requirements specification. This process plays a major role in software development lifecycle: according to recent study [47], requirements-related factors are among the leading reasons for software project failures. Mistakes in requirements may be very costly to fix and even lead to catastrophes [18].

A well-known example of requirements-related issue that caused several deaths is the Therac-25 incident. Therac-25 was a radiotherapy machine that caused massive overdose of radiation due to software malfunctioning. One of the causes of failure was improper reuse of the earlier machines' software and poor requirements traceability. The development based on the wrong assumption that software of the earlier machines worked correctly [14].

Software failures can also be attributed to missing or incorrect properties of the system's environment. In 2015 the Washington state prison case has been revealed: due to improper calculations, more than 3200 inmates were released too early, and went on to commit several homicides [130].

Many approaches to requirements specification have been proposed; yet the balance between requirements efforts and the quality of the obtained requirements is hard to achieve: quality requirements require certain rigor and formality, which in turn requisite substantial training for requirements engineers. Affordable approaches, on the contrary, are much easier to grasp, but leave a lot of room for deficient requirements. Requirements traceability is also a challenge: due to the necessity to manually create and maintain traceability links, requirements traceability is often perceived as an inefficient practice and is applied only in 40% of software projects [36]. Poor requirements traceability practices may lead to disaster: requirements can never be fixated at the beginning of the project, they constantly evolve and change. The ability to adapt to changes in requirements is tightly related to the traceability between

requirements and other project artifacts.

1.2 Purpose of the thesis

The need to quickly adapt to changes in requirements can be satisfied by seamless software development which ensures fast and smooth traceability between requirements and other software artifacts. In the past decade a considerable effort was made to investigate how requirements can be integrated into a seamless software development process. B. Meyer suggested the notion of Multirequirements [79], which incorporates expressing requirements in three layers: natural language, programming language and graphical. The idea was further developed in SOOR [87] and SIRCOD [38] approaches. SOOR approach relies on the notion of specification driver, a contracted routine for capturing requirements, and provides a library of reusable requirements templates based on common requirements patterns. SIRCOD suggests a tool-supported domain-specific language, RSML, which gives the ability to automatically translate requirements expressed in constrained natural language to Eiffel and provides a framework for tracing requirements from natural language documents to implementation code.

Inspired by these works, the present thesis investigates the practical application of seamless requirements: what should be the process of seamless software development from requirements to code? What tool support is required to ensure traceability between the requirements and other project artifacts? The thesis seeks to establish a proper balance between requirements effort and outcome by suggesting a usable and affordable approach that facilitates producing unambiguous, reusable, verifiable and traceable requirements.

1.3 Overview of the contributions

The thesis introduces a Unified Object-Oriented approach to Requirements (UOOR), which unifies scenarios and object-oriented techniques. It demonstrates that the approach is affordable and practically usable and that it facilitates producing unambiguous, verifiable and traceable requirements. The specific advances are the following:

- A demonstration that the concept of class is general enough to describe not only objects in a narrow sense but also scenarios such as use cases and user stories and other important artifacts such as test cases and oracles.

- A methodology that a requirements engineer may use as a guide for requirements specification process.

- A partial formal model of requirements engineering through the definition of relations and their formal mathematical properties.

- The notion of seamless requirements traceability, which relies on propagation of traceability links, themselves based on formal properties of relations between project artifacts.
- The Traceability tool, which supports requirements traceability by creating typed links between project elements, and tracking changes from requirements to the affected code elements.
- An evaluation of the approach by applying it to a significant real-world case study and conducting a controlled experiment.

1.4 Structure

The thesis consists of three parts.

Part I introduces the reader to the problem. Chapter 1 provides general introduction, presents the purpose of the thesis, overviews the contributions and presents the structure of the manuscript. Chapter 2 sets the ground for the research by answering the following questions:

- What are requirements?
- Is there a difference between requirements and specifications?
- What are requirements dimensions?
- What are the requirements types?
- What is the role of scenarios in requirements?

Chapter 3 explores the characteristics of a requirements engineering approach that are necessary to make it usable in industry; further it reviews the related works and evaluates them towards the identified criteria.

Part II presents the contributions of the thesis. Chapter 4 devises the Unified Object-Oriented approach to Requirements and provides guidance on producing UOOR specification. Chapter 5 explores the notion of seamless requirements traceability. It devises the typed relations between project elements and formal properties of such relations. Chapter 6 presents a tool, to be integrated into EiffelStudio, which facilitates managing requirements traceability links. Chapter 7 evaluates the UOOR approach. It presents the application of the UOOR approach to a significant real-world project - the Roborace. Next, it reveals the results of a controlled experiment, conducted at the University of Toulouse, and evaluates the perception of the approach.

Part III summarizes and evaluates the contributions, lists the limitations of the devised approach and highlights the perspectives for future work.

Chapter 2

Requirements engineering terminology

The thesis presents an approach to requirements specification. But what is the subject of such an approach? What are requirements? Is there a difference between requirements an specifications? And what particular aspects of requirements should be covered?

This chapter reviews requirements standards, main requirements engineering schools and their authors, requirements structure and guidelines textbooks to establish common requirements terminology. In particular, it addresses the following questions:

- What are requirements? (section 2.2).
- What is the difference between requirements and specifications? (section 2.3).
- What are requirements dimensions and what are their respective roles in requirements? (section 2.4).
- What are the requirements types? (section 2.4.3).
- What is the role of scenarios in requirements? (section 2.5).

The answers to these questions help to establish the subject of the requirements specification approach, i.e what a devised approach to requirements should address.

2.1 Studied approaches

Considering main schools and their authors, the reviewed approaches are:

- A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications* [64].
- K. Wiegers and J. Beatty. *Software Requirements* [128].

- M. Jackson and P. Zave. [137, 136, 55].
- P. A. Laplante. *Requirements engineering for software and systems* [65].
- D. Leffingwell. *Agile software requirements* [68].
- K. J. McDonald. *Beyond requirements: analysis with an agile mindset* [75].
- C. Larman. *Applying UML and patterns: an introduction to object oriented analysis and design and iterative development* [66].
- A. Cockburn. *Writing effective use cases* [28].
- J. Dick, E. Hull, K. Jackson. *Requirements engineering* [33].
- D. Bjørner. "*Domain engineering*" [17].

The studied standards and related guides are:

- *ISO Standard 29148: 2011 Systems and Software Engineering* [3].
- *ISO/IEC/IEEE Standard 29148: 2018 Systems and Software Engineering* [5].
- *ISO/IEC/IEEE 24765:2017 Systems and software engineering – Vocabulary* [4].
- *IREB Glossary* [41], and a related textbook by K. Pohl [107].
- International Institute of Business Analysis (IIBA). *A guide to the business analysis body of knowledge (BABOK)* [23].
- IEEE Computer Engineering Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK)* [114].

Textbooks covering requirements structure and guidelines studied here are:

- S. L. Pfleeger and J. M. Atlee. *Software Engineering: Theory and practice* [104] (chapter on requirements).
- I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide* [117] [62].
- B. Meyer *Handbook of Requirements and Business Analysis* [81].
- S. Robertson and J. Robertson *Mastering the requirements process: Getting requirements right* [109].

2.2 What are requirements?

Even in standards, and even for the basic concepts – such as "requirement" – some definitions are far from being ideal. For example, IEEE standard [1] defines "requirement" as

A statement that identifies a product or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability (by consumers or internal quality assurance guidelines).

Not only the definition is long and ambiguous, it is prescriptive, whereas a good definition should be descriptive. The definition states that a requirement is *"a statement ... which is unambiguous, testable or measurable ... "* Does it mean that a statement that is ambiguous cannot be a requirement? In fact it is not true: natural language requirements are often ambiguous, yet they *are* requirements (maybe not the perfect ones). Next, why only these three characteristics appear in the definition? Shouldn't a requirement be necessary and appropriate, as stated in IEEE standard [5]? In fact it is neither justified, nor appropriate to list quality factors in the definition: it distracts the reader from its essence and gives the impression that some quality factors (those mentioned in the definition) are more important than others (the omitted ones), which is incorrect.

Other definitions of a "requirement" in the literature differ considerably from each other:

1. *"A prescriptive statement to be enforced by the software-to-be, possibly in cooperation with other system components, and formulated in terms of environmental phenomena"* [64].
2. *"A desired relationship among phenomena of the environment of a system, to be brought about by the hardware/software machine that will be constructed and installed in the environment"* [55].
3. *"A statement of a customer need or objective, or of a condition or capability that a product must possess to satisfy such a need or objective"* [128].
4. *"A usable representation of a need"* [23].
5. *"A statement which translates or expresses a need and its associated constraints and conditions"* [5].
6. *"A statement that specifies (in part) how a goal should be accomplished by a proposed system"* [65].
7. *"A statement of a system service or constraint"* [62].
8. *"A desired capability or behavior that a software system should possess"* [68].

9. *"Capabilities and conditions to which the system – and more broadly, the project – must conform"* [66].
10. *"A capability or property that a system shall have"* [41].
11. *"A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents"* [4].
12. *"An expression of desired behavior"* [104].
13. *"A description of how the system should behave, or of a system property or attribute"* [117].
14. *"Something that the product must do, or a property that the product must have, that is needed or wanted by the stakeholders"* [109].
15. *"A property that must be exhibited by something in order to solve some problem in the real world"* [114].
16. *"Descriptions of how the system should behave, application domain information, constraints on the system's operations, or specifications of a system property or attribute"* [62].

The first thing, which is common in most of these definitions is that a requirement is a statement/representation/expression/description. In other words, requirements engineering concerns not the needs, capabilities and properties, but the way they are expressed in the requirements documentation.

Further analysis of the definitions gets more complicated, as some definitions are extremely vague. What exactly do requirements express *"Desired behavior"*, as stated in definition (12)? Behavior of what/whom? A *need*, as claims the definition (4)? Whose need? Besides, requirements do not necessarily express needs – they can express constraints of limitations. *"A property that must be exhibited by something in order to solve some problem in the real world"*? This definition is way too hesitant, as "exhibited by something" sounds too vague for an IT project.

The definitions may be further grouped based on their similarity. Definitions (1) and (2) highlight the role of the environment in requirements specification: requirements must be expressed in terms of environmental phenomena (not in terms of internal phenomena of the machine). These definitions distinguish requirements from specifications: requirements are formulated in terms of environmental phenomena whereas specifications (or software requirements) are *"enforced solely by the software-to-be and formulated only in terms of phenomena shared between the software and the environment"* [64].

Definitions (3), (4), (5), (6) claim that a requirement is an expression of a need or an objective. Whereas needs and objectives are the primary sources of requirements, some of the requirements come from constraints or regulations, rather than from the needs.

Definitions (7) – (14), (16) state that requirements express desired capabilities or behavior. Definition (12) *"An expression of desired behavior"* [104] could be a valid requirement definition if we considered a wide range of subjects (behavior of the system, development team, environment entities and so on) and if the term "behavior" was used in a very broad sense, covering not only functional, but also non-functional characteristics of behavior. However, this is not what first comes to mind when reading this definition, so it is at the very least ambiguous. If we read the definition as "an expression of desired behavior of a system", it turns just wrong, since requirements include many elements that do not relate to system's behavior, such as design constraints [6]. Similar reasoning applies to the definition (8).

Definitions (7), (10), (11), (13) can be summarized into "system capabilities (behaviors) and constraints (properties)". Such a definition indeed covers most of the requirements elements. Definitions (9) and (16) highlight that requirements should cover project constraints [66] and domain information [62]. We will discuss the role of project dimension in requirements in more details in section 2.4.4.

To summarize the above reasoning, we suggest the following definition of the term "requirements":

Definition 2.1. Requirements are statements describing capabilities and constraints to which the system must conform.

2.3 Requirements vs specifications

The term "specification" is often used as a synonym for the term "requirements" [64], whereas some authors distinguish the two notions [101, 53]. According to Jackson [53], requirements are formulated only in terms of environment phenomena, i.e. in terms of things, observable in the surrounding world; specifications are descriptions of the behaviors that a machine should produce in order to meet the requirements, i.e. they are the interfaces between requirements and programs. Specifications are thus refinements of the requirements, yet not necessarily they can be derived directly from the requirements: some requirements are constrained by the domain knowledge, which has to be taken into account when producing specifications. Similarly, van Lamsweerde [64] distinguishes system requirements (formulated only in terms of environment phenomena) and software requirements (formulated in terms of phenomena shared between the system and its environment). Software requirements are thus a subset of system requirements.

In the context of this thesis we do not distinguish requirements and specifications, so the term "requirement" encompasses both meanings.

2.4 Requirements dimensions

Requirements engineering focuses on defining the objectives of a system, independently of their realization. Many approaches have been proposed to tackle the difficult tasks involved in this effort; among some of the best known are contributions by Michael Jackson and Pamela Zave [137, 55, 136], Axel van Lamsweerde [64], and others we consider in this state of the art.

Generally speaking, requirements describe what a system should do (functional requirements) and how it should do it (non-functional requirements). However, this point of view is too simplistic. In their seminal work Jackson and Zave established the key role that environment plays in formulating the requirements [137], whereas Axel van Lamsweerde [64] emphasized the role of goals in requirements. Meyer et al. attempted to provide a taxonomy of requirements elements [83], which subsequently formed the basis of the PEGS approach to requirements. According to PEGS, the four requirements dimensions are Goals, Environment, System and Project.

This section reviews requirements engineering literature (textbooks, research articles and standards) to study the role that each of the four dimensions plays in requirements.

2.4.1 Goals

The term "goal" refers to high-level needs or objectives. The term has slightly different meaning in different approaches. According to [81, 65, 68, 109, 64] goals are needs or objectives of the **organisation**, that the system must address. According to IREB [41, 107] and BABOK [23], goals describe intentions of **stakeholders**. According to Pfleeger and Atlee [104, 114], goals express the purpose of the proposed software, from the **customer**'s perspective.

The importance of the role that goals play in understanding requirements is highlighted by many authors:

- Goals, unlike other requirements elements, address the widest audience of the project (all stakeholders) [81].
- Goal establish the key benefits that system implementation must bring and thus establish the priorities to consider when taking decisions in the course of the project [81].
- Goals are high-level objectives, whereas requirements specify how the goals should be fulfilled by the system [65].
- System objectives are goals on the level of the entire system, which can be further refined to lower-level goals, software requirements and environment assumptions. A requirement is a fine-grained goal *"that is under the responsibility of a single agent of the software-to-be"* [64].

- Goals are high-level requirements, which help to identify requirements that will provide the positive contribution towards the goals' achievement [109].
- Goals serve as the basis for system requirements [107].
- Goals are used to validate that all requirements are necessary [104].
- Goals define the vision of the system and the scope of the project. Requirements must be aligned with the goals [128].

To summarize the discussion, we suggest the following definition of the term "goals":

Definition 2.2. Goals are high-level objectives of the organisation that the system must address.

In other words, we consider goals to be the *source* of requirements, rather than *requirements per se*.

2.4.2 Environment

In the late 1980 the desire to reuse the knowledge of the system's environment in production of new systems gave rise to domain engineering [11]. Dines Bjørner [17] studied the application of domain engineering to requirements and emphasized that significant part of requirements can be derived from domain descriptions. The term "domain", although similar to the term "environment", has a slightly different meaning. *Environment* refers to the world phenomena directly related to the particular system and may include phenomena outside the system's domain (the term "application domain", introduced by Michael Jackson in his late works [53] is also used in the same meaning). *Domain* refers to a universe of a discourse (such as library management or traffic control) and encompasses phenomena that are related to many systems in that sphere [17, 128, 64, 66, 23, 75].

To distil the meaning of the term environment, let us analyze the definitions of the term "Environment", provided in the studied literature:

1. *"Other automated systems which are interfaced to [the system] and business processes which may use the system"* [117].
2. *"External entities, such as users, hardware devices, and other systems"*[128].
3. *"The part of the world with which the machine will interact, in which the effects of the machine will be observed and evaluated"* [54].

4. Components pertaining to the machine's surrounding world, such as people or business units, physical devices, legacy or foreign software components with which the software-to-be needs to interact [64].

5. *"Anything affecting a subject system or affected by a subject system through interactions with it, or anything sharing an interpretation of interactions with a subject system"* [2].

6. *"The circumstances, objects, and conditions that will influence the completed system"* [6].

7. *"The context in which the system will function, outside the system itself"* [73].

8. *"The set of entities (such as people, organizations, devices and other material objects, regulations, and other systems) external to the project and system but with the potential to affect the goals, project or system or to be affected by them"* [81].

9. The surrounding world, its laws and regulations [65]. *"Non-functional requirements are requirements that are imposed by the environment in which the system is to operate"*[1]

From the above definitions (1)-(8), we can extract the key characteristics of the environment:

- It is part of the world.
- It is external to the system.
- It interacts with the system or affects it in some way.

To summarize the discussion, we suggest the following definition of the term "environment":

> **Definition 2.3. Environment** is the part of the world, external to the system, that affects or can be affected by the system.

Environment properties relate to requirements in the following way:

- Requirements properties are not requirements themselves, but they serve as a rich source of requirements [128].

[1]This last statement is very strong. In fact, not necessarily the non-functional requirements are imposed by the environment, and not necessarily only non-functional requirements can be imposed by the environment. For example, some non-functional requirements are derived from goals or stakeholders' decisions, rather than from environmental constraints. At the same time, some functional requirements, such as "the system shall calculate the dosage based on the patient's weight" can be imposed by the environment.

- Domain properties and assumptions are required to properly translate system requirements (formulated in terms of environment phenomena) to software requirements (formulated in terms of phenomena shared between environment and the system) [64, 55]. Satisfaction of software requirements (specifications) denoted as S together with domain properties (denoted as E) must imply satisfaction of system requirements (denoted as R): $S, E \models R$.

- Environment imposes non-functional requirements [65].

- Domain rules express the constraints on how the domain works. They are not requirements, but can clarify incomplete or ambiguous requirements [66].

To summarize, we suggest the following definition of environment properties:

Definition 2.4. Environment properties are statements related to the system's environment that affect the system in the following ways:

- They express directly the properties of the components of the environment.

- They serve as a source of requirements.

- They serve as input in requirements refinement.

The role of environment in requirements will be further explicated in section 4.5.

2.4.3 System

Requirements pertaining to the system are at the heart of any requirements approach. What differs though is what types of system requirements the approach aims to cover. The present section establishes the classification of system requirements and defines the major categories of requirements.

There are several views on how one can categorize requirements:

- All requirements on the system, that are not functional, are non-functional [64, 128, 75, 66, 23, 114].

- Functional requirements, non-functional requirements, and domain requirements (requirements derived from the application domain) [65].

- Functional requirements, non-functional requirements, and design constraints, where design constraints are technical, business or contractual restrictions on system design or development process [68, 107, 41].

- Functional requirements, interface requirements, process requirements (requirements imposed by laws and regulations), quality (or non-functional) requirements, usability requirements, human factors requirements [5]. This standard also mentions design constraints, defined as *"constraints on the system design imposed by external standards, regulatory requirements or project limitations"*.

- Functional requirements, quality (nonfunctional) requirements, design constraints (design decisions that restrict the set of solutions), process constraints (restrictions on the development process) [104].

The term "functional requirements" is the least controversial one: all of the approaches agree that a functional requirement specifies a function that a system shall perform. There is no such accordance towards non-functional requirements.

Several authors highlight that the boundary between functional and non-functional requirements is not solid [64, 128]. For example, safety and security requirements may fall into both categories simultaneously, as in the example *"the safety injection signal shall be on whenever there is a loss of coolant except during normal start-up or cool down"* [64].

In some approaches, the terms "non-functional requirements" and "quality requirements" are synonyms [5, 104, 107]. In other approaches, quality requirements (sometimes also called "quality attributes") are a subset of non-functional requirements [128, 64, 65, 66]. Some approaches do not mention the term "quality requirements" (or "quality attributes") and thus do not allocate quality requirements to a separate category [81, 75, 109]. Finally, some approaches, such as Cockburn's [28] or Jaskson's [55], focus solely on functional requirements and thus do not discuss non-functional requirements.

If we take aside the terminology aspect, there is agreement between the majority of the authors that quality requirements are non-functional requirements (or some part of non-functional requirements). What else should be specified as non-functional requirements is a more controversial ground.

According to Wiegers, non-functional requirements include quality attributes, external interfaces, and design and implementation constraints [128]. Laplante lists five categories of non-functional requirements: quality, design, economic (constraints on development cost), operating, and political/cultural (constraints from laws and standards) [65]. Van Lamsweerde [64] mentions quality requirements, compliance requirements (constraints from laws and regulations), architectural requirements (distribution and installation constraints), and development requirements (constraints on the development process). In Volere [109] there are eight kinds of non-functional requirements: products appearance, usability, performance, the operating environment of the product, maintainability, security, cultural and political, and legal.

Based on the analysis of the studied literature, we can identify three conceptually different types of requirements: functional requirements, non-functional requirements and constraints. The reason that constraints are allocated to a separate category is that they represent something in-between functional and

non-functional requirements. On one hand, constraints restrict system functions, so they are somewhat functional. On the other hand, constraint describe the *how*, not the *what*, so in this way they are non-functional.

We suggest the following definitions:

Definition 2.5. Functional requirement is a description of *what* the system should do.

The key property of a functional requirement is that it describes a function or a service that the system must provide.

Definition 2.6. Non-functional requirement is a description of *how* the system will perform.

Note that requirement is functional if it describes a function or a service that the system must provide, so the requirement *"the safety injection signal shall be on whenever there is a loss of coolant except during normal start-up or cool down"* is considered to be functional even though it can also be considered a safety (non-functional) requirement.

Although some of the approaches do not allocate constraints to a separate category of requirements [64, 65], most of the authors admit the difference between constraints and functional requirements [137, 136, 55, 128, 68, 66, 107, 33, 104, 117]. The difference between constraints and non-functional requirements is more subtle: some authors define constraints in a broad way, including restricting not only the system's functions behavior, but also more general properties [64, 128].

Nevertheless, it seems more natural to allocate such properties to non-functional requirements and define as constraints requirements that combine functional and non-functional properties:

Definition 2.7. Constraint is a property that restricts the system function behavior.

2.4.4 Project

There is no consensus between the authors whether project requirements should be included into requirements specification. Some of the authors do not discuss project requirements [65, 28]. At the same time, many authors highlight project aspects that must be considered in requirements specification.

According to Zave [136], estimating costs, risks and schedules are the tasks of requirements engineers.

Wiegers [128] provides a detailed list of project requirements, which concern the project itself rather than the product to be built. According to Wiegers,

project's success depends on success of translating software requirements to project plans and software designs. In particular, estimates, project plans and schedules are based on requirements. Those estimates have to be updated when requirements change. Wiegers also distinguishes project-level constraints, such as schedule, staff and budget limitations. Nevertheless, project requirements, although a shared responsibility of the business analyst and a project manager, should not be included into a System Requirement Specification and best fit into the project management plan.

Van Lamsweerde [64] highlights that the project type influences the requirements engineering process. Despite the fact that the author does not distinguish project requirements as a part of requirements document, he mentions some of project requirements, such as development requirements (constraints on the development process), as a part of non-functional requirements.

In the Volere approach [109], project tasks and project risks are included into requirements document. The "Task" section of the Requirements Specification Template is devoted to the planning of the project aiming at delivering the final product. It lists the tasks to be done and the estimates of required time and resources for each task. The "Risks" section is dedicated to identifying project risks based on the requirements knowledge as input. Each risk is accompanied with the estimation of its probability.

Pfleeger and Atlee [104] identify **process constraints** as the restrictions on the techniques or resources that can be used for system construction. In particular, process constraints cover the characteristics of the required personnel and other resources, documentation requirements, and standards restricting system construction process.

Laplante [65] attributes to nonfunctional requirements "operating constraints". These include staff availability and skills, and the availability of the system for maintenance - which are essentially project constraints.

According to the IEEE standard [5], project constraints (which are *"constraints to performing the project within cost and schedule"*) should be included to requirements documentation.

To summarize, we suggest the following definition of the term "project requirements":

Definition 2.8. Project requirements are requirements related to a *process* of software development.

Project requirements, however, not necessarily should be a part of requirements specification and can be included instead into a project plan.

2.5 The role of scenarios

The role of scenarios in different requirements approaches differs significantly: whereas some authors argue that scenarios are examples of system behavior, others state that scenarios are requirements, not merely examples.

McDonald uses the term “examples” to denote *“an approach to defining the behavior of a system using realistic examples instead of abstract statements”*[75]. Similarly, van Lamsweerde describes scenarios as *“concrete examples of how things are running in the system-as-is, or how they should be running in the system-to-be.”* These authors highlight that scenarios lack abstraction and thus can be considered as examples of system behavior, rather than its specification.

Agile approaches, on the contrary, consider scenarios to be the requirements. Cockburn claims that scenarios are *“not all of the requirements [...], but they are **all** of the behavioral requirements”.* [28]. Leffingwell is even more radical: he claims that user stories are “*the general-purpose agile substitute for what traditionally has been referred to as software requirements*” [68].

To summarize, there is no agreement in the literature with respect to the role of scenario in requirements: some authors argue that scenarios are requirements, whereas others state that scenarios are solely examples of system’s behavior. There is a way, however, to unify those views: in this unified approach scenarios serve as behavioral requirements, defining the admissible sequences of operations; at the same time, they serve as the source of more abstract functional requirements and constraints.

The discussion on the role of scenarios will be continued in section 4.7.

2.6 Conclusion

This chapter established the requirements engineering terminology relevant for the present thesis, based on critical review of the existing literature. It defined the notions of requirements and environment; explored the difference between requirements and specifications; explored requirements classification and established the notions of functional requirements, non-functional requirements and constraints; explored the four requirements dimensions: goals, environment, system and project – and their role in requirements specification; discussed the controversial role of scenarios in system’s requirements.

The chapter answers the question: “What a devised approach to requirements should address?” Summarizing the above discussion, we can formulate the following research questions (manuscript sections covering the respective questions appear in parentheses):

- How functional requirements should be specified? (Section 4.6)
- How constraints should be specified? (Section 4.6)
- How environment properties should be specified? (Section 4.5)
- What is the role of scenarios in requirements specification? (Section 4.7)

The next chapter 3 will continue exploration of the state of the art by defining the characteristics that a requirements approach should have and evaluating the existing approaches against these criteria.

Chapter 3

Related work

This thesis presents a practical usable approach to requirements specification. To have the potential to be adopted in industry, a requirements engineering approach must cover two aspects: it should facilitate producing quality requirements; and it should be clear and easy to learn and apply.

This chapter explores the characteristics of a requirements engineering approach that are necessary to make it usable in industry; further it reviews the related works and evaluates them towards the identified criteria.

Section 3.1 explores some of the qualities of “good” requirements. Section 3.2 investigates the characteristics of the requirements approach that make it usable. Section 3.3 summarizes the criteria for comparing the related works and provides an overview of the studied approaches. Sections 3.4 - 3.12 provide the review of the related works. Section 3.13 provides a table summarising the characteristics of the reviewed approaches.

3.1 Qualities of “good” requirements

According to the IEEE international standard [6], a good SRS should be correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable, and traceable. Nevertheless, it is hard to achieve some of these characteristics if the requirements are documented in informal natural language, whereas according to the industrial survey, conducted by Fricker et al. [36] only 6% of projects utilize formal notations. This section explores the characteristics of “good” requirements that are hard to achieve with natural-language-based specifications. Sections 3.4 - 3.12 will provide the evaluation of the studied requirements approaches towards achieving these qualities.

3.1.1 Verifiability

A requirement is verifiable if and only if it has the means to prove that the system satisfies the specified requirement [6]. Although the notion of verifiability is defined in IEEE international standard, not much of guidance is provided on how to achieve this property apart from the statement that *verifiability is enhanced when the requirement is measurable* [6].

3.1.2 Unambiguity

Ambiguity refers to possible different interpretations of one and the same requirement. Natural language, which is a predominant way of specifying requirements, is an innate source of ambiguity [76]. The only way to avoid the ambiguity is to introduce some level of formality: to constrain the natural language or to utilize formal or semi-formal methods and notations. At the same time, formal methods are not widely used. The key reasons for that are that, on one hand, formal methods require learning related notation and modeling techniques which often requires strong mathematical background, and on the other hand, formal specifications are not readable which makes difficult sharing the information and ensuring specification accuracy. In view of this, we can clearly identify the need for the method of producing unambiguous requirements specifications.

3.1.3 Traceability

Requirements traceability is the ability to follow both the sources and consequences of requirements [81]. Requirements traceability ensures that the impact of changes in requirements specification is easily localizable in the code, which significantly decreases the time and costs of assessing the impact of changes and implementing the changes. Traceability relations can also be used to assist the process of verifying that system meets its requirements.

Even in regulated or safety critical domains, such as healthcare or military, traceability links are often created at the very end of the process and contain many deficiencies, such as incomplete, missing or erroneous traceability data [27]. In less formal projects, traceability if often perceived as a "made up problem" or an "unnecessary evil" [27]. Consequently, only 40% of software projects practice requirements traceability [36].

At the same time, the projects with missing traceability links are vulnerable to change and evolution: it can be tremendously hard to identify all the system elements where the changes should be introduced.

3.2 Characteristics of a usable approach to requirements

A requirements approach is a strategy of developing and managing requirements for a project together with supporting guides and tools. In order to be used in industry, an approach must meet certain criteria, which will be explored in further subsections.

3.2.1 Explicit methodology

There exist many approaches to requirements, and the most mature and used ones are covered in textbooks, video tutorials and university curricula. In order

to switch from such an approach to a new one, a person would expect to find in the supporting materials the detailed guidance on how to apply the approach. It was demonstrated that application of a new requirements methodology is more efficient when requirements engineers use the procedural knowledge of an approach (i.e. the examples of application of the methodology) [123].

3.2.2 Amount of training required

Requirements specification approaches place different demands on the qualification of requirements engineers. For example, natural language requirements do not require specific knowledge, whereas formal methods require significant training. Too high prerequisites on requirements engineers background may be a barrier for the method's adoption [32].

3.2.3 Tool support

Relying on tool-supported requirements engineering facilitates capturing, tracing, analyzing and managing changes to requirements [49]. According to state of practice surveys [61, 71, 34], 40 to 67% of projects use requirements specification and management tools, with a tendency that the access to such tools is better in large multi-national corporations [71].

Tool support is essential for ensuring requirements traceability and managing changes in requirements: manual performance of such tasks is tedious and decreases project's agility.

3.2.4 Requirement artifacts reusability

Reusability is a degree to which an asset can be used in more than one system [4]. Reusability is widely adopted in software engineering: libraries of reusable components became an integral part of programming languages. Nevertheless, requirements reuse has not reached that level yet.

The study of Irshad et al. [51] reviews the approaches to requirements reuse. The majority of the approaches (57%) suggest reusing requirements in a textual form, or the form of requirements reuse is undefined. Other forms of requirements reuse include templates, use cases, modeling language-based artifacts, formal models, and features. In practice, the level of requirements reuse remains low and mostly relies on copying and modifying natural language requirements from previous projects [100]. Improper reuse or requirements is dangerous: simple copy-pasting requirements from preceding projects may lead to catastrophes as was the case in Therac-25 project [14].

3.2.5 Seamlessness

Software development may rely on several different notations, such as natural language, modelling language, formal language, and programming language. The process of switching from one notation to the other, when not seamless,

is prone to errors. Seamless software development uses a uniform method and notation throughout all activities, such as problem modeling and analysis, design, implementation and maintenance [78]. Seamless software development facilitates traceability between requirements and other software artifacts.

3.3 Overview of the studied approaches

To have the potential to be adopted in industry, a requirements engineering approach must cover two aspects: on one hand, it should facilitate producing quality requirements; on the other hand, it should be clear and easy to learn and apply. The details of both aspects have been discovered in sections 3.1 and 3.2; uniting them we get a list of criteria for comparing the existing requirements approaches:

- Explicit methodology.
- Amount of training required.
- Tool support.
- Requirement artifacts reusability.
- Requirements verifiability.
- Requirements unambiguity.
- Traceability support.
- Seamlessness.

Sections 3.4 - 3.12 provide the review of requirements engineering approaches with respect to the criteria listed above. Requirements approaches vary from completely informal natural language texts to rigorous specifications in formal notations [52], [8]. Formal methods fall outside of the review, since they target a specific group of software projects (mission-critical systems) and require substantial training. Among informal and semi-formal approaches the review covers the following ones:

- Natural language-based approaches [128, 74, 127].
- Scenarios (use cases [58], user stories [29], use cases 2.0 [56, 57]).
- UML [30] and SysML [97].
- Scenario-driven approaches (RUCM [133, 134], Use Case Maps [9, 21], OOAD [19]).
- OO-Method [98, 102].
- Goal-oriented approaches [63, 132].

- Test-driven software development [15].
- Behavior-driven software development [120, 113].
- Contract-based and seamless approaches (ACL/VF [12, 13], Multirequirements [79], SIRCOD [38], SOOR [87], PEGS [81]).

3.4 Natural language-based approaches

Natural language (NL) requirements are requirements formulated in the form of unrestricted NL text, or NL text, restricted in a certain way. Detailed guidance to producing unrestricted NL requirements is provided by Wiegers [128]. The ambiguity, inherent to the unrestricted natural language, gave rise to multiple restricted natural language-based approaches, which rely on patterns or controlled natural languages.

A well-known example of pattern-based requirements approach is EARS (Easy Approach to Requirements Syntax) [74]. In EARS, a small set of syntactic rules defines the structure of requirements statements. Every requirement belongs to one of five types, described as follows:

1. Ubiquitous requirements - do not have precondition or trigger (The <system name> shall <system response>). *"The LMS shall have its interface in English"*.
2. Event-driven requirements - are initiated by a trigger event (WHEN <optional preconditions> <trigger> the <system name> shall <system response>). *"When a patron logs in, the LMS shall display a list of his reserved and checked out books"*.
3. Unwanted behaviors - are requirements covering handling undesirable situations (IF <optional preconditions> <trigger>, THEN the <system name> shall <system response>). *"If a patron receives three penalties, then the LMS shall suspend his subscription for one year"*.
4. State-driven requirements - are active when a system is in a given state (WHILE <in a specific state> the <system name> shall <system response>). *"While a patron has a maximum allowed number of holds, the LMS shall not allow him to place hold on books"*.
5. Optional features - are requirements that are applicable only when the system includes a particular feature (WHERE <feature is included> the <system name> shall <system response>) *"Where the book sample is available, the LMS shall allow the patron to download the sample"* [74].

A wide variety of tools, such as DOORS [48], Enterprise Architect [119], Jama Connect [116], support NL-based approaches. Such tools require manual creation of traceability links and often do not support direct traceability from requirements to source code elements.

Relax [127] is a requirements language for self-adapting systems, which semantics is defined in a temporal fuzzy logic. The language can model the relations between environment properties and properties, monitored by the system. RELAX operators, used for expressing temporal properties, include standard operators of temporal logic enriched with operators that support expressing uncertainty. The requirements can be relaxed with an alternative or RELAXation modifier (such as EVENTUALLY), so that at run-time a requirement can be temporarily unsatisfied. Consider the requirement "*The synchronization process SHALL be initiated when a book is checked out*". In practice, there might be circumstances that such a requirement cannot be satisfied. The RELAXed requirement can handle this uncertainty: "*The synchronization process SHALL be initiated AS EARLY AS POSSIBLE AFTER a book is checked out*". The approach is supported by Eclipse-based editors COOL RELAX and RelaxEditor [40, 35], yet at the time of writing these tools are not available.

Evaluation of the NL-based approaches:

- **Explicit methodology:** There exist methodologies supporting NL-based requirements.
- **Amount of training required:** NL requirements do not require substantial training.
- **Tool support:** There exist tools supporting management of NL requirements, such as [48].
- **Requirement artifacts reusability:** NL requirements are reused by copy-paste.
- **Requirements verifiability:** Some extent of verifiability is achieved by formulating measurable requirements in NL.
- **Requirements unambiguity:** Requirements in NL are inherently ambiguous. The ambiguity can be partially eliminated by constraining the natural language.
- **Traceability support:** Traceability of NL requirements is supported by manually creating traceability links.
- **Seamlessness:** Software development based on NL requirements is not seamless.

3.5 Scenarios

Scenarios are used to capture requirements by describing a specific use of the system-to-be that is of a value to its user. Scenarios capture examples of interaction with the system, looking at the system as a "black box" (i.e. from

the outside). Subsections 3.5.1 - 3.5.3 review differed types of scenarios, used in requirements: use cases, user stories and use cases 2.0. The section is concluded with the assessment of scenarios against the evaluation criteria.

3.5.1 Use cases

Use cases have become one of the major formalisms for expressing requirements thanks to Ivar Jacobsons work and his 1992 book [58]. Their spread happened at about the same time as the spread of OO methods for programming and design. A use case describes a unit of requirements in the form of a possible scenario of user interaction with the system.

There are several use case notations; the examples in this article will rely on a notation due to Cockburn [28]. Table 3.1 presents an example, related to a library system. It specifies a scenario whereby a library user borrows a book from the library, with such steps as placing the book on hold, then checking it out, and returning it by the specified deadline.

The core part of the use case is the **main success scenario**, giving the sequence of steps of the use case. Additional possibilities of the notation include:

- Steps consisting of several sub-steps, similar to calling routines in programming.
- Conditional steps, in which the use case follows either of two sub-scenarios depending on the outcome of a certain condition.

Such mechanisms suggest a strong analogy between a use case and an *algorithm* or a program. The two concepts have differences: a use case (and other kinds of scenarios such as user stories, reviewed next) describes the interaction between a human actor and the system, whereas an algorithm or program is meant to be carried out by a computer.

The **level** entry characterizes the level of abstraction. A use case can describe a process at many levels, from the highest (a birds eye view of an overall business process, meant to be complemented by further use cases for the details) down to the detailed descriptions of the systems actual operation.

A **precondition** is a limiting condition governing the applicability of the use case.

A **trigger** is an event that starts the use case.

The last two concepts differ in the following way: a precondition is a condition that must hold for the use case to be applicable, but does not by itself cause its execution; a trigger does. (The precondition is necessary, the trigger is sufficient.)

The **success guarantee** characterizes the state resulting from successful execution of the use case, for the "main success scenario". In the terminology of mathematical software verification, from which the term "precondition" is borrowed, it would be called a "postcondition".

An **extension** describes a departure from the "main success scenario". Extensions serve two separate purposes:

Name	Borrow_a_booksoort
Scope	System
Level	Business summary
Primary actor	Patron
Context of use	The patron wants to check out a book
Preconditions	The book is available in the library's catalog
Trigger	The patron finds in a library catalogue the book he wants to borrow and requests the system to place a hold on this book
Main success scenario	* The system changes the book status to on_hold * The patron checks out the book * The patron returns the book
Success guarantee	The patron has borrowed the book and returned it within the checkout duration.
Extensions	A. The book is not available *The system denies placing hold on the book B. The hold expires due to exceeding maximum hold duration. * The system changes the hold status to "expired" and the book becomes available C. The patron cancels the hold * The book status changes to available D. The patron does not return the book within the maximum check out duration * The book status changes to overdue * The patron returns the book
Stakeholders and interests	Patron (borrows a book) Library personnel (enforces adherence to library policies)

Table 3.1: Description of the *"Borrow a book"* use case.

- An extension can specify an alternative to the main success scenario, to be applied when that scenario hits a condition that prevents it from proceeding normally.

- Extensions also support the reuse of elements common to several use cases, which can then be divided into a base use case, covering the common elements, and specific extensions.

The use case lists, towards the beginning, the **main actor** responsible for carrying out instances of the scenario. It concludes with a **list of stakeholders**: others who may be affected [94].

3.5.2 User stories

User stories [29], similarly to use cases, express a typical interaction with the system, but at a much smaller level of granularity. They play an important role in requirements in agile methods, which promote incremental program construction: the basic agile development iteration involves a developer picking the next item from a list of functions to be implemented, implementing it, and moving on to the next one. A use case is generally too complex for such atomic units of development.

The standard format for a user story includes three elements:

- A role (As a...), corresponding to the main actor of a use case.
- A desired function (I want to...), part of the systems behavior.
- A business purpose (so that...), corresponding to one of the goals of a system

An example user story in a library system is: *As a patron, I want to check out a book so that I can read it at home.*

3.5.3 Use case 2.0

Use Case 2.0 [56, 57], a revision of the original use case methodology, combines ideas of use cases and user stories through the notion of use case slice. A use case slice is a selected part of a use case, which is also a unit of testing (usable in a test-driven design methodology). Breaking down a use case into slices makes it possible to consider it at different phases of development and to build the system incrementally.

As an example, a slice of the use case *Borrow_a_book* is *Overdue_checkout* (table 3.2). It outlines its narratives as a set of flows: the basic flow of *Borrow_a_book* and one of its extensions (Alternative Flow D), "*The patron does not return the book within the maximum check out duration.*"

Name	Overdue_checkout
State	Scoped
Priority	Should
Flows	BF: "*Borrow a book*" + AF D: "*The patron does not return the book within the maximum check-out duration*"
Tests	The checkout duration exceeds the limit
Estimate	3/20

Table 3.2: Description of the use case slice *"Overdue checkout"*.

The standard format of a use case slice includes:

- A name, used to track it through the development cycle.
- A state, with possible values scoped, prepared, analyzed, implemented or verified.

- A priority, expressed by the MoSCoW acronym: Must, Should, Could, Would.
- References: flows and tests.
- An estimate of the work needed to implement the slice.

Evaluation of scenarios (use cases, user stories and use cases 2.0):

- **Explicit methodology:** Scenarios are not requirements methodology, yet substantial material is available to support scenario-based specification.
- **Amount of training required:** Using scenarios does not require substantial training.
- **Tool support:** Tools are not required for formulating scenarios, yet a variety of tools exist to support scenarios visualisation.
- **Requirement artifacts reusability:** [99] provides a set of use case patterns and blueprints. The patterns are sketchy and do not provide a framework for reuse in its classical meaning (as defined in section 3.2.4).
- **Requirements verifiability:** No specific means for verifiability are provided.
- **Requirements unambiguity:** As formulated in natural language, scenarios are inherently ambiguous.
- **Traceability support:** There is no traceability support.
- **Seamlessness:** Software development is not seamless in scenario-based approaches.

3.6 UML and SysML

UML [30] is a unified notation used for system analysis and design. UML relies on a set of diagrams, such as use-case diagram, class diagram and sequence diagram. It is possible in UML to associate contracts with individual operations through natural language or the Object Constraint Language (OCL) notation. SysML [97], an extended profile of UML, treats requirements as first class entities, establishing direct links between requirements and other software artifacts (such as tests). Weilkiens [126] illustrates requirements specification process with SysML and [10, 131, 125] provide applications of SysML to all phases of software development. SysML does not provide semantics for requirements although it is possible to associate contracts with individual operations through natural language or the OCL notation.

Evaluation of UML and SysML:

- **Explicit methodology:** UML and SysML are standardized notations, rather than methodologies.
- **Amount of training required:** Due to many different diagrams and notation elements, included to the standards, a significant amount of training is required.
- **Tool support:** A wide variety of tools support UML and SysML.
- **Requirement artifacts reusability:** There is no framework for UML artifacts reusability.
- **Requirements verifiability:** Requirements can be formulated with OCL to be a subject of formal verification.
- **Requirements unambiguity:** Since UML and SysML do not provide semantics, requirements remain ambiguous.
- **Traceability support:** SysML treats requirements as first class entities, establishing direct links between requirements and other software artifacts.
- **Seamlessness:** The software development process based on UML or SysML models is not seamless.

3.7 Scenario-driven approaches

Scenario-driven approaches rely on scenarios for capturing requirements, but add more rigor by providing a method for producing semi-formal or formal requirements artifacts from scenarios. RUCM (section 3.7.1) and OOAD (section 3.7.3) approaches facilitate producing a set of UML diagrams from scenarios. Use case maps (section 3.7.2) combine use cases with formal specification techniques.

3.7.1 Restricted Use Case Modeling

The Restricted Use Case Modeling approach (RUCM) [133] relies on a use case template and a set of restriction rules to reduce ambiguity of use case specification and facilitate transition to analysis models, such as UML class diagram and sequence diagram. The use case template is similar to the suggested by Cockburn (see section 3.5.1) with a few modifications:

- The "Dependency" field lists include and extend relationships to other use cases.
- The "Generalization" field lists generalization relationships.

- The main flow does not have branching and has a postcondition.
- Alternative flows can be of one of the three types, depending on which step in the reference flow they refer to: specific (refer to one specific step), global (refer to more than one step) and bounded (refer to any of the steps).
- Each alternative flow has a postcondition.
- Specific and global flows have an indication, from which steps of which flow can they branch.

Additionally, RUCM provides 26 restriction rules, which constrain the use of natural language and force using the predefined keywords for control structures. Rules R1-R7 constrain the NL use only in use case action steps; rules R8-R16 constrain the NL use in the entire use case; rules R17-R25 specify the keywords for control structures, such as INCLUDE USE CASE, IF-THEN-ELSE-ELSEIF-ENDIF, VALIDATE THAT, DO-UNTIL and ABORT; rule R26 enforces that each flow (basic and alternative) must have own postconditions.

The aToucan tool automates generation of UML class, sequence and activity diagrams [134]. First, it translates the use case, formulated in RUCM, to an intermediate UCMeta model. The transformation relies on a NL parser, which extracts model elements from the syntactic structure of each sentence, and on SentencePatterns and SemanticPatterns packages to identify functional roles and semantic meanings of sentences in use cases. Next, based on a set of rules, the aToucan tool transforms an intermediate model into a UML analysis model, including class, sequence and activity diagrams. The tool can generate traceability links from the textual use cases to the generated class diagram.

Evaluation of RUCM:

- **Explicit methodology:** The approach is supported with methodology.
- **Amount of training required:** The approach requires limited training.
- **Tool support:** The approach is supported by the aToucan tool [134].
- **Requirement artifacts reusability:** There is no specific framework for requirements reusability.
- **Requirements verifiability:** Requirements are not verifiable.
- **Requirements unambiguity:** Writing use cases in RUCM partially removes ambiguity.
- **Traceability support:** The tool can generate traceability links from the textual use cases to the generated class diagram, but not to the source code.

- **Seamlessness:** Software development is not seamless.

3.7.2 Use Case Maps

A Use Case Map (UCM) [9, 21] depicts several scenarios simultaneously. UCM represent use cases as causal sequences of responsibilities, possibly over a set of abstract components. In UCM pre- and postconditions of use cases as well as conditions at selection points can be modeled with formal specification techniques such as ASM or LOTOS [7].

Evaluation of Use Case Maps:

- **Explicit methodology:** The approach is supported with methodology.
- **Amount of training required:** The approach requires significant amount of training.
- **Tool support:** The approach is supported by the jUCMNav tool [59].
- **Requirement artifacts reusability:** There is no specific framework for requirements reusability.
- **Requirements verifiability:** Due to formal specification in LOTOS, requirements are verifiable.
- **Requirements unambiguity:** Formalisation of requirements eliminates ambiguity.
- **Traceability support:** There is no specific traceability support.
- **Seamlessness:** Software development is not seamless.

3.7.3 Object-Oriented Analysis and Design

Object-Oriented Analysis and Design (OOAD) [19] is a unified methodology for use-case-driven analysis and design, supported by UML [30] as a unified notation, and a set of graphical diagrams. OOAD suggests applying OO techniques (class-based decomposition, OO modeling) to the initial requirements, produced at the earlier stages of the development process. This method relies on a set of diagrams to capture the software system details and to be able to communicate them to the project stakeholders.

Evaluation of OOAD:

- **Explicit methodology:** The approach provides detailed guidance on creating diagrams, yet the process of capturing initial requirement is described briefly.

- **Amount of training required:** Due to many different diagrams and notation elements, included to the standards, a significant amount of training is required.
- **Tool support:** A wide variety of tools support UML [105, 84].
- **Requirement artifacts reusability:** There is no framework for UML artifacts reusability.
- **Requirements verifiability:** Constraints can be formulated with OCL to be a subject of formal verification.
- **Requirements unambiguity:** The requirements are not formalized and remain ambiguous. Fromulation constraints with OCL may eliminate ambiguity.
- **Traceability support:** There is no specific traceability support.
- **Seamlessness:** Software development is not seamless.

3.8 OO-Method

The OO-Method [98, 103, 102] combines conventional OO specification techniques [19] with formal specification, relying on the OASIS object-oriented specification language [72]. System specification consists of four complementary conceptual models: object model, dynamic model, functional model and presentation model. The Integranova tool supports the specification process with an interactive interface and provides automated generation of the implementation code.

Evaluation of OO-Method:

- **Explicit methodology:** The approach is supported with methodology.
- **Amount of training required:** The approach requires limited training due to the tool support.
- **Tool support:** Integranova tool [50] supports the specification process.
- **Requirement artifacts reusability:** The approach does not provide framework for requirements reusablity.
- **Requirements verifiability:** Requirements are verifiable due to formalisation.
- **Requirements unambiguity:** Requirements formalisation eliminates ambiguity.

- **Traceability support:** There is no specific traceability support.
- **Seamlessness:** Software development is not seamless.

3.9 Goal-oriented requirements engineering

In goal-oriented requirements engineering requirements are obtained through a series of refinements of high-level goals [63], [132]. Goal model is an annotated graph displaying how lower-level goals contribute to higher-level ones. The leaves of the goal refinement tree are either software requirements or expectations on the environment. With the help of the Objectiver tool [108], requirements can be linked to other artifacts, such as goals, environment agents, or operations. However, traceability links to natural language requirements documents or implementation artifacts are out of scope of the approach.

Evaluation of goal-oriented requirements approaches:

- **Explicit methodology:** The approach is supported with the detailed methodology.
- **Amount of training required:** The approach requires limited training.
- **Tool support:** Tools such as Objectiver [108] and jUCMNav [59] support the approach.
- **Requirement artifacts reusability:** There is no framework for requirements reusability.
- **Requirements verifiability:** There is no verification mechanism.
- **Requirements unambiguity:** According to the approach, the requirements are formulated in natural language and thus are ambiguous. The requirements can be formalised with notations such as Event-B, which are not native to the approach.
- **Traceability support:** Requirements are linked to other requirements artifacts, such as goals, agents and operations. The requirements are not linked to the implementation artifacts.
- **Seamlessness:** Software development is not seamless.

3.10 Test-driven software development

In test-driven software development processes unit tests can be viewed as the means of capturing requirements functionality. In test-driven development (TDD) [15] a software engineer writes unit tests before implementing systems

functionality in small iterations. A unit is a minimal testable software component, such as a method, a group of related methods, or a class. Unit testing can be automated (relying on a framework such as jUnit or TestNG) or manual. As in TDD tests are written prior to implementing the code, testing occurs immediately after the implementation. Tests serve as a guide to code writing. Further they stay with the software and provide a quick feedback to systems changes.

Evaluation of test-driven software development:

- **Explicit methodology:** The approach provides a methodology and a significant number of illustrative examples.
- **Amount of training required:** The approach requires requirement engineers to have basic quality assurance knowledge.
- **Tool support:** The approach relies on unit testing frameworks and tools, such as JUnit [60] and TestNG [121].
- **Requirement artifacts reusability:** Although a number of patterns support TDD [15], the approach to requirements reuse in TDD has not been proposed yet.
- **Requirements verifiability:** The requirements can be verified by dynamic testing [85].
- **Requirements unambiguity:** Converting requirements into tests resolves ambiguity.
- **Traceability support:** The unit tests, which capture requirements in TDD are directly linked to the implementation code.
- **Seamlessness:** The approach is seamless. Requirements are seamlessly captured in a programming language and are directly linked to the implementation. However, TDD relies on dynamic testing and does not provide means for static verification [15].

3.11 Behavior-driven software development

Behavior-driven development (BDD) [120, 113] is a software development process which further develops the TDD ideas. In BDD requirements are formulated as user stories, following a specific template. Dedicated tools transform user stories to parameterized unit tests.

Evaluation of test-driven software development:

- **Explicit methodology:** BDD provides a methodology and a significant number of illustrative examples.

- **Amount of training required:** This approach requires requirements engineers to have a basic quality assurance knowledge.
- **Tool support:** The approach relies on unit testing frameworks and tools, such as JUnit [60], TestNG [121], and Cucumber [31].
- **Requirement artifacts reusability:** Requirements are not reusable.
- **Requirements verifiability:** The requirements can be verified by dynamic testing.
- **Requirements unambiguity:** Converting requirements into tests resolves ambiguity.
- **Traceability support:** The unit tests are directly linked to the implementation code.
- **Seamlessness:** This approach is seamless.

3.12 Contract-based and seamless approaches

3.12.1 ACL/VF framework

In ACL/VF frramework [12, 13] use cases capture requirements, which are further formalized as grammars of responsibilities. Another Contract Language (ACL) contracts (pre- and postconditions and invariants) specify constraints, which scenarios' or responsibilities' execution poses on the system's state. In this approach requirements model is decoupled from the candidate implementation: a dedicated binding tool maps elements of requirements model to the elements of candidate implementation.

Evaluation of ACL/VF framework:

- Explicit methodology: The approach goes with a methodology.
- Amount of training required: The approach requires familiarity with design by contract, ACL and formal grammars.
- Tool support: The approach is supported by a binding tool (VF the Validation Framework). At the moment of writing the thesis the tool is not publicly available.
- Requirement artifacts reusability: The approach does not provide a framework for reusing requirements artifacts.
- Requirements verifiability: Requirements can be statically verified.

- Requirements unambiguity:. Capturing requirements with contracts removes ambiguity.
- Traceability support: As the requirements model is decoupled from the implementation, traceability links are established by the binding tool.
- Seamlessness: The approach is not seamless, as contracts are specified in ACL, rather than implemented in programming language.

3.12.2 Multirequirements

The multirequirements approach [79] suggest using a single notation (Eiffel programming language) for requirements, design and implementation. Requirements are formulated in 3 interconnected layers: natural language, software contracts in programming language and diagrams.

Evaluation of Multirequirements:

- **Explicit methodology:** The approach does not provide a methodology.
- **Amount of training required:** The approach requires familiarity with design by contract.
- **Tool support:** The approach relies on EiffelStudio [115], an IDE for the Eiffel language.
- **Requirement artifacts reusability:** The approach does not provide a framework for reusing requirements artifacts.
- **Requirements verifiability:** Requirements can be statically verified.
- **Requirements unambiguity:** Capturing requirements with contracts removes ambiguity.
- **Traceability support:** The framework for traceability links is not provided.
- **Seamlessness:** This approach is seamless.

3.12.3 SIRCOD

The SIRCOD approach [38] provides a pipeline for converting natural language requirements to programming language contracts. The approach relies on the domain specific language, RSML, for automating conversion from natural language to programming language.

Evaluation of SIRCOD:

- **Explicit methodology:** The approach relies on existing requirements document.
- **Amount of training required:** The approach requires familiarity with design by contract.
- **Tool support:** The approach is supported by the Inspecto tool [118].
- **Requirement artifacts reusability:** The approach does not provide a framework for reusing requirements artifacts.
- **Requirements verifiability:** Requirements can be statically verified.
- **Requirements unambiguity:** Capturing requirements with contracts removes ambiguity.
- **Traceability support:** The requirements have direct links to other software artifacts.
- **Seamlessness:** The approach is seamless.

3.12.4 Seamless Object-Oriented Requirements

In the Seamless Object-Oriented Requirements approach (SOOR), requirements are documented as software classes which makes them verifiable and reusable [87]. The key notions of SOOR are specification drivers, and semantic assertions (contracts expressed by pre- and post-conditions). Specification drivers are contracted routines, expressed only in terms of their formal arguments, that serve specification purposes. Specification drivers take objects to be specified as arguments and express the effect of operations on those objects with pre- and postconditions [88]. Seamless object-oriented requirements are concrete classes capturing requirements as specification drivers. Specification drivers capture formal semantics of requirements. Each specification driver is supported with a comment, which captures a natural language version of the same requirement. Seamless object-oriented requirements serve as:

- Proof obligations, since each specification driver captures formal semantics of a requirement.
- Parameterized unit tests.

Seamless Object-Oriented Requirement Templates (SOORT) are requirements patterns captured in generic and deferred classes. Libraries of requirements templates for software components and control software temporal properties, implemented in Eiffel, are publicly available [86].

Evaluation of SOOR:

- **Explicit methodology:** The approach does not provide an explicit methodology and relies on existence of NL requirements document.
- **Amount of training required:** The approach requires basic design by contract knowledge.
- **Tool support:** The approach is supported by the AutoReq tool [89].
- **Requirement artifacts reusability:** The approach provides a framework for requirements reuse (SOORT) and a library of reusable templates.
- **Requirements verifiability:** Requirements can be statically verified.
- **Requirements unambiguity:** Converting requirements into specification drivers removes ambiguity.
- **Traceability support:** The requirements artifact are source code elements, so the can be directly linked to the implementation code.
- **Seamlessness:** The approach is seamless.

3.12.5 PEGS

The PEGS approach attempts to provide a definition and taxonomy of requirements. According to this approach, requirements pertain to a Project intended, in a certain Environment, to achieve some Goals by building a System. Thus, requirements specification consists of four books: Project, Environment, Goals and Systems, which correspond to each of these components [81]. The approach provides principles and techniques of requirements specification, yet does not provide an explicit methodology.

Evaluation of PEGS:

- **Explicit methodology:** The approach is not prescriptive, but through case studies, listed in a companion book (to be published soon) provides a guide to obtain efficient specifications.
- **Amount of training required:** The approach requires basic design by contract knowledge.
- **Tool support:** The approach is not supported by a dedicated tool.
- **Requirement artifacts reusability:** The approach does not provide a framework for requirements reuse.

- **Requirements verifiability:** Requirements captured with contracts can be statically verified.
- **Requirements unambiguity:** The requirements formulated in OO way are not ambiguous.
- **Traceability support:** The approach does not provide specific requirements traceability support.
- **Seamlessness:** The approach is seamless.

3.13 Conclusion

The table 3.3 below summarizes the evaluation of the reviewed approaches versus the proposed criteria.

None of the reviewed approaches satisfies all of the criteria identified in section 3.3. TDD and BDD approaches seem to be very close to the target. Indeed, they do not require substantial training, and at the same time allow producing unambiguous and verifiable requirements. The downside of these approaches is that they rely on scenarios, which as will be shown below are not abstract enough to be *requirements*. If there are few scenarios, they can be properly documented and agreed with the stakeholders, but they will not cover all of the requirements. At the same time, tests are not easily readable by all stakeholders. If scenarios attempt to cover *all* possible situations, their number explodes, so that it is extremely hard to express, manage and trace requirements in such a way. It has to be noted as well that BDD and TDD do not provide mechanisms for static verification.

SIRCOD and SOOR approaches also cover most of the target characteristics. Indeed the present thesis largely relies on the advancements of the two approaches but gives more focus to the approach's usability and requirements traceability management.

	Methodology	Required background	Tool support	Require-ments reusabil-ity	Require-ments verifiability	Require-ments unambiguity	Traceability support	Seamlessness
NL-based approaches	Yes	Some	Yes	No	No	Partial	No	No
UML and SysML	No	Substantial	Yes	No	Partial	Partial	Yes	No
Scenarios	No	Some	Yes	No	No	No	No	No
RUCM	Yes	Some	Yes	No	No	Partial	Partial	No
Use Case Maps	Yes	Substantial	Yes	No	Yes	Yes	No	No
OOAD	Yes	Substantial	Yes	No	Partial	Partial	No	No
OO	Yes	Some	Yes	No	Yes	Partial	No	No
GORE	Yes	Some	Yes	No	No	Yes	Partial	No
TDD	Yes	Some	Yes	No	Yes	Yes	Yes	Yes
BDD	Yes	Some	Yes	No	Yes	Yes	Yes	Yes
ACL/VF	Yes	Substantial	No	No	Yes	Yes	Partial	No
Multirequirements	No	Some	Partial	No	Yes	Yes	No	Yes
SIRCOD	Partial	Some	Yes	No	Yes	Yes	Yes	Yes
SOOR	No	Some	Yes	Yes	Yes	Yes	No	Yes
PEGS	No	Some	Partial	No	Yes	Yes	No	Yes

Table 3.3: Summary of the reviewed approaches to requirements.

Part II

The unified solution

Chapter 4

Unified Object-Oriented approach to Requirements

The idea of capturing software requirements with contracts has been proposed a long time ago and gave rise to several approaches that we reviewed in section 3.12. However, as we demonstrated in the summary of section 3, those approaches focus on different aspects of OO requirements and do not provide a methodology of their specification. The thesis develops ideas put forward in previous work such as Multirequirements [79], SOOR [87], SIRCOD [38] and PEGS [81] approaches, to enrich them with a methodology of requirements specification. The resulting Unified Object-Oriented approach to Requirements (UOOR) benefits from OO techniques, which have proven to be successful in software implementation, and scenarios, which are commonly used for requirements elicitation.

This chapter devises the Unified Object-Oriented approach to Requirements. It starts from the theoretical and technical background 4.2, then presents the key elements of the approach 4.3, followed by a guide on producing UOOR specification (sections 4.4 -4.9).

4.1 Introduction

This section introduces the background of the UOOR approach: the targeted projects, the UOOR fundamental idea and notation and the case study that will be used for running examples in this chapter.

4.1.1 Targeted projects

The level of criticality of formulating requirements properly depends on the project type. To illustrate this, we refer to the "ABC of software engineering", described in [80]. "A" stands for *acute* projects - the life-critical or mission-critical software. As malfunctioning of such software may cause severe money losses or human deaths, its development must go through rigorous process, involving use of formal methods and thorough verification and validation procedure. "B" refers to *business applications* that serve key business processes

in the organizations - their use may not cause deaths or severe injuries, but can lead to significant money losses. “C” is for *casual* applications, such as web apps or many other kinds of applications, whose malfunctioning is irritating but does not cause significant troubles. The development of the “A” projects, as mentioned above, relies on rigorous software development processes and is covered by a variety of formal specification methods. The development of “C” type projects often has an ad-hoc requirements process and relies more on fast prototyping and development in small iterations. The development of “B” type projects can not rely on ad-hoc requirements, as consequences for an enterprise can be severe. At the same time, applying formal methods for such projects is usually too costly. Moreover, formal methods rely on formal notations, which generally cannot be read by customers, which impedes communication regarding the requirements. The UOOR approach, devised in the present thesis, targets these “B” type projects and suggests a methodology of producing unambiguous, traceable and verifiable requirements.

4.1.2 Fundamental idea and notation

The Unified Object-Oriented approach to Requirements unifies scenarios as a commonly used requirements technique with software contracts as a requirements formalisation approach, which does not require specific background in formal methods [90]. The reasons for combining these two techniques are the following:

- Scenarios are widely used in industry and are perceived as an excellent tool for eliciting requirements and communicating with project stakeholders.
- Scenarios, formulated in natural language, are ambiguous and lack abstraction.
- Software contracts provide means for requirements formalisation.
- Software contract do not rely on a specific mathematical notation and are as easy to formulate as “if ... then ” statements. Capturing requirements with contracts makes them verifiable, reusable and traceable.

As discussed earlier in section 3.2.5, the seamlessness of software development from requirements to implementation and verification may significantly facilitate requirements traceability. With this in mind, we suggest to use a statically typed object-oriented programming language to serve as the formal notation for requirements. The Eiffel language [77] seems to be a perfect choice due to its readability and native support of contracts. Further in this thesis all the examples will be in Eiffel, and the tool support will be based on the facilities of EiffelStudio - a general-purpose IDE for Eiffel language. Nevertheless, Eiffel is not the only possible option - in fact, any statically typed OO language supporting contracts can serve as the notation. For example, the RQCODE approach provides a framework for seamless expression of security requirements in Java [95, 96, 111].

4.1.3 Running example

This chapter provides the examples of the requirements produced for the Library Management System (LMS) case study [93].

LMS is a system that manages the process of book borrowing in a public library. The two main categories of LMS users are librarians (library employees) and patrons (library members allowed to borrow books). In particular, the LMS covers the following functionalities:

- Inserting and deleting books to the books catalog.
- Registering book reserves, checkouts and returns.
- Searching books in a catalog.
- Creating, modifying and deleting personal profiles.

4.2 Theoretical and technical background

4.2.1 Seamless software development

The conventional approach to software development implies using several notations: natural language for requirements, formal notation for requirements formalisation, UML for diagrams and programming language for implementation. Transition from one notation to the other is error-prone. Moreover, introducing change in one notation often is not easily propagated to artifacts expressed in another notation.

Seamless software development addresses this issue by suggesting to use a uniform method and a uniform notation throughout the entire software development lifecycle. It means that an implementation programming language (rather than a dedicated modeling language or notation) is used as the notation for requirements formalization.

4.2.2 Contract-based requirements

The core idea of object-oriented modeling (including OO requirements, OO design, OO programming and other applications, collectively called "object technology") is to describe systems through the objects they manipulate and to express this description in terms of the operations applicable to objects and the properties of these operations (and not, for example, in terms of the objects physical representations). A type of potential objects, characterized by their operations (also called "features") and associated properties (also called "contracts") is called a class. "Class" is the central concept of object technology, and an OO model is defined by a set of classes and their interconnections.

Information hiding implies that modules of a systems description (in OO modeling, classes) can refer to others in terms of their abstract specification. To be useful, such specifications should not just be structural, giving the types of arguments and results of the operations in a class, but also semantic, describing

the abstract properties of these operations and the class as a whole. Design by Contract techniques [78] provide this semantic specification for both operations (preconditions and postconditions) and classes (class invariants).

Contracts have many applications, including as documentation of the software and as a guide to both exception handling and the proper use of inheritance. When applied to code, they also provide a systematic approach to testing and debugging (in a development environment that provides the ability to evaluate contract elements at run time).

The application of most direct relevance to requirements, as illustrated by the PATRON example sketched below (list. 4.1), is to model the System or Environment by providing not only structural properties (object types and the applicable operations) but also through precise semantics (the properties of these types and operations).

In this example the name of a class reflects the modeled component of the system's environment - a patron. The PATRON class is deferred, since it provides only semantics of its features, and the features are not implemented. Feature num_reserved is a query, which reflects a number of books currently reserved by a patron. Feature place_hold specifies operation applicable to objects of type patron: a patron can place hold on a book. Precondition, introduced with the keyword **require**, expresses a constraint: *"A patron can place hold on a book only if it is available".* Class invariant, introduced with the keyword **invariant** expresses the environment constraint *"A patron cannot have more than 5 books on hold".*

```
deferred class
    PATRON
feature
    num_reserved: INTEGER
       -- Number of books reseved by the patron

    place_hold(b: BOOK)
       -- reserve a book
        require
            b.is_available
        deferred
        end

invariant
    num_reserved ≤ 5
end
```

Listing 4.1: Example of a requirements class.

Such semantic models lend themselves to automatic verification. Beyond run-time monitoring of contracts, which only applies to executable code, classes equipped with contracts can be verified using automatic tools such as JML [67] and AutoProof, the verifier for Eiffel [122].

4.2.3 Specification drivers

The object-oriented style of modeling has modular units (classes) each organized around one type of objects. Correspondingly, contract techniques apply within a class, making it possible for example to express the requirement after a patron returns a book, this book is considered available as a postcondition of an operation return in a class BOOK.

Some properties apply to several objects, possibly of different types. In particular, this is the case in scenarios, expressing sequences of operations. While the OO style would seem to break down in such cases, it actually handles it in a simple way through the introduction of specification drivers [88]. The idea (generalizing techniques already present in some classic Design Patterns such as Visitor [39]) is to express such cross-object properties through classes designed specifically for specification purposes.

As an illustration, listing 4.2 presents a specification-driver assertion, which describes a generic scenario of using the relevant features and specifies its effect through the postcondition:

```
class
    LIBRARY_BOOK_USAGE_STORIES
feature

    reserve_book_successfully (b: BOOK; lb: LIBRARY; p: PATRON)
        require
            p.num_reserved <5
            b.is_available
        do
            lb.place_book_on_hold (b, p)
        ensure
            b.is_on_hold
            b.patron = p
        end
end
```

Listing 4.2: Example of a specification driver.

Specification drivers retain the OO style of requirements specification but make it more general by covering arbitrary properties, not necessarily expressible within a single class of the original OO model.

4.2.4 Autoproof

Autoproof is a tool providing static verification of the functional properties of Eiffel programs [122]. The verification process is autonomous, yet the input programs should be annotated. The tool checks whether the implementation satisfies its contracts: for every routine it ensures that it is called in a state satisfying its precondition and that after execution its postcondition and class

invariant are satisfied. If some of the contracts may be violated, Autoproof outputs warning, pointing at the contract that may be violated. The Proof2Test tool [45] can automatically generate tests from the failed proofs. The generated tests can be executed in AutoTest [82], a testing tool for Eiffel programs, integrated into EiffelStudio.

4.3 Key elements of the UOOR approach

The UOOR approach builds on the general ideas of object-oriented requirements, to which it adds its own specific elements.

The following concepts are common to all OO requirements approaches:

- Object types, described through applicable operations – queries (providing information) and commands (updating information).
- Software contracts, which capture the semantics of operations.

In addition to these basic ideas, UOOR relies on the following concepts:

- Specification drivers [88], which capture the system's behaviours (section 4.7).
- Seamless requirements traceability, which facilitates accommodating changes in requirements (chapter 5).

The UOOR approach provides requirements specification in three dimensions: an object model, a functional specification and a behavioral specification (see fig. 4.1).

The **object model** captures key abstractions in the application domain in the form of software classes. This static model specifies abstract data types for all relevant environmental phenomena.

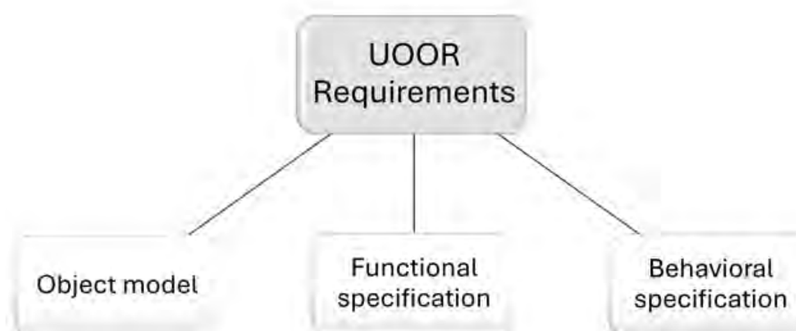

Figure 4.1: Three dimensions of UOOR requirements.

Functional specification provides a specification of individual operations. Operations correspond to features of classes of the object model. Their abstract specification relies on in-class contracts.

Behavioral specification defines permissible sequences of operations. Behavioral specification relies on the following mechanisms:

- Specification drivers capture scenarios as example sequences of operations and may serve testing purposes.
- Software contracts provide more abstract specification of properties that would otherwise be expressed as time-ordering constraints.

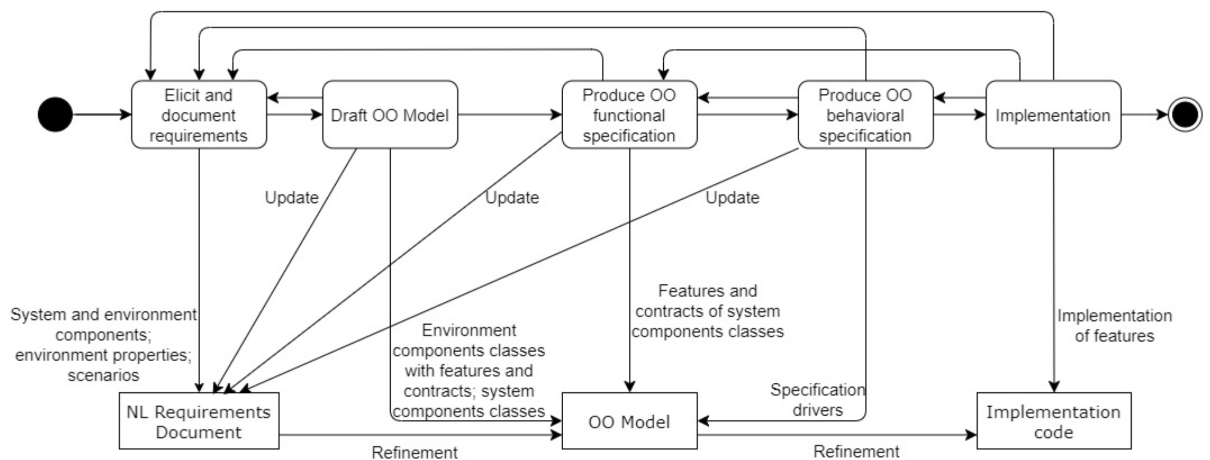

Figure 4.2: Overview of the UOOR approach.

Once adopted, the object-oriented approach pervades the whole process of requirements elicitation and analysis. In other words, the idea is not to produce some traditional kind of requirements and then make the result object-oriented; instead, OO principles help structure the entire effort by providing a unifying conceptual framework for describing the system and its environment.

Although the process is iterative, we can list its basic steps:

1. Eliciting and documenting requirements: The outcome of this process is a requirements document in a natural language (section 4.4).
2. Drafting OO model: OO model captures the key components of the system and its environment with classes, linked with client and inheritance relations (section 4.5).
3. Producing functional OO requirements: Functional OO specification enriches the OO model with features, expressing functional requirements, and their contracts, expressing environment properties and constraints (section 4.6).
4. Producing OO behavioral specification: Abstract properties of system's functions (such as time-ordering constraints) are extracted from scenarios

and added to the OO model; concrete examples of scenario sequences are expressed as specification drivers (contracted routines of scenario classes) and serve verification purposes (section 4.7).

5. Refining requirements: The implementation process relies on refinement: implementation classes inherit from requirements classes and thus must satisfy the requirements expressed as contracts (section 4.8).

Fig. 4.2 envisions the overview of the UOOR approach. The following sections 4.4 - 4.7 provide the details of each step.

4.4 Eliciting and documenting requirements

The UOOR approach unifies the existing requirements approaches, so it does not advocate any particular methodology for eliciting and documenting requirements. Nevertheless, the PEGS template [81] serves as an excellent starting point for further OO analysis of requirements. In the PEGS template, requirements are organized into four books, each corresponding to one of the requirements dimensions: goals, environment, system, project. The template guides the requirements engineer to specify important requirements elements, such as:

- System and environment components.
- Environment assumptions, constraints and invariants.
- System functions.
- Scenarios.

Whereas any other template may be used to document requirements in the UOOR approach, it is important that it includes the aforementioned elements. Another important property of the requirements template, used in the UOOR approach is that it should document each requirement in a separate paragraph. Otherwise, the approach leaves freedom in the process of requirements elicitation, and provides specific guidance on how to proceed with the elicited data (i.e. how to analyze scenarios and environment properties).

The approach will also tackle the important task of tracing requirements from requirements document to other project artifacts and in the reverse direction (see sections 4.6, 4.8.2, 6.4).

4.5 Modeling components of the system and its environment

To produce an OO model, a requirements engineer should describe key abstractions in the application domain with software classes. These classes should

cover both the system and its environment. For the Library Management System, examples of such classes are: LIBRARY (system class), BOOK and PATRON (environment classes). During its operation, the system manipulates objects - instances of these classes. In the LMS, such objects will be computer representations of books kept in the library, patrons of the library, the librarys catalog and other library objects, physical or conceptual.

In the PEGS template, the system and environment components are listed in the sections E.2 "Environment components" and S.1 "System components", so we can link the natural language description of the component with the class, abstracting this component (the process of creating traceability links is described in sections 4.8.2 and 6.2).

Object technology supports modeling systems at various levels of abstraction. In particular, both a feature and a class may be either effective or deferred. "Effective" means fully implemented (and hence ready to execute as part of a program). A feature or class is deferred if its definition does not include an implementation or (for a class) includes a partial implementation only; it may still, however, include an abstract specification of the properties of the feature or class, in the form of a "contract" as explained below. Deferred features and classes are particularly useful for requirements, since requirements focus on specifying behavior rather than implementing it. For example, LIBRARY_ITEM may be a deferred class since it describes an abstract concept with several possible concrete realizations, such as BOOK. In some cases, requirements classes have to be effective, in particular, scenario classes and test classes.

Client and inheritance relations capture the relations between the environment and system components. For example, operations on a book involve, among other objects, a patron and a library. The class representing a book will be a client of the classes representing the other two concepts. A class inherits from another if it represents a specialized or extended version of the others concept. For example, book and magazine both belong to the general category of library item, which can be represented by a class LIBRARY_ITEM. BOOK and MAGAZINE are "descendants" of LIBRARY_ITEM through the inheritance relation. As another example, there could be two categories of patrons: regular and research patrons with different restrictions, such as the number of books on hold or a hold duration. In that case, classes REGULAR_PATRON and RESEARCH_PATRON would inherit from the class PATRON, which captures features and contracts applicable to all patrons.

Properties of data types are defined through operations: queries and commands. Queries provide information about the objects, whereas commands update the corresponding objects.

A specification of the properties of systems and their objects through the list of associated classes and their features only gives structural properties. To provide the actual semantics of these elements other than through the implementation we should also express their abstract properties. Contracts fulfill this need. They include:

- The precondition of a feature, specifying the conditions under which it can be used.

- Also for a feature, the postcondition, expressing properties resulting from its application.
- For a class, the class invariant, expressing consistency properties applicable to all objects ("instances") of the class.

Such a contract element consists of assertions, each of which is an individual boolean property applicable to the corresponding objects, such as (for a `BOOK` instance) "the book is currently on loan". We rely on contracts to express:

- Properties, imposed by environment, such as *"a patron can reserve no more than 5 books* (we will discuss them below in this section).
- Properties that specify system's function, such as *"if a patron has less than five holds, placing a hold is successful and the book becomes reserved"* (we will discuss them in section 4.6).

The important part of the OO model are the properties imposed by the system's environment. According to [81] those properties can be of one of the three types:

- Assumptions - properties, that are satisfied by the environment, so the system takes them for granted (*"The length of a book title does not exceed 100 characters"*).
- Constraints - obligations coming from the environment that the system must comply with (*"A patron can reserve no more than 5 books"*).
- Invariants - environment properties that the system must maintain (*"A book can have exactly one of the following statuses: available, on hold, borrowed, due"*).

Various types of environment properties play different role in software development.

Assumptions simplify the work of the developers. For example, the assumption *"The length of a book title does not exceed 100 characters"* provides information on the size of manipulated data, when performing operations with the book titles. At the same time, assumptions (and this one in particular) are falsifiable: they can be wrong. Expressing the assumptions explicitly with assertions helps in verifying their validity: if during system verification and validation, the assertion expressing the system assumption is violated, the respective error message will be displayed. The developers will then be able to assess and address the consequence of reconsidering the assumption.

Constraints and invariants express the properties of the environment that the system must respect. Such constraints can be expressed as class invariants in the classes, denoting the environment components. Thus, the invariant *"A book can have exactly one of the following statuses: available, on hold, checked out, due"* can be expressed with the following class invariants (listing 4.3):

```
deferred class
    BOOK
feature
    is_available: BOOLEAN
    is_on_hold: BOOLEAN
    is_checked_out: BOOLEAN
    is_due: BOOLEAN
invariant
    is_available implies not (is_on_hold or is_checked_out or is_due)
    is_on_hold implies not (is_available or is_checked_out or is_due)
    is_checked_out implies not (is_available or is_on_hold or is_due)
    is_due implies not (is_available or is_on_hold or is_checked_out)
end
```

Listing 4.3: Implementation of the invariant *"A book can have exactly one of the following statuses: available, on hold, checked out, due"*.

The constraint *"a patron can reserve no more than 5 books"* can be expressed as the invariant of the class PATRON (listing 4.4):

```
deferred class
    PATRON
feature
    num_reserved: INTEGER
       -- Number of books reseved by the patron
invariant
    num_reserved ≤ 5
end
```

Listing 4.4: Implementation of the invariant *"A patron can reserve no more than 5 books"*.

Properties that refer to the operations, applicable to the components of the environment (such as the operation place_hold(b: BOOK) of the class PATRON) can be expressed with pre- and postconditions. The environment constraint, related to this operation, is the following: *"A patron can place hold on a book if the book is available."*

This constraint is implemented as a precondition of the feature place_hold of the class PATRON (listing 4.5):

```
deferred class
    PATRON
feature
    num_reserved: INTEGER
        -- Number of books reseved by the patron

    place_hold(b: BOOK)
        require
```

```
            b.is_available
        deferred
        end
invariant
    num_reserved ≤ 5
end
```

Listing 4.5: Implementation of the constraint *"A patron can place hold on a book if the book is available."*

The constraint, related to the feature "place hold" saying that a patron cannot reserve more than 5 books, is implicit. But what happens if a patron has 5 books reserved and attempts to reserve one more? One option is that the book reservation will be denied. Another option is to cancel the oldest hold and complete the last one. To avoid the ambiguity we add one more constraint (which is in fact an implicit business rule): *"A patron can place hold on a book if she has less than 5 books on hold."*

To express this constraint, we add one more precondition to the feature place_hold (listing 4.6):

```
deferred class
    PATRON
feature
    num_reserved: INTEGER
       -- Number of books reseved by the patron

    place_hold(b: BOOK)
        require
            b.is_available
            num_reserved <5
        deferred
        end
invariant
    num_reserved ≤ 5
end
```

Listing 4.6: Implementation of the constraint *"A patron can place hold on a book if she has less than 5 books on hold."*

4.6 Producing OO functional specification

In object-oriented requirements, elements of system functionality (system features) correspond to features (procedures) of software classes. The properties of those operations are captured with contracts.

Consider the following requirement:

"The Library Management System shall provide the ability to place hold on books."

To express this requirement in an object-oriented style, we add the feature place_book_on_hold to the LIBRARY class (listing 4.7).

```
deferred class
    LIBRARY
feature
    has_patron(p: PATRON) : BOOLEAN
      -- Is p an active patron of the library?
        deferred
        end

    has_book(b:BOOK): BOOLEAN
      -- Is b in the library's catalogue?
        deferred
        end

    place_book_on_hold (b:BOOK; p: PATRON)
      -- Reserve a book b by patron p
        deferred
        end
```

Listing 4.7: Implementation of the requirement *"The Library Management System shall provide the ability to place hold on books."*

Further specification can be derived from the environment constraint:
A patron is limited to five holds at any given moment.
This constraint is already expressed in the OO model in the invariant of the class PATRON. However, we have to infer the specification of the operation place_book_on_hold:

- If hold is allowed, placing a hold is successful and the book becomes reserved.
- If hold is not allowed, placing a hold is not successful, and the book remains available.

Listing 4.8 provides the OO implementation of these two constraints:

```
deferred class
    LIBRARY
feature
    has_patron(p) : BOOLEAN
      -- Is p an active patron of the library?
        deferred
        end

    has_book(b): BOOLEAN
      -- Is b in the library's catalogue?
        deferred
```

```
        end

    place_book_on_hold (b, p)
      -- Reserve a book b by patron p
        deferred
        ensure
            old p.num_reserved <5 implies (b.is_on_hold and b.patron.
               is_equal(p) and p.num_reserved= old p.num_reserved+1)
            old p.num_reserved ≥ 5 implies (b.is_available and
               p.num_reserved = old p.num_reserved)
        end
end
```

Listing 4.8: Implementation of the constraints on the `place_book_on_hold` feature.

An object-oriented requirement, expressed in Eiffel, is further linked with the natural language requirement, stored in text format, by adding a hyperlink to a requirements document and an annotation to the respective requirements class. Clicking the hyperlink from the class code will open the requirements document at the bookmark, corresponding to the respective requirement. Clicking the hyperlink in the requirements document will open the source code at the feature that is specified by the given requirement. The technical details of linking requirements are discussed in sections 6.2, 6.4.

4.7 Producing an OO behavioral specification

As noted in 3.5, it is common to rely on use cases or other forms of scenario to model system behaviors. These techniques are not, however, the only way to describe behavior. In this section we review how UOOR helps achieve this goal.

A scenario is a certain pattern of exercising the features (operations) of one or more classes; the use cases have little to do with object-orientation, since they are essentially procedural. Nevertheless they play a complementary role and serve elicitation and testing purposes. According to the use case 2.0 approach, a successor of the use-case-driven approach, developed by Ivar Jacobson [57], use case stories play the key role in behavioral specification. A story describes a possible path through a use case that is of value to a user or other stakeholder.

In UOOR a story is expressed as a routine, exercising the features of the respective classes. The routines, specifying a set of related use case stories can be grouped in a separate class. In that case a story is a typical instance of the concept of specification driver, presented in section 4.2.3.

Below is a general use case stories class exercising features of classes `BOOK`, `PATRON` and `LIBRARY` (listing 4.9):

```
class
```

```
    LIBRARY_BOOK_USAGE_STORIES
feature

    reserve_book_successfully (b: BOOK; lb: LIBRARY; p: PATRON)
        require
            p.num_reserved < 5
            b.is_available
        do
            lb.place_book_on_hold (b, p)
        ensure
            b.is_on_hold
            b.patron = p
        end

    reserve_book_num_holds_exceeded (b: BOOK; lb: LIBRARY; p: PATRON)
        require
            p.num_reserved ≥ 5
            b.is_available
        do
            lb.place_book_on_hold (b, p)
        ensure
            b.is_available
        end


    holding_available_book_by_two_patrons(b: BOOK; p1, p2: PATRON; l:
        LIBRARY)
        require
            b.is_available
            p1 ≠ p2
            l.has_patron (p1)
            l.has_patron (p2)
        do
            l.place_book_on_hold (b, p1)
            l.place_book_on_hold (b, p2)
        ensure
            l.book_is_on_hold (b, p1)
            not l.book_is_on_hold (b, p2)
        end

    -- Other use case stories
end
```

Listing 4.9: Use case stories class.

Use case stories, expressed in OO style, also serve as test cases. When actual arguments are passed, use case stories become tests. Let's take a story

''Reserve a book when number of holds is exceeded". The preconditions of a story set restrictions on the arguments that should be passed to the routine: a book must be available and a patron must have at least five books on hold. When we pass the actual arguments, satisfying the precondition, the execution of the routine will satisfy its postconditions in case of correct implementation of the feature `place_book_on_hold`.

OO techniques avoid premature *time-ordering decisions*. While it is possible for an OO specification to express a time-ordered scenario such as a use case, object technology also supports a more general and abstract specification style, based on contracts.

Scenarios specify the order in which operations will get executed. Enforcing such an ordering specification at the level of requirements is often a premature decision. In reality, the order of the steps is not cast in stone. Using a preset ordering is convenient to describe desirable scenarios, or more generally the expected ones. But what happens in life is not always what we hope for, or expect. What if the customer returns a damaged book? Should the book not remain unavailable until it is repaired? To specify scenarios that depart from the standard ones, it is possible to use extensions. But this solution does not scale. Writing ever more use case extensions to cover all such situations leads to an explosion of special cases which soon becomes intractable. In practice, it is possible to write use cases to cover the most common scenarios, but they are only a small subset of the possible ones, in the same way that, in programming, tests can only cover a minute subset of possible inputs.

Class `BOOK` specifies these logical constraints in the form of contracts (listing 4.10). It is possible to specify a strict order of operations o_1, o_2, ..., as in a use case, by having a sequence of assertions p_i such that operation o_i has the contract clauses **require** p_i and **ensure** $p_i + 1$; but assertions also make it possible to specify a much broader range of allowable orderings. Logical constraints are more general than sequential orderings.

The specific sequence of actions described in the use case (Main scenario) is compatible with the logical constraints: one can check that in the sequence

```
        -- Main scenario of the use case ''borrow a book'':
    place_hold (patron: PATRON)
    checkout (patron: PATRON)
    return (patron: PATRON)
```

the postcondition of each step implies the precondition of the next one (the first has no precondition). Prescribing this order strictly is, however, overspecifying. For example, it may be possible to perform additional operations between `place_hold` and `check_out`.

```
deferred class
    BOOK
feature
        -- is_available initialized to True
        -- is_on_hold, is_checked_out, is_due initialized to False
    is_available, is_on_hold, is_checked_out, is_due: BOOLEAN
```

```
    place_hold (p: PATRON)
      -- Place a hold on a book. Set is_on_hold
        require
            is_available
        deferred
        ensure
            is_on_hold
            not is_available
        end

    checkout (p: PATRON)
        -- Check out the book
        require
            is_on_hold
        deferred
        ensure
            is_checked_out
        end

    return
        -- Return the book to the library
        require
            is_checked_out or is_due
        deferred
        ensure
            is_available
        end
end
```

Listing 4.10: Illustration of time-ordering constraints.

4.8 From requirements to code

Seamless software development process relies on iterative refinement of requirements into executable code. Section 4.8.1 describes the details of the refinement process; section 4.8.2 explores how requirements traceability is ensured.

4.8.1 Refinement

In the UOOR approach, the development process consists of refinement steps: elicited requirements are refined into OO requirements that are further refined to implementation code. The detailed refinement process consists of the following steps:

- Component requirements in natural language are refined to deferred classes in Eiffel that constitute the Object model of the system and its environment. The hyperlink in a requirements document links a component requirement with the respective class. The EIS note links the class a with a component requirement in a requirements document.
- Environment constraints are formulated as contracts in classes, implementing the environment components. The EIS note links environment constraint with the respective feature in the object model.
- Environment constraints are further refined into natural language functional requirements and constraints, which are added to a natural language requirements document and as assertions in OO requirements classes, describing related components.
- Implementation classes inherit from requirements classes and provide the implementation of the deferred features. Due to inheritance mechanism, they must satisfy the contracts formulated in their ancestor requirements classes.

4.8.2 Traceability links

In order to ensure two-way traceability, a requirements engineer should create traceability links of two types: from NL document to code artifacts and from code artifacts to NL document.

Links that connect NL requirements artifacts with code artifacts appear in NL documents as hyperlinks. Such hyperlinks connect:

- Each component requirement to a respective OO component.
- Each functional requirement and related constraints to a respective feature.
- Each environment property to a related environment component.

The corresponding reverse links, which connect code artifacts with NL requirements, appear in the code as annotations (the technical details of creating links in EiffelStudio are presented in section 6.2; section 6.4 presents link management facilities, provided by the Traceability tool, supporting the UOOR approach).

4.9 System verification and requirements reuse

In the UOOR approach requirements are integrated into the entire software development process. UOOR provides means for verifying the solution against the requirements (section 4.9.1). Moreover, requirements can be reused in future projects as libraries of domain-specific components specifications (section 4.9.2).

4.9.1 System verification

UOOR requirements enable both static and dynamic verification [46] of the implemented system.

Since OO requirements are code elements, an IDE provides basic consistency checks at compile time. When contract checking is enabled at runtime, the IDE monitors contracts violations. Since every contract can have a unique tag, a developer can trace an exception to the violated contract.

Contract specification serves as oracles for dynamic testing. In addition, scenarios, implemented as specification drivers, serve as tests when passed actual arguments.

A static verifier, such as Autoproof [122], can be used to ensure static verification of system's functional correctness.

Verifiability of a requirement refers to the ease of checking that the constructed system meets the requirement. Since in UOOR requirements are captured by contracts, EiffelStudio provides the following mechanisms for requirements verification:

- When contract checking is enabled at runtime, the IDE monitors contracts violations. Since every assertion can have a unique tag, a developer can trace an exception to the violated assertion.
- Contracts serve as test oracles, which simplifies producing tests.
- Since requirements are compilable software elements, the IDE provides basic consistency checks such as type checking.
- Autoproof [122] provides static verification facilities for contracted Eiffel programs. This tool checks whether the implementation satisfies its contracts. If some of the contracts may be violated, Autoproof outputs warning, pointing at the contract that may be violated (see fig. 4.3).

Feature	Line	Result
BOOK (invariant admissibility)		Successfully verified.
BOOK.make (creator)		Successfully verified.
BOOK.make		Successfully verified.
BOOK.reserve	25	Postcondition reserved_book_is_on_hold may be violated.

Figure 4.3: Autoproof output for the incorrect implementation of the class BOOK.

4.9.2 Requirement artifacts reusability

Based on the capabilities, provided by the object-oriented technology, requirements obtained using the UOOR approach can be reused not but copy-pasting

natural language texts, but as libraries of domain-specific components specifications in the form of contracted deferred software classes. Being implementation-independent, such specifications support different implementations.

UOOR organizes requirements around classes, which are abstractions of the objects in the application domain. Such abstractions can be organized into libraries and further shared between several projects. Thanks to genericity, requirements can be abstracted to generic modules, whose parameters represent types. The example of such a generic module is CATALOG [CATALOG_ITEM], abstracted from a BOOK_CATALOG of the library case study, with such operations as "has item?", "add item", "remove item" (listing 4.11).

```
deferred class
    CATALOG [CATALOG_ITEM]

feature

    count: INTEGER
      -- Number of elements in catalog

    is_empty: BOOLEAN
      -- Is catalog empty?
        deferred
        ensure
            Result = (count = 0)
        end

    has (el: CATALOG_ITEM): BOOLEAN
      -- Is el an element of catalog?
        deferred
        ensure
            not_found_in_empty: Result implies not is_empty
        end

    add (element: CATALOG_ITEM)
      -- Add a new element to catalog
        deferred
        ensure
            count = 1 + old count
        end

    remove (element: CATALOG_ITEM)
      -- Remove element from catalog
        deferred
        ensure
            count = old count - 1
        end
```

```
end
```

Listing 4.11: Example of a reusable requirements class.

Such a class defines basic operations, applicable to objects of type catalog, but stays free from implementation details. Due to genericity, the class can be used for catalogs of different object types, not necessarily books or library items.

4.10 Conclusion

This chapter introduced the Unified Object-Oriented approach to requirements. It demonstrated that the concept of class is general enough to describe not only “objects” in a narrow sense but also scenarios such as use cases and user stories. Further it provided a methodology of the UOOR approach, which a requirements engineer may use as a guide to requirements specification process.

UOOR relies on requirements expressed in a NL form. These requirements, however, are not assumed to be fixed: in the process of refinement, the NL document is updated when needed so that it reflects the latest version of the requirements.

- The first refinement step is drafting OO model based on component requirements and environment properties. This model expresses components of the system and its environment as abstract classes. The properties of environment are expressed as contracts (preconditions, postconditions and class invariants) of the environment classes.

- The second refinement step enriches the OO model with the functional specification of the system. Desired system functions are expressed as the features of the system classes; contracts express the constraints on the system features.

- The next refinement step enriches the OO model with behavioral specification. The behavioral specification is both abstract and concrete. Abstract properties of system’s functions (such as time-ordering constraints) are extracted from scenarios and added to the OO model. Concrete examples of scenario sequences are expressed as specification drivers (contracted routines of scenario classes) and serve verification purposes.

- The implementation process relies on refinement: implementation classes inherit from requirements classes and thus must satisfy the requirements expressed as contracts.

- Two-way clickable traceability links connect NL requirements (component requirements, functional requirements, constraints and scenarios) with the related code artifacts.

The next chapter will introduce the notion of seamless requirements traceability, which facilitates creation and management of requirements traceability links.

Chapter 5

Project elements traceability

Requirements traceability, which we define as the ability to follow both the sources and consequences of requirements [81], remains an important issue of software engineering. An industrial survey conducted in 2015 by Fricker et al. [36] evaluated the frequency of use of various requirements engineering techniques in industrial projects; traceability scored only 40%. Typical industrial practice is creating and maintaining requirements traceability matrices [69]. As traceability matrices are created and maintained manually, they require substantial manual effort yet still often store incomplete or incorrect data as the project evolves [25]. Automated requirements traceability approaches address this issue by relying on information retrieval. It significantly decreases the burden of creating the traceability links, but the level of precision is very low (10-35%) which means that the majority of created links are incorrect. The latest advances in automated traceability provide better results, yet still the precision is limited: only 33-43% of retrieved links are correct [124].

This chapter introduces the notion of seamless requirements traceability. It demonstrates that seamless software development can facilitate creation and management of requirements traceability links by exploiting the formal properties of relations between project artifacts and propagating traceability links based on these formal properties.

To introduce seamless requirements traceability, this chapter presents a seamless requirements traceability model, which is an abstraction of project elements and relations between them. The relations between project elements are determined with informal definitions accompanied with formal mathematical properties of those relations. Formal properties of the relations serve as the basis for relations propagation.

5.1 Seamless requirements traceability

As it was established in section 3.1, requirements traceability is the ability to follow both the sources and consequences of requirements. In other words, traceability links connect requirements with their sources and with related project elements. In practice, the navigation between the project elements,

connected with a traceability link, can have different nature.

On one end are manually created traceability matrices, which establish relations between requirements and related test cases in a tabular form. Such matrices are created and updated manually, requirements and test cases are added by copy-pasting from the related documents. Consequently, creating and maintaining such matrices requires substantial efforts. At the same time, they do not provide direct navigation from a requirement to a test case, they merely establish a visual representation of correspondence between requirements and related test cases; moreover they usually link requirements only to test cases, not to the other project artifacts.

Dedicated requirements management tools (such as Polarion [112] and IBM DOORS [48]) significantly decrease the requirements traceability effort. They allow creating clickable links between requirements and other project artifacts (for example, by drag-and-drop). In some cases it requires integration between several applications since project elements are not handled in a single application.

Seamless requirements traceability goes even further. In addition to supporting the direct links between requirements and other project artifacts, it relies on relation propagation, which is based on formal properties of relations. Seamless requirements traceability can be characterized by three fundamental properties:

- Requirements are directly connected with other project artifacts with clickable links.
- A substantial number of links is propagated rather than created manually.
- Changes in requirements can be traced to the related project artifacts.

5.2 UOOR traceability information model

Traceability information model describes the relation between software project artifacts [22]. The basic traceablity model has been established by the members of the Agile Project Management Forum [26] (see fig. 5.1). This model reflects the common practice of agile projects by covering the most frequent tracing scenarios.

The agile traceability information model includes three main blocks: requirements, tests (that are part of a test suite) and implementation. The relation of requirements (user stories) and implementation is not direct: requirements are linked with test cases by external tools or by annotating test cases with user story ID; the relation of code to tests (and further to requirements) is implicit: we know that code implements the requirements if it passes the tests, yet there is no direct link from code to requirements.

Seamless development can extend this model by adding OO requirements, which link natural language requirements to tests and implementation code. In a coarse grained view the model looks as presented in fig. 5.2.

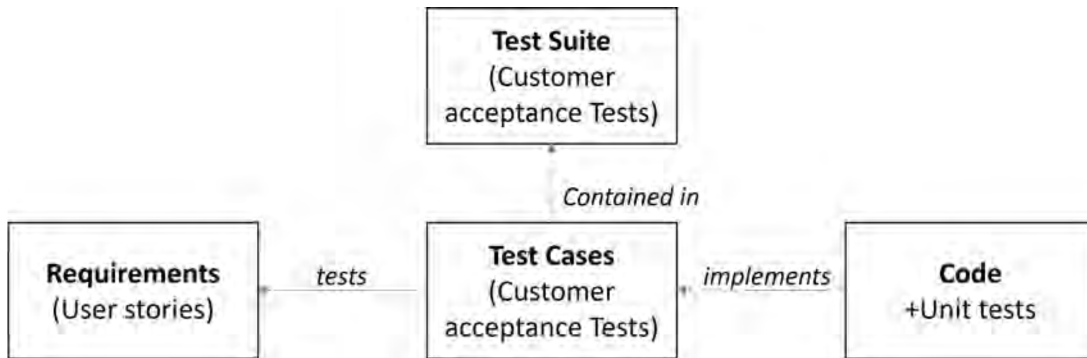

Figure 5.1: Agile traceability information model.

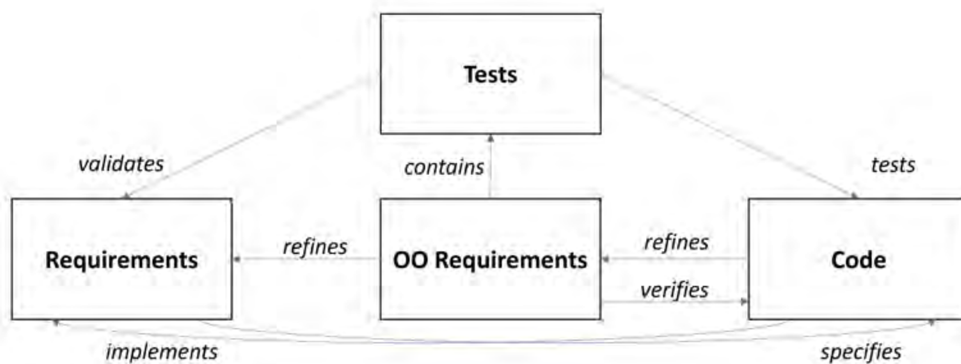

Figure 5.2: UOOR traceability information model,

Due to seamless requirements traceability, in the UOOR traceability information model requirements are linked directly to the implementation and test artifacts. In order to provide a more fine-grained overview of project elements and relations between them, sections 5.3-5.4 define the main types of project elements and relations between them.

5.3 Project elements

Project elements are the basic blocks of the model of requirements engineering activity. Essentially project elements are types of artifacts produced in the course of software development project. It is important to identify and define project elements types for the following reasons:

- To consider possible implementation of project elements (such as, for example, a class or a section of a document).
- To make sure that requirements can be traced to certain project artifacts (such as implementation or test artifacts).
- To introduce type-dependent propagation rules.

As presented in fig. 5.2, project artifacts belong to one of four general types: natural language requirements, OO requirements, tests and implementation. Sections 5.3.1-5.3.4 provide the definitions of these four types and of project elements belonging to each of these groups.

5.3.1 Natural language requirements

Definition 5.1. Natural language requirements are requirements expressed in a textual form.

Natural language requirements are documented in the majority of software projects [36]. For the sake of further traceability analysis, we assume that each natural language requirement is a bookmarked section of a document.

To provide more fine-grained traceability, several sub-types of natural language requirements can be identified.

Definition 5.2. A **component requirement** is a statement that the system or its environment contains a certain part.

In large systems requirements are often grouped by the respective system components. In that case, if a component requirement is not explicit, the title of a subsection, describing component requirements, serves as a component meta-requirement.

Definition 5.3. A **functional requirement** is a statement that the system shall provide a certain functionality.

Example 1. A "shall statement" or a user story are typical examples of functional requirements.

Definition 5.4. A **constraint** is a property that restricts the behavior of a system's function.

Whereas a functional requirement describes what the system must do (such as *"provide the ability to place hold on books"*), a constraint expresses the limitations imposed on the system's function (such as *"Patrons who have not paid their penalty fee cannot borrow books"*).

Definition 5.5. A **scenario** is a description of interaction of an actor with the system to reach a certain objective.

Example 2. Typical examples of scenarios are use cases and use case stories.

Scenarios are more detailed than functional requirements, since they describe steps of actor's interaction with the system. In addition, scenarios are less abstract than functional requirements and constraints, since they describe examples of interaction, but not all the possible sequences.

5.3.2 OO requirements

Definition 5.6. Object-oriented (OO) requirements are requirements expressed in a programming language (Eiffel in the context of this thesis).

Similarly to natural language requirements, OO requirements can be divided into several subtypes.

Definition 5.7. OO component is an abstract representation of a component of a system or its environment in the form of a class.

Example 3. A class `BOOK` is a typical example of an OO component of the environment in the Library Management System case study.

Definition 5.8. OO functional requirement is a feature of a requirements class that expresses functionality, provided by the component.

Example 4. The feature `reserve_a_book` of a requirements class `LIBRARY` is an OO functional requirement.

Definition 5.9. OO constraint is an assertion specifying an abstract property of a feature or of a class.

Example 5. The postcondition "**old** p.num_reserved $\geq$ 5 **implies** (b.is_available **and** p.num_reserved = **old** p.num_reserved)" of the feature `reserve_a_book` of a requirements class `LIBRARY` is an OO constraint.

Similarly to natural language functional requirements and constraints, the difference between OO requirements and OO constraints is the following: OO functional requirement is simply a name of a feature that will implement the required service, provided by the objects of a given type. OO constraint is the definition of a feature, expressed as its contract (set of assertions).

Definition 5.10. OO Scenario is an abstract representation of a scenario in the form of a routine, exercising the features of implementation classes.

Example 6. Feature `reserve_book_successfully`, presented in section 4.7 is an example of OO scenario.

5.3.3 Tests

Definition 5.11. Tests are programming language artifacts used to test part of the functionality of a software system.

Definition 5.12. Test case is a single testing scenario.

Definition 5.13. Test suite is a set of test cases covering one or more classes or features.

5.3.4 Implementation

Definition 5.14. Implementation artifacts are code artifacts that are part of the developed system.

Definition 5.15. Implementation component is a class or a cluster (a group of classes) of the implemented system.

Definition 5.16. Implementation feature is a feature of the implemented system.

The summary of project element types and their definitions is presented in Table 5.1.

5.4 Relations between project elements

The second key element of the model of requirements engineering activity is the relations between project elements. Typed relations simplify traceability analysis: when assessing the impact of change of a given requirement, selecting links of a certain type will yield the elements that have a certain relation to the requirement. Furthermore, it is possible to introduce formal properties of the relations, which can serve as a basis for relation propagation.

This section does not attempt to cover all possible types of relations between project elements. Instead, it introduces some basic types and explores formal properties that can be expressed for these relations.

For each relation (except for the reflexive ones) an inverse relation will be defined. Inverse relations are particularly useful for propagating two-way traceability. The summary of project relations and their properties is presented in fig. 5.2.

Project element type	Definition	Possible implementation
Component requirement	A statement that the system or its environment contains a certain part	Bookmarked section of a document
Functional requirement	A statement that the system shall provide a certain functionality	Bookmarked section of a document
Constraint	A property that restricts the behavior of a system's function	Bookmarked section of a document
Scenario	A description of interaction of an actor with the system to reach a certain objective	Bookmarked section of a document
OO component	An abstract representation of a component of a system or its environment in the form of a class	Class
OO functional requirement	A feature of a requirements class that realizes a functional requirement	Feature
OO constraint	An assertion specifying an abstract property of a feature or of a class	One or more pre- or post- conditions or class invariants
OO scenario	An abstract representation of a scenario in the form of a routine, exercising the routines of implementation classes	One or more specification drivers
Test case	A single testing scenario	Feature
Test suite	A set of tests covering one or more classes or features	Class or cluster
Implementation component	A class or cluster of the implemented system	Class
Implementation feature	A feature of the implemented system	Feature

Table 5.1: Project element types.

5.4.1 Repeats

Quality requirements documents should not contain repeating requirements, yet in practice this situation is possible. Moreover, one requirement can be expressed in two different notations (for example, natural language and graphical) - this is also the case when the two elements will be linked with the "repeats" relation. While repetition is possible (and is sometimes intentional) in NL documents, it should be avoided in the programming language artifacts, so the "repeats" relation links only natural language requirements artifacts.

Definition 5.17. A requirement A **repeats** a requirement B iff anything which is described by A is also described by B, and anything which is described by B is also described by A.

***Property* 5.1.** *A repeats B* $\implies$ *B repeats A*

***Property* 5.2.** *A repeats B; B* R_1 *C* $\implies$ *A* R_1 *C (where* R_1 *is some relation).*

***Property* 5.3.** *A repeats B; A* R_2 *D* $\implies$ *B* R_2 *D (where* R_2 *is some relation).*

In other words, properties 5.2-5.3 state that if B has a relation of type R_1 with some element C, then A should also have a relation of type R_1 with element C. And in reverse, if A has a relation of type R_2 with some element D, then B should also have a relation of type R_2 with element D. So, if A repeats B, they have the same set or relations.

5.4.2 Complements

Definition 5.18. A **complements** B iff A and B cooperate towards the achievement of some higher aim.

***Property* 5.4.** *A complements B* $\implies$ *B complements A*

5.4.3 Constrains and is constrained by

Definition 5.19. A requirement A **constrains** another requirement B iff it states a condition that B must satisfy.

Definition 5.20. A requirement B **is constrained by** another requirement A iff A constrains B.

The "constrains" relation links constraints with functional requirements.

Example 7. The constraint *"If a patron has five holds, placing a hold is not successful, and the book remains available"* constrains the functional requirement *"The Library Management System shall provide the ability to place hold on books".*

***Property* 5.5.** $constrains^{-1} = is_constrained_by$

***Property* 5.6.** $is_constrained_by^{-1} = constrains$

Properties 5.5, 5.6 follow from the definitions of relations constrains and is_constrained_by. Formulating them as properties enables propagation of the inverse relations (see section 5.5).

***Property* 5.7.** *A constrains B; C constrains B* $\implies$ *A complements C*

5.4.4 Refines and generalizes

Definition 5.21. A **refines** B iff anything that is described by A is also described by B (some additional details may be provided by B).

Refinement is one of the key relations in the UOOR approach, since natural language requirements convert into implementation through a series of refinement steps. The examples of the "refine" relations are the following:

- An OO requirement A refines a natural language requirement B, if A expresses B in a programming language.

  Example 8. OO functional requirement expressed as the feature `reserve_a_book` of a requirements class `LIBRARY` is a refinement of the functional requirement "The Library Management System shall provide the ability to place hold on books".

  Example 9. OO component implemented as a class "BOOK" is a refinement of the environment component "book" identified in the LMS requirements document.

- Class B refines class A if B inherits from A. In that case all inherited features of B will have a "refines" relation with the respective features of A.

  Example 10. OO component `RESEARCH_PATRON`, inheriting from the class `PATRON`, refines the OO component `PATRON`.

  Example 11. Feature `place_hold` of the class `RESEARCH_PATRON` refines the OO functional requirement, expressed by the feature `place_hold` of the class `PATRON`.

Definition 5.22. B **generalizes** A iff A refines B.

Example 12. OO component `PATRON` generalizes the OO component `RESEARCH_PATRON`, since the OO componet `RESEARCH_PATRON`, refines the OO component `PATRON`.

Example 13. OO functional requirements expressed by the feature `place_hold` of the class `PATRON` generalizes the OO functional requirement, expressed by the feature `place_hold` of the class `RESEARCH_PATRON`.

***Property* 5.8.** *(refines; refines) - id* $\subseteq$ *refines*

In other words, the property 5.8 declares that refines is a transitive relation.

***Property* 5.9.** *inherits* $\subseteq$ *refines*

Indeed, if class A inherits from class B, all assertions of class B (that define the semantics of operations) are also inherited and must be respected.

Property **5.10.** $refines^{-1} = generalizes$

Property **5.11.** $generalizes^{-1} = refines$

Properties 5.10, 5.11 follow from the definitions of the relations refines and generalizes.

5.4.5 Implements and specifies

Definition 5.23. A program element B **implements** a requirement A, iff it realizes the functionality specified in A.

The "implements" relation links a natural language requirement with the related part of the source code.

Definition 5.24. A requirement A **specifies** a program element B iff B implements A.

Property **5.12.** $implements^{-1} = specifies$

Property **5.13.** $specifies^{-1} = implements$

Properties 5.12, 5.13 follow from the definitions of the relations "implements" and "specifies".

Property **5.14.** *A refines B* $\Longrightarrow$ *A implements B (where: A is an implementation artifact; B is a natural language requirement).*

The property 5.14, together with the property 5.8 state that a requirement can be traced to the respective implementation artifact via a sequence of refinement steps.

5.4.6 Contains and is a part of

Definition 5.25. A **contains** B iff B is a constituent of A.

Definition 5.26. B **is a part of** A iff A contains B.

Example 14. Test suite A contains test cases B, C and D.

Example 15. Class `PATRON` contains a feature `place_hold`.

Property **5.15.** $contains^{-1} = part_of$

Property **5.16.** $part_of^{-1} = contains$

Property **5.17.** $(part_of; contains)$ - $id \subseteq complements$

Property **5.18.** $(contains; contains)$ - $id \subseteq contains$

5.4.7 Tests and is tested by

Definition 5.27. Test A tests implementation artifact B iff passing A is required (but possibly not sufficient) for B to be considered implemented correctly.

***Property* 5.19.** $tests^{-1} = is_tested_by$

***Property* 5.20.** $is_tested_by^{-1} = tests$

***Property* 5.21.** *A tests B, C tests B* $\Longrightarrow$ *A complements C*

5.4.8 Validates and is validated by

Definition 5.28. Test artifact A validates requirement B iff passing A is required for B to be considered implemented correctly.

***Property* 5.22.** $validates^{-1} = is_validated_by$

***Property* 5.23.** $is_validated_by^{-1} = validates$

***Property* 5.24.** *A tests B, B refines C* $\Longrightarrow$ *A validates C (where: A is a test artifact; C is a natural language requirement).*

5.4.9 Refers to

Definition 5.29. A refers to B iff A refers to B by its name.

***Property* 5.25.** $Is_a_client \subseteq refers_to$

***Property* 5.26.** $(refers_to; refers_to)$ - $id \subseteq refers_to$

Table 5.2 presents the summary of relations' definitions and formal properties. Figure 5.3 presents project elements and some of the relations between them (some of the relations are not displayed for the sake of clarity).

5.5 Relations propagation

Requirements traceability is *"the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)"* [42]. The ability to trace a requirement to implementation and testing artifacts eases the process of validating delivered software against requirements and ensures the adaptability of software product to requirement changes. Manual establishment of traceability links, however, is

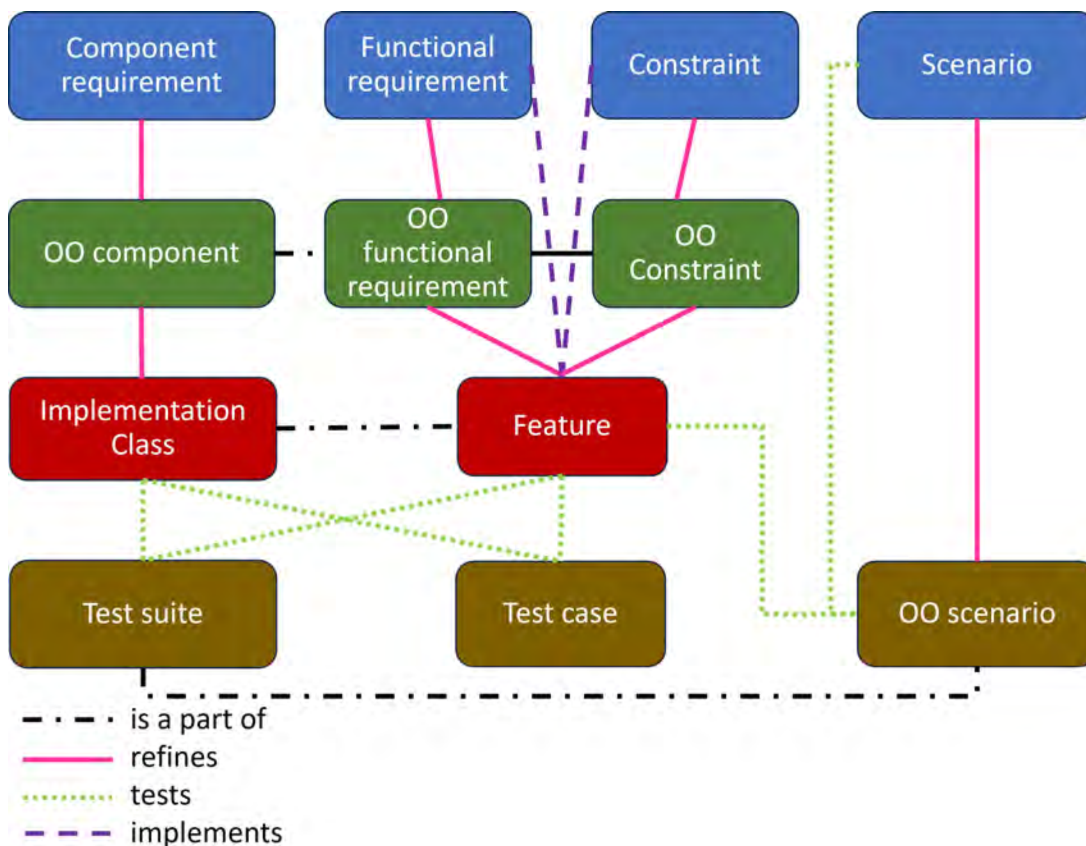

Figure 5.3: Key project elements and relations.

tedious and time consuming. Seamless software development may address this issue by providing the link propagation, so that the links are inferred from the initially created links, project element types and formal properties of relations between project elements.

To illustrate the idea, let's consider the "refine" relation. In seamless software development a requirements turns into implementation as the result of a sequence of refinement steps. The first step - linking a natural language requirement with its OO implementation - is manual, yet similarly to the leading requirements management tools in the field, such a link can be established in a few clicks. OO requirement is further linked with the implementation artifact via inheritance: a class, implementing a requirement, inherits from the requirements class. Not necessarily this relation is direct: in fact, there may exist several layers of inheritance. It is possible, however, to extract the ancestors for implementation artifacts: for each class it is possible extract its ancestors and proper ancestors; for each feature it is possible to identify, in which ancestor class it was introduced. This way, it is possible to propagate the refinement links based on their formal properties (transitivity).

When all the "refine" links are propagated for a given requirement, the "implements" relation can be inferred: if a natural language requirement and an implementation artifact are linked with the "refines" relation, they have to

be linked with the “implements” relation.

5.6 Conclusion

This chapter introduced the notion of seamless requirements traceability. It demonstrated that by modeling requirements engineering activity as a set of project elements connected with typed links, it is possible to propagate traceability links based on formal properties of relations. As the same time, if it is possible to propagate traceability links between requirements and related implementation code, it is possible to track changes in requirements to the related code artifacts.

Seamless requirements traceability, while a theoretical concept, requires tool support: as it was established before, adequate tool support is one of the key issues in requirements traceability. Current functionality of EiffeStudio, however, does not support typed project elements and relations. The next chapter will present the Traceability tool, which serves as a proof of concept for seamless requirements traceability management.

Relation	Element types	Definition and formal properties
Repeats	A: ELEM B: ELEM	A repeats B iff anything which is described by A is also described by B, and anything which is described by B is also described by A. A repeats B $\implies$ B repeats A A repeats B; B R_1 C $\implies$ A R_1 C A repeats B; A R_2 D $\implies$ B R_2 D (where R_1, R_2 are some relations).
Complements	A: ELEM B: like A	A complements B iff A and B cooperate towards the achievement of some higher aim. A complements B $\implies$ B complements A
Constrains	A: CSTR B: FR	Requirement A constrains another requirement B iff it states a condition that B must satisfy. constrains^{-1} = is_constrained_by A constrains B, C constrains B $\implies$ A complements C
Refines	A: ELEM B: ELEM	A refines B iff anything which is described by A is also described by B (some additional details may be provided by B). (refines; refines) - id $\subseteq$ refines inherits $\subseteq$ refines refines^{-1}= generalizes; generalizes^{-1}= refines
Implements	A: NLRQ B: IM	An implementation feature or implementation component A implements a requirement B, if it provides the functionality or constraint stated in B. implements^{-1} = specifies; specifies^{-1} = implements A refines B $\implies$ A implements B (where: A is implementation feature or component; B is requirement or constraint)
Contains	A: ELEM B: ELEM	A contains B iff B is a constituent of A. contains^{-1} = part_of; part_of^{-1} = contains (part-of; contains) - id $\subseteq$ complements; (contains; contains) - id $\subseteq$ contains
Tests	A: TEST B: ELEM	A tests B iff passing A is required (but possibly not sufficient) for B to be considered implemented correctly tests^{-1} = is_tested_by A tests B, C tests B $\implies$ A complements C
Validates	A: TEST B: NLRQ	A validates B iff passing A is required for B to be considered implemented correctly validates^{-1} = is_validated_by A tests B, B refines C $\implies$ A validates C (where: A is a test artifact; C is a NL requirement)
Refers to	A: ELEM B: NLRQ	A refers to B iff A refers to B by its name. Is_a_client $\subseteq$ refers_to (refers_to; refers_to) - id $\subseteq$ refers_to

Table 5.2: Relations between project elements and their properties.

Chapter 6

UOOR traceability tool

6.1 Introduction

A recent systematic study extracted the key challenges related to requirements traceability [70]:

- Inadequate tool support.
- High cost of maintaining traceability.
- Late update of the information.
- Benefits of traceability not recognized.

These challenges can be addressed by seamless requirements traceability. Indeed, one of the key benefits of seamless software development is the ability to facilitate traceability between requirements and other project artifacts, such as implementation and tests. This ability, however, requires adequate tool support. To exploit all benefits of seamlessness, the following functionality is required:

- To link code artifacts (features and classes) with external documents.
- To receive notifications of changes in requirements and to trace from requirements to related project artifacts (implementation and tests).
- To create typed relations between project elements.
- To infer traceability links base on the formal properties of the relations.

The UOOR Traceability tool [43] serves as a proof of concept for traceability links management in the UOOR approach[1]. The tool is integrated into the EiffelStudio and enhances its functionality in the following ways:

- Assigning types to project elements.

[1]We thank Zakaria Hachm, internship student from ENSIAS, Rabat, Morocco, for the implementation of this tool

- Creating typed links between project elements.
- Displaying related links for a given project element.
- Displaying notifications of changes in requirements for a given project element.

6.2 Existing Tools and Technologies

Over 200 requirements engineering tools are present in the market, so by no means this section intends to cover all of them. A comprehensive evaluation of the most used tools is presented in [24]. The goal of this section is to evaluate the traceability capabilities of the tools that have the most comprehensive traceability support and of the tools that support seamless traceability.

IBM DOORS (Dynamic Object-Oriented Requirements System) is a well-known requirements management tool, which supports direct linking of requirements with other project artifacts [48]. The Traceability Explorer in DOORS helps visualize which objects have links, navigate to linked objects and perform impact analysis based on those links. Impact analysis can be performed for DOORS artifacts and does not include the relations between requirements and code. Although DOORS is a mature tool with rich functionality, it is more suitable for large-scale enterprises due to its complexity and cost.

Jama Connect [116] is a web application that handles requirements management and testing. The "Live Traceability" links requirements across the entire development lifecycle and supports the maintenance of change through a change management process. Traceability links can be established between requirements of various levels and tests, but not to the source code.

The Systems Modeling Language (SysML) [97] defines a list of relationships between requirements and other project elements. **Enterprise Architect** [119], a widely used requirements management tool supporting SysML, allows engineers to create traceability links based on the SysML relationship types. Based on the established links, the tool can render relationship matrix, which visualizes relations between requirements and other elements (including requirements). Enterprise Architect establishes relations between requirements and design artifacts, but does not link the requirements with the source code.

Polarion REQUIREMENTS[112] is a web-based tool that supports the creation of traceability links between all types of project artifact. The app provides a list of predefined link types, which can be extended by a user. When adding a link type, a user can define a list of rules applicable to links of that type. The rules describe permissible relations between different types of project artifacts. The tool does not have the ability of links propagation, so each link has to be created manually.

EiffelStudio has a built-in tool, named Eiffel Information System (EIS), which provides facilities to create and maintain links with artifacts outside of the IDE. EIS supports two-way links:

- referencing a class or a particular feature of a system's class from external applications.
- referencing a IDE or a document at a given bookmark from a class or a particular feature of a system's class.

To refer to a feature `place_hold` of a class PATRON, one can use the following URI: `eiffel:?cluster=project&class=PATRON&feature=place_hold`. By entering the URI into the address field of a browser, or clicking on the hyperlink, an existing EiffelStudio instance or new EiffelStudio instance will try to resolve the URI and display corresponding resources. Since specification drivers are routines of specification classes, they can be referenced directly. Contracts can be traced to the corresponding procedures (pre-, and postconditions) or to a corresponding class (class invariants).

To reference an external resource from Eiffel Studio, the annotations should be added to a class text. The annotation appears in the note part of a class (fig. 6.1) or of a feature (fig. 6.2) definition, as in the examples below.

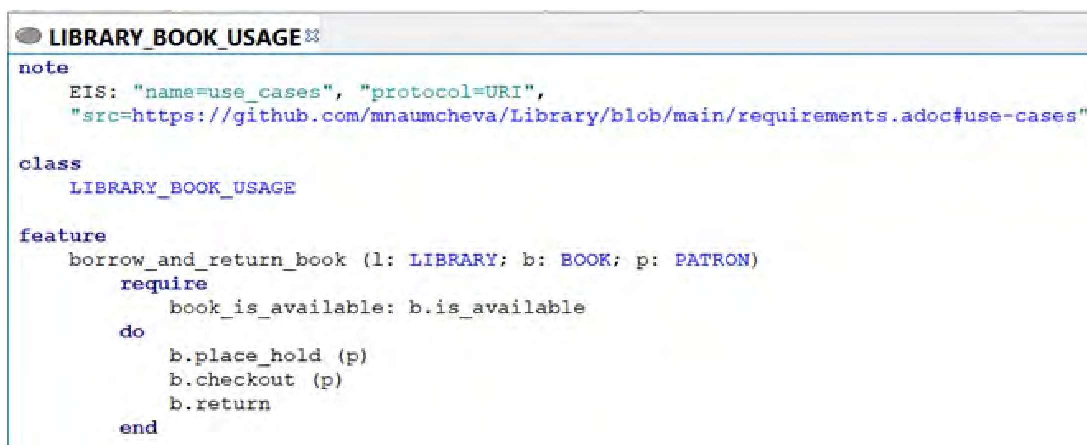

Figure 6.1: Referencing requirements document in EIS from a class.

Figure 6.2: Referencing requirements document in EIS from a class feature.

The functionality of EiffelStudio is not sufficient to exploit the benefits of seamless traceability, which can be established in a project following seamless

software development paradigm. In particular, in Eiffelstudio each traceability link has to be created manually which significantly increases the burden of traceability maintenance. Furthermore, EiffelStudio does not support typed traceability links.

SIRCOD tool [38] is a tool prototype developed to support the SIRCOD requirements approach within EiffelStudio. The tool enhances the capabilities of EiffelStudio in two ways:

- Automated bookmarks extraction from a Word document.
- Documentation view for system classes.

In SIRCOD, the natural language version of each requirement is stored as a feature of a requirements class (fig. 6.3).

```
requirement_1: STRING

    note
        EIS: "protocol=DOCX", "src=C:\Documents\LMS Requirements document.docx",
             "bookmark=If_a_patron_has_less_than_five_holds_pl"
    do
        Result := "If a patron has less than five holds, placing a hold is successful"
    end
```

Figure 6.3: Requirement in SIRCOD.

Automated bookmarks extraction assigns a bookmark to every paragraph of a word document so that a requirements engineer can easily create a link between code artifact and a related requirement (assuming that each requirement belongs to a separate paragraph of a requirements document).

Documentation view displays requirements in natural language associated with the Eiffel class in order to facilitate maintaining consistency between requirements and implementation (fig. 6.4).

```
feature

    drone_behaviour_doc: STRING_8
    note
            EIS: C:\Users\zakar\testt00.docx
        doc: the drone shall pick up a parcel, go to destination and drop the parcel and test the result.
    end
                note
                    eis: name=Unnamed, protocol=URI, src=C:\Users\zakar\testt00.docx, bookmarks=Titre 2
                    doc: true

end -- class APPLICATION
```

Figure 6.4: Documentation view.

To summarize, a number of mature tools for traceability links creation and management exist in the market, yet they do not provide seamless traceability from requirements to code. The SIRCOD tool, supporting seamless requirements, enhances the capabilities of EiffelStudio with documentation view and

automated bookmarks extraction, yet it does not provide the ability to create typed links and propagate traceability links based on formal properties of relations.d

6.3 Traceability tool interface

The Traceability tool is built on top of the Eiffel Information System (EIS), so it extends the EIS panel with several buttons: “Traceability links”, “Add link”, “Delete link” and “Track changes”. The ”Traceability links” button is the one a user has to press in order to access the Traceability tool: after pressing the button a user can see the traceability links table and manage the links (add, edit, remove, track changes). Figure 6.5 is a screenshot of the tool interface showing a traceability grid and the tool buttons.

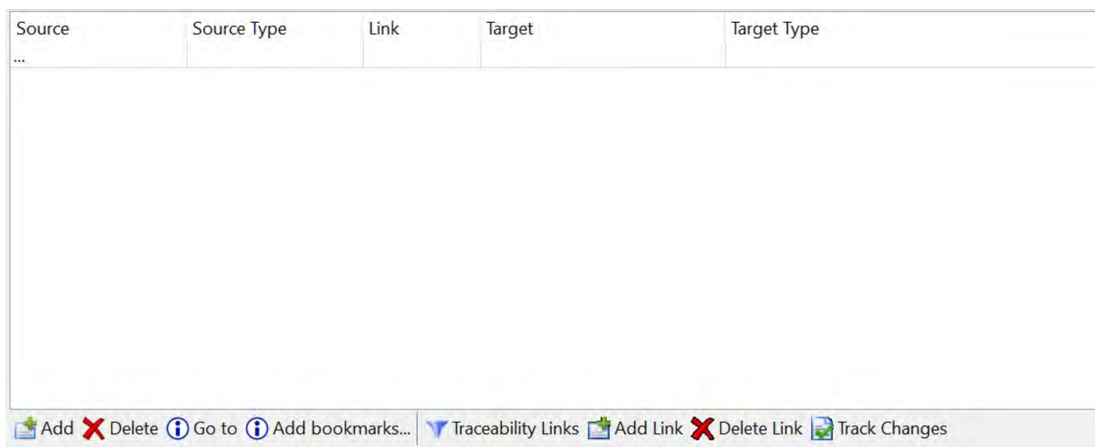

Figure 6.5: Traceability tool interface.

6.4 Link management

The tool provides the following link management functionalities:

- Creating a link.
- Viewing links.
- Editing a link.
- Deleting links.

Adding a link. When we click to add a new traceability link, the interface allows us to select various elements for establishing the connection. Specifically, we can choose the source, the link type, the source & target types and then define the target of the link.

For the target, there are two possible options(fig. 6.6): “Select Class/Feature (for creating a link to internal project artifact) and “Select Bookmark” (for creating a link to an external document).

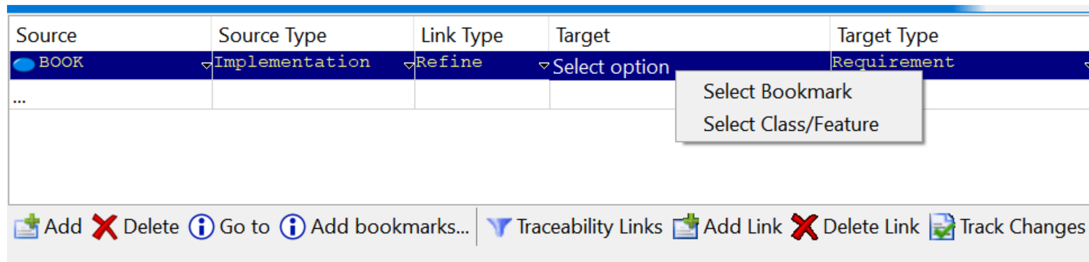

Figure 6.6: Option Selection

- **Internal Project Artifact:** This option allows us to select another component within the project. We can choose from all available features and classes inside the project (fig. 6.7, 6.8).

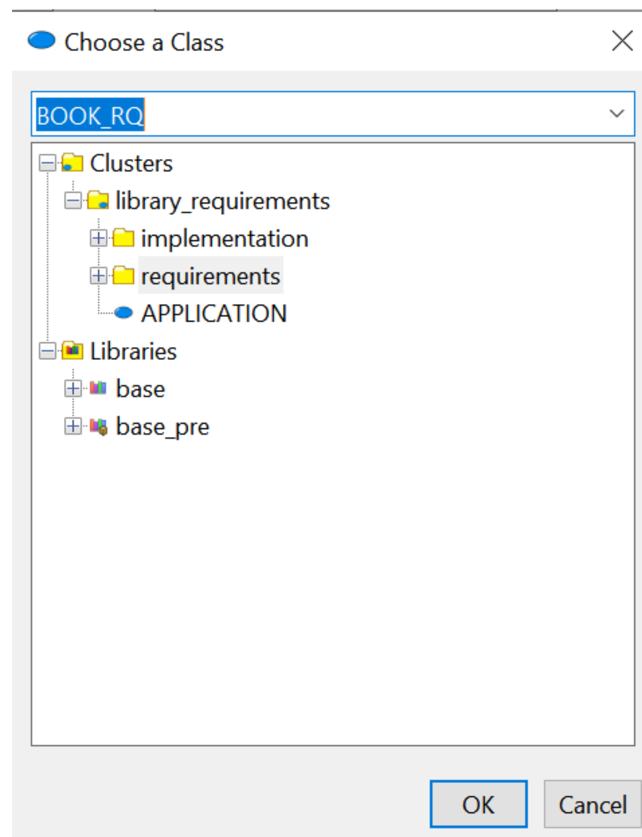

Figure 6.7: Class selector

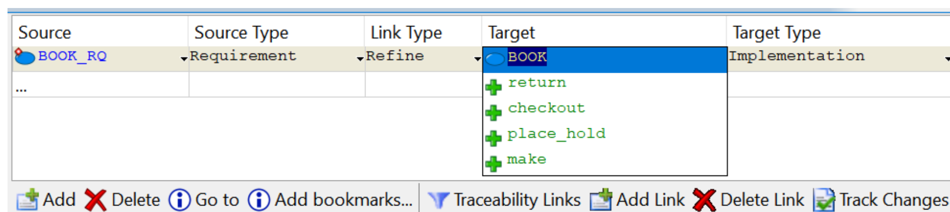

Figure 6.8: Feature selector.

- **External Document or Specific Requirement:** Alternatively, it is possible to create a link to an external document or a specific requirement by selecting a bookmark within that document.

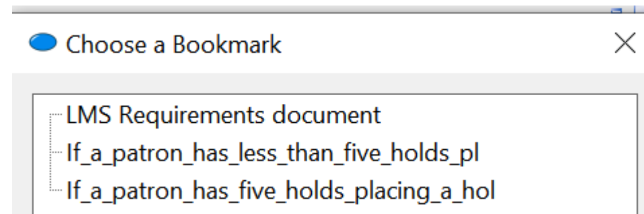

Figure 6.9: Selecting Bookmark.

Once a traceability link is created, we can visualize it directly within the code. The tool provides a dedicated interface to view and manage these links.

Viewing Links. Within the class code each link appears as a note with several parameters, including the path to the linked element, types of project element and the link type (fig. 6.10). Links can be modified within the class code, and after compilation the tool will display the modified links. Similarly, all changes made within the tool will be reflected in the note after recompilation. The path, displayed in the note and the tool is clickable, and clicking it will open the referenced project element. The note can appear at the level of a class (to link a class with another project element) or at the level of a feature (to link a feature with another project element)

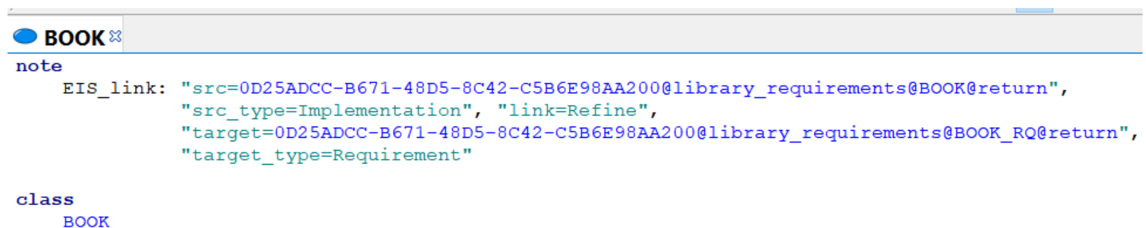

Figure 6.10: Links in the code.

Editing links. A link can be modified inside the tool by clicking at a cell of a traceability table that has to be modified. In particular, the type of the link, project element types and the path to the linked element can be modified this way. Any link can be deleted by selecting it and pressing the "Delete link" button. In order to store the information, the project should be recompiled.

6.5 Tracking changes

The important issue in managing requirements traceability is to identify how changes in requirements affect the source code. The Traceability tool addresses this issue by linking requirements in a requirements document with the related code elements and monitoring changes in text requirements.

To evaluate whether there has been a change in requirements, a developer should press the “Track changes” button. The tool will assess whether the requirements document has been changed and compare the content at each bookmark with the text of the previous version of the document. The outcome of the analysis is presented in a table, which links the modified requirements in a requirements document with the affected code elements (fig. 6.11). The table displays the path to requirements document, modification date, affected project element and modified requirement. Clicking on “Click do display” opens the new version of the modified requirement so that a developer could make relevant adjustments in the code.

External File	Modified At	Affected target	Affected Requirement
C:\Users\maryb\Des...	09/25/2024	place_hold	Click To Display

Add Delete Go to Add bookmarks... Traceability Links Add Link Delete Link Track Changes

Figure 6.11: Tracking changes.

6.6 Tool limitations and evaluation

The Traceability tool, developed for the UOOR approach, has demonstrated that it is feasible:

- To established typed links between software project elements within the IDE.
- To track changes in requirements and receive notifications that a requirement, related to a given code element, has been modified.

Being a proof of concept, this tool has a lot of room for improvement, in particular:

- Improving user interface by providing customizable dashboards at different levels of granularity.
- Implementing links propagation.
- Providing advanced notification system, integrated with communication tools (email, messengers).

6.7 Conclusion

This chapter presented a tool integrated into EiffelStudio that supports seamless requirements traceability by providing the ability to create typed links between project elements and to track changes from requirements to the affected

code elements. The tool demonstrates the advantages of seamless software development and enhances the usability of the UOOR approach.

The perspectives of the tool's development include implementing the propagation of links based on the formal properties of relations. It will allow to fully exploit the benefits of seamless requirements traceability.

Chapter 7

Evaluation of the UOOR approach

Any new methodological approach needs evaluation on practical cases. While we have not tried UOOR in a large setting, we conducted two experiments, one devoted to requirements analysis for an existing industry project, and the other focused on an academic course assessment.

Section 7.1 presents the application of the approach to a significant real-world case study to evaluate the expressiveness of the approach. Section 7.2 overviews the approach from the users' perspective and evaluates its usability and affordability.

7.1 Roborace case study

In software engineering research case study is accepted as a suitable empirical research methodology, except for small "toy examples", which lack real-life context [110]. With this in mind, we selected to apply the UOOR approach to a significant real-life project - the Roborace championship for autonomous cars [94].

The objective of the study is exploratory. It seeks the answers for the following research questions:

- Is UOOR approach expressive enough to formulate requirements for a significant project?
- Does the UOOR methodology facilitate requirements specification?

7.1.1 Case study design

The study was conducted in collaboration with the SIT Autonomous Team, which was one of the participants of the Roborace championship. The team principal participated in semi-structured requirements elicitation interviews and provided technical documentation. The author of the thesis produced requirements for Devbot's software, following the UOOR approach, based on the obtained information and information from open sources (such as publications of competing teams and Roborace press releases). The produced specifications were reviewed by the research team members.

Due to confidentiality restrictions, the internal documents and interviews are not disclosed to the public. The requirements rely on the publicly available information.

7.1.2 Roborace project overview

Roborace is a global championship between autonomous cars. The hardware (the race cars) is the same for all participating teams; each gets access to an autonomous race car called Devbot 2.0, and develops software to drive it in races, in a completely autonomous way. Each season sees changes in the goals and rules, and the introduction of new conditions.

The Roborace competition provides facilities for driverless technology development and testing. The project team aims to use this opportunity to develop software that can be licensed and sold. The hardware is developed and maintained by Roborace. The vehicle is fully ready for autonomous racing and technical specification is provided to the project team. Initially the software is partially provided by Roborace (such as the localization module) and the performance requirements are moderate (limited maximum speed, no physical competition with the other vehicles). Gradually the teams participating in the racing competition should transfer to their own software and be able to handle more sophisticated tasks, such as obstacles avoidance, overtaking maneuvers, high speed racing, localization without GNSS.

The race takes place on a circuit. The cars start at a designated spot on a starting grid and have to accomplish a given number of laps faster than the competing teams. The competitors race independently and their racing time is compared after all teams have finished the race. No physical objects other than the competing race cars are present on a racetrack, but there can be virtual objects, of three kinds: static obstacles, loots and ghost cars. Cars get bonus time for collecting loots and penalty time for hitting obstacles or ghost cars. The total time is defined as the race time minus bonus time plus penalty time.

The race car is a battery electric vehicle with various sensors for perceiving the environment and localizing itself: 5 lidar, 8 ultrasonic, 6 cameras, 1 radar, 1 GPS/IMU, 1 slip-angle sensor. All the information from the sensors is united in the main ECU, the Nvidia Drive PX2. With this ECU, a platform for deep learning, sensor fusion and surround vision is available for the software developers. The software running on the Drive PX2 provides the information for the real time motion controller Speedgoat mobile target machine which offers real-time computation performance.

Software components are split into three groups: computational intense, time critical and offline. The first, running on the NVIDIA Drive PX2, consists of data and/or computational intense algorithms. This includes the complete perception and planning parts of the software. In contrast, highly time critical but less complex algorithms, such as Sensor fusion, odometry information collection and vehicle control, are running on the Speedgoat Real-Time ECU. Offline software is used to generate a map of a racetrack and create a time- and energy- optimal raceline based on the sensor data collected on a racetrack

before the race [16].

7.1.3 Roborace use cases

A coarse-grained overview of a race car functionality can be presented with a list of use cases:

- Race without obstacles.
- Avoid obstacles or stop.
- Update speed limit.
- Race with virtual obstacles.
- Race with virtual race cars.
- Move to pit.
- Perform an emergency stop.
- Perform a safe stop.

We analyze the first of these, *"Race without obstacles"*. The table 7.1 below provides a specification of the use case in the style introduced in [28].

Table 7.1: A detailed description of the *"Race without obstacles"* use case.

Name	Race_no_obstacles
Scope	System
Level	Business summary
Primary actor	Race car Operator
Secondary actor	Roborace Operator
Context of use	Race car has to obey an instruction
Preconditions	* Race car is on the racetrack grid. * Race car is not moving. * The global plan (trajectory and velocity profile) minimizing the race time is calculated. * The green flag is shown.
Trigger	The system receives a request from the race car operator to start the race
Main success scenario	* The system calculates the local plan (path and velocity profile) during the race trying to follow the global plan as close as possible. * The race car follows the local plan. * After finishing the required number of laps the race car performs a safe stop.

Success guarantee	The race car has completed the required number of laps and stopped.
Extensions	A. The red flag is received during the race * The race car recalculates a global plan to perform an emergency stop. * The race car performs an emergency stop. B. The yellow flag is received during the race. * The system sets the speed limit according to the received value. * The race car finishes the race following the global trajectory and not exceeding the new speed limit. C. The difference between the calculated (desired) location and real (according to the sensors) location is more than a given threshold. * The race car recalculates a global plan to perform an emergency stop. * The race car performs an emergency stop.
Stakeholders and interests	Race car Operator (requests the car to start the race). Roborace Manager (sets the race goals and policies). Roborace Operator (shows the green, yellow, red flags).

7.1.4 Modeling components of the system and its environment

In autonomous driving domain the system interacts with many environment components. A dedicated effort was made to make all assumptions and constraints explicit. The interviews were combined with domain exploration (by studying related papers and normative documents) to ensure that important environment properties are not overlooked.

The core components of the system could be identified at the very beginning of the project. It enabled assigning functionality to particular modules, rather than to a system as a whole (the case study focused on the planning module). The RACECAR class (listing 7.1) captures properties applicable to the system as a whole.

```
class
    RACECAR
feature
    control_module: CONTROL_MODULE
    perception_module: PERCEPTION_MODULE
    planning_module: PLANNING_MODULE
    localization_and_mapping_module:
        LOCALIZATION_AND_MAPPING_MODULE
end
```

Listing 7.1: Example of a system class.

Listing 7.2 provides an example of an environment class.

```
class
    RACETRACK
feature
    raceline: RACELINE
      -- Optimal raceline for the track
    map: MAP
      -- Coordinates of the bounding lines
end
```

Listing 7.2: Example of an environment class.

Due to the complexity of the domain, the components of the environment were grouped into three clusters:

- Environment - elements of the system's environment, with such classes as RACETRACK (listing 7.2), MAP and OBSTACLE.
- Interfaces - the race car's sensors and actuators, which are external to the developed software, but are part of the cyber-physical system and serve as interfaces between the system and its environment. They include such classes as SENSOR and LIDAR.
- Auxiliary classes - classes that define abstract concepts of the environment, that are not identified as the environment's elements. They include such classes as LOCATION and ORIENTATION.

Thanks to the inheritance mechanisms, we organized the description of these concepts into hierarchies; for example:

- Obstacles can be dynamic or static, so classes DYNAMIC_OBSTACLE and STATIC_OBSTACLE inherit from class OBSTACLE.
- Similarly, the classes describing sensors, such as LIDAR (listing 7.4), RADAR, and CAMERA, inherit from the class SENSOR (listing 7.3), which captures the characteristics applicable to all sensors. In addition, each class describing a particular sensor type captures specific features of that sensor.

```
deferred class
    SENSOR
feature
    position: LOCATION_3D
      --location in the world coordinates of the scene
    update_rate: REAL
      --sensor update rate
end
```

Listing 7.3: Code of the SENSOR class

The SENSOR class has such features as position, which captures the sensor location on a vehicle, and update_rate which reflects the frequency of sensor output. The LIDAR class inherits from SENSOR and adds such features as point_cloud and orientation.

```
deferred class
    LIDAR
inherit
    SENSOR
feature
    point_cloud: ARRAY2 [LOCATION]
    -- m by n matrix of detected points in lidar coordinate system

    object_points_distance: ARRAY2 [REAL]
    -- m by n matrix of distances to object points

    orientation: ORIENTATION
    -- Lidar orientation in the world coordinates of the scene
end
```

Listing 7.4: Code of the LIDAR class.

General environment constraints, such as the constraint *"If the yellow flag is, up race cars should limit their speed to a dedicated "safe speed""* are expressed with the class invariants, as in the following extract from the RACECAR class (listing 7.5).

```
class
    RACECAR
feature
    green_flag_is_up: BOOLEAN
    yellow_flag_is_up: BOOLEAN
    red_flag_is_up: BOOLEAN
    safe_stop_activated: BOOLEAN
    max_speed: REAL
    current_max_speed: REAL
        -- Current speed limit
    safe_speed: REAL
        -- Safe speed limit
invariant
    yellow_flag_is_up implies current_max_speed = safe_speed
    green_flag_is_up implies current_max_speed = max_speed
    red_flag_is_up implies safe_stop_activated
end
```

Listing 7.5: Implementation of the constraint *"If the yellow flag is up race cars should limit their speed to a dedicated "safe speed""*.

Environment assumptions are specified with pre- an postconditions, as the assumption related to the possible sequences of raising flags[1] by the Roborace (listing 7.6).

```
class
    ROBORACE
feature
    raise_yellow_flag
        require
          green_flag.is_up
        do
        ensure
            yellow_flag.is_up
            not green_flag.is_up
            not red_flag.is_up
        end
    raise_red_flag
        require
            green_flag.is_up or yellow_flag.is_up
        do
        ensure
            red_flag.is_up
            not green_flag.is_up
            not yellow_flag.is_up
        end
    end
```

Listing 7.6: Illustration of an environment assunption.

7.1.5 Producing OO functional specification

In the Roborace project each mission focuses on the accomplishment of some scenarios, such as "Race without obstacles". Thus the elements of the system's functionality were extracted from scenarios.

Initially, the system's functions appear simply as the features' names (listing 7.7):

```
class
    PLANNING_MODULE
feature
    calculate_race_trajectory (circuit_map: MAP; vehicle_param:
        VEHICLE_PARAMETERS; strategy: INTEGER)
      -- Calculate the optimal racing line for a circuit
        do
        end
```

[1] A flag is a signal sent to a race car to indicate a change in racing conditions.

```
end
```

Listing 7.7: Initial implementation of a requirement "Planning module shall calculate a raceline".

The features are further enriched with contracts, formulating the requirements. Below is an implementation of the requirement "At every position on a raceline the speed in the velocity profile shall not exceed the maximum race car's speed" (listing 7.8).

```
class
    PLANNING_MODULE

feature
    calculate_race_trajectory (circuit_map: MAP; vehicle_param:
        VEHICLE_PARAMETERS; strategy: INTEGER)
      -- Calculate the optimal racing line for a circuit
        do
        ensure
            velocity_limit_obeyed: across raceline.velocity_profile
                as rl all rl.item <car.max_speed end
        end
end
```

Listing 7.8: Implementation of a requirement *"At every position on a raceline the speed in the velocity profile shall not exceed the maximum race car's speed"*.

Contracts also capture abstract properties derived from time-ordering constraints, such as that the global plan must be calculated before calculating the local plan (listing 7.9):

```
class
  PLANNING_MODULE

feature
    calculate_local_plan: LOCAL_PLAN
      -- Calculate local path from current location to converge to
         global path
        require
            car.global_plan_is_calculated
        do
        ensure
            car.local_plan_is_calculated
        end
```

Listing 7.9: Capturing logical constraints.

7.1.6 Producing OO behavioral specification

The *"Race without obstacles"* use case, previously expressed in tabular format becomes simply a routine race_no_obstacles in the requirements class ROBORACE_USE_CASES sketched below (listing 7.10). It relies on conditional expressions to consider the use case alternative flows.

```
race_no_obstacles
    require
        not car.is_moving
        car.global_plan_is_calculated
        car.green_flag_is_shown
        car.is_on_starting_grid
    local local_plan: RACELINE
    do
      --Sequence of system actions in use case main flow
        from
        until
            car.race_is_finished or
            car.red_flag_is_shown or
            car.location_error_is_detected
        loop
            if car.yellow_flag_is_shown then update_speed end
            local_plan := car.planning_module.calculate_local_plan
            car.control_module.move (local_plan.speed, local_plan.
                orientation)
        end
        if
            car.red_flag_is_shown or car.location_error_is_detected
        then
            emergency_stop
        else
            safe_stop
        end
    ensure
        not car.is_moving
        car.is_in_normal_mode implies car.race_is_finished
    end
```

Listing 7.10: Implementation of the use case *"Race without obstacles"*.

The race_no_obstacles routine relies on implementing the routines update_speed, safe_stop, and emergency_stop: the respective features are called inside the use case. These features are implementation of the respective use cases, and such dependency corresponds to «include» and «extend» relationships between use cases.

The ROBORACE_USE_CASES class is thus a collection of routines corresponding to the system's use cases (listing 7.11).

```
class
    ROBORACE_USE_CASES

feature
    car: RACECAR

    safe_stop
        require
            car.is_in_normal_mode
        do
            car.control_module.safe_stop
        ensure
            not car_is_moving
        end

    emergency_stop
        require
            car.red_flag_is_shown or car.location_error_is_detected
        do
            car.control_module.emergency_stop
        ensure
            not car.is_in_normal_mode
            not car.is_moving
        end

    update_speed
        require
            car.yellow_flag_is_shown
        do
            car.update_max_speed (car.yellow_flag_speed)
        ensure
            car.max_speed = car.yellow_flag_speed
        end

    race_no_obstacles
        do
            --implementation is listed above
        end

    avoid_obstacle_or_stop
        do
        end

    race_with_virtual_obstacles
        do
        end
```

```
    race_with_virtual_race_cars
        do
        end

    move_to_pit
        do
        end
end
```

Listing 7.11: Roborace use-case class.

Relation between use cases and test cases

Use case stories define test cases for use cases [57]. The class ROBORACE_USE_CASE_STORIES inherits from ROBORACE_USE_CASES class. It includes a collection of routines corresponding to use case stories.

When a use case takes the form of a routine with contracts, extracting use case stories from such a routine becomes a semi-automated task. For example, the emergency_stop use case accepts two options in its precondition — (1) when the red flag is shown or (2) when a location error is detected. These options map to the following use case stories written according to the UOOR approach (listing 7.12):

```
class
    ROBORACE_USE_CASE_STORIES

inherit
    ROBORACE_USE_CASES

feature
    emergency_stop_red_flag_story
        require
            car.red_flag_is_shown
        do
            emergency_stop
        end

    emergency_stop_location_error_story
        require
            car.location_error_is_detected
        do
            emergency_stop
        end
end
```

Listing 7.12: Use case stories extracted from the *"emergency stop"* use case.

These routines represent the two different paths through the `emergency_stop` use case, characterized by their preconditions. The connection with the parent use case is visible because the stories call the routine encoding the use case. The two routines must be exercised at least once with test input that meets their preconditions.

A similar analysis makes it possible to extract 5 use case stories from the *"Race without obstacles"* use case:

- 3 for each possible loop exit condition.
- 1 corresponding to the true antecedent of the implication in the second postcondition assertion.
- 1 corresponding to the true consequent and false antecedent of the said implication.

The full collection of the extracted use case stories may be found in a publicly available repository [91].

7.1.7 Lessons learnt from the case study

With respect to **environment properties specification**, it turned out that project stakeholders tend to take for granted environment-related information. The UOOR methodology can address the issue by serving as a structural basis for elicitation activities: The approach requires to explicitly specify components of the environment and environmental properties such as assumptions, constraints and invariants.

The *"Race without obstacles"* use case provides a good illustration of the **difference between contract-based and scenario-based specification**. As a specification, this scenario expresses, among other properties, that the system calculates a local plan and then follows it. It states this property in the form of a strict sequence of operations which, however, only covers some of the many possible scenarios.

It does list extensions, but only three of them, and does not reflect the many ways in which they can overlap. For example:

- It can happen that the green flag is shown some time after the yellow flag; but the extensions do not even list the green flag.
- In the same way, the red flag can be shown after a yellow flag.

An attempt to add extensions to cover all possibilities would have no end, as so many events may occur as to create a combinatorial explosion of possible sequencings.

One way out of this dead end would be to use temporal logic [106], which provides a finite way to describe a possibly infinite but constrained set of sequences of events or operations. UOOR relies on a different idea: use

logical rather than sequential constraints. Sequential constraints become just a special case: we can express that `A` must come before `B` simply by defining a condition `C` as part of both the postcondition of `A` and the precondition of `B`. But the logic-based specification scheme covers many more possibilities than just this special case. The specification the example just mentioned is presented in listing 7.6.

The UOOR approach helped to **structure the entire process of requirements elicitation and analysis**:

- It urged to identify and specify the elements of the system's environment.
- This process revealed the assumptions and constraints that could be overlooked otherwise.
- Analysis of use cases revealed more abstract properties, than time-ordering constraints.

7.1.8 Limitations

This case study serves as a proof of concept on a significant ongoing project, yet it can not be considered to be a systematic empirical validation. Consequently, it is not possible to state any guarantees that the UOOR approach will increase productivity or decrease defects. The case study rather serves as an observation that object-oriented technology with logical constraints is more general than scenario-based techniques and encompasses them as a special case.

7.1.9 Conclusion

With respect to the research questions, we can make the following conclusions.

The case study demonstrated that the UOOR methodology facilitates requirements specification by structuring the process of requirements elicitation and analysis. It helps to find the questions to be asked during requirements elicitation and guides how to analyze scenarios to produce logical specification.

On the Roborace case study we have demonstrated that with the UOOR approach, one can:

1. Express the fundamental abstractions in the form of requirements classes.
2. Express the fundamental constraints in the form of invariants for these classes.
3. Express typical usage scenarios with specification drivers. (Unlike the previous two, this task does not make any attempt at exhaustiveness, since examples can only cover a fragment of all possibilities; instead, it concentrates on the scenarios of most interest to stakeholders, and those most likely to cause potential issues or bugs.)

4. As a consistency check, ascertain that the scenarios (item 3) preserve the invariants (item 2).

More generally, the combination of an object-oriented approach to structure the requirements (1), equipped with invariants (2) as well as other forms of contracts (preconditions, postconditions), with use cases to illustrate the requirements through examples of direct interest to stakeholders (3) and shown to preserve the invariants (4) provides a promising method for obtaining correct and practically useful requirements.

7.2 UOOR user study

The study conducted at the University of Toulouse [92] evaluates the perception of the UOOR approach and its potential to be adopted in industry. Since software engineers are already equipped with a set of well-known requirements techniques, their willingness to study and adopt a new approach is based on the following factors:

- Is this approach easy to learn? Will it require much time and efforts to study it?
- Does this approach provide an added value? Will it help me to improve requirements specifications?

To address these concerns, the study formulates the following research questions: (i) is limited training sufficient for learning OO contract-based requirements? (ii) does learning contract-based OO requirements techniques help to produce better UML specifications?

7.2.1 Study design

The study was conducted as a part of the "OO Analysis and Design" course at the University of Toulouse. This course, like its many counterparts in other universities [37], introduces UML as requirements modeling language.

The study had two parts:

- A controlled experiment [129] based on the "Instantrame[2]" case study, where students produced requirements applying two different approaches.
- A questionnaire, where students reflected on their experience.

In total, 31 students participated in the study: bachelor's students in their third year and master's students in their first year. Course instructors provided a textual description of a case study to students, which they further used to elicit requirements and produce various requirements artifacts.

The experiment was split into two parts. In the first part (1.5 hours) theory on UOOR requirements was presented to students. They were already familiar with UML and scenario modeling. Further, they had a task to describe two scenarios for each of the two given use cases according to a provided template. In the second part (4.5 hours), students were randomly split into two groups and had to complete two tasks:

1. (2-3 hours) Students of Group 1 specified UOOR requirements for the first use case. Students of Group 2 produced a sequence diagram for the first use case.

[2]Instantrame is a social network on smartphones allowing anyone with an account to share photos

2. (2-3 hours) Students of Group 1 worked on a sequence diagram for the second use case. Students of Group 2 specified UOOR requirements for the second use case.

After submitting the results of their work, students filled in an online questionnaire. The questionnaire included two types of questions. Single-choice questions were formulated as statements that participants had to evaluate based on a Likert scale (‘Strongly disagree’, ‘Disagree’, ‘Agree’, ‘Strongly agree’, ‘Neutral’ choices). We used those questions to collect quantitative feedback on the use of the UOOR approach. The questionnaire also included open questions to collect additional qualitative feedback, such as the perceived advantages of the approach, the difficulties that participants faced using the approach, what are the potential improvements of the UOOR, and how applying UOOR helps to improve UML-based specifications.

7.2.2 Study results

All of the 31 study participants have participated in a survey, yet one of them answered only part of the questions. Although the population size is not large enough to draw definite conclusions, the study provides a preliminary outlook on the usability of the UOOR approach.

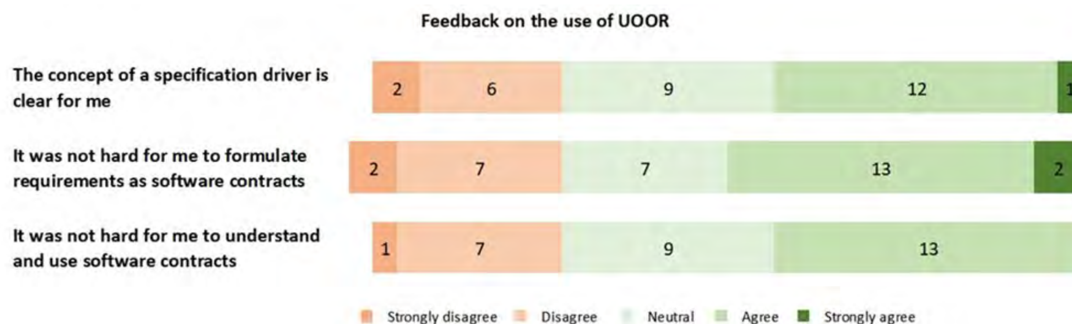

Figure 7.1: Feedback on the use of UOOR.

26% of respondents declared that it was hard to understand and use software contracts for requirements specification, whereas 43% had a positive experience and 30% were neutral (fig. 7.1).

All participants were able to list the advantages of formulating requirements in UOOR. The participants highlighted that the UOOR methodology is easy to understand and follow and that it facilitates producing detailed, specific, readable requirements that are easy to validate (table 7.2).

What are the reasons to use UOOR?
Readability and clarity
Brings more details at the specification level
Modularity
The methodology is easy to understand and follow
Facilitates identification and analysis of use case scenarios
Obliges to write preconditions and postconditions
Well-detailed, very specific
Easy reuse, which facilitates maintenance
Better possibility of requirements validation

Table 7.2: Summary of responses to the question What are the reasons to use UOOR?

The particular difficulties, stated by the experiment participants, were: not enough familiarity with contracts; not enough examples provided; not enough practice (table 7.3).

What difficulties have you faced when applying UOOR?
Not enough familiarity with Eiffel language and contracts
It is difficult to formulate pre- and post-conditions for some scenarios.
Need more examples in order to adapt
Need more practice

Table 7.3: Summary of responses to the question What difficulties have you faced when applying UOOR?

The questionnaire responses indicate that UOOR helped students to improve their UML specifications in the following ways: to think of elements we hadn't thought of, for example additional preconditions, to discover details that need to be added to features, to better define and implement use cases, to identify alternative scenarios for the systems use case, to better analyze the requirements in a global way for a specification (table 7.4). This is a significant result, since people are more likely to adopt an approach that provides immediate benefits, such as improving requirements specifications produce with currently used approach.

How applying UOOR helped to improve UML specification?
Facilitates discovering details that need to be added to features
Facilitates better analysis of the requirements in a global way
Understanding of the purpose of the steps to be carried out when writing specifications in UML
Facilitates identifying alternative scenarios for the systems use cases
Facilitates better definition and implementation of use cases

Table 7.4: Summary of responses to the question How applying UOOR helped to improve UML specification?

The questionnaire also collected the feedback of the participants on the usability of the UOOR approach. The responses indicate that more detailed description of the approach with more examples would improve its usability, as well as diagram and tool support 7.5.

What could make the UOOR approach more usable?
A tool allowing the construction of an architecture according to requirements
Diagrams to make it more concrete and understandable
A more detailed description of the approach
Having more examples

Table 7.5: Summary of responses to the question “What could make the UOOR approach more usable?”

7.2.3 Limitations and threats to validity

The experiment is limited to a single course at the University of Toulouse, so the results might not be applicable to other educational settings or populations. Another threat is that the subjects of the experiment were students, rather than software engineering professionals. Nevertheless, it has been demonstrated that software engineering students and professionals demonstrate similar results in accomplishing relatively small tasks [44].

7.2.4 Discussion

The difficulties reported by the participants, such as not enough familiarity with the Eiffel language and contracts, indicate that the course prerequisites might not have been explicit enough. Since Eiffel language and Design by Contracts are part of the standard curriculum of the bachelor’s software engineering program, we assumed that all study participants are familiar with these topics.

However, some master students come from foreign programs which results in deviations in students' background.

Several participants claimed that not enough explanations and examples were provided. To address the issue, the thesis provides a detailed methodology with two illustrative examples. The approach will also be published in a textbook accompanied with a website and a github repository, which will serve as a forum for discussing the approach.

7.2.5 Conclusion

To be embraced by the industry practitioners, the requirements approach should satisfy two criteria: (i) it should not require substantial training, (ii) it should be perceived as useful for current practices. In the experiment, presented in this section, we evaluated the UOOR approach against these two criteria. With respect to the study participants, both of the criteria were satisfied: (i) most of the participants were able to learn the approach in a limited time, (ii) all participants were able to list the advantages of using the approach; applying the approach helped to improve UML-based specifications. The results provide preliminary evidence that the UOOR method has a potential to be adopted by requirements engineering practitioners.

7.3 Conclusion

This chapter provided exploratory empirical evaluation of the UOOR approach and assessed its expressiveness and usability. It demonstrated that the approach supports the expression of the fundamental abstractions in the form of requirements classes equipped with contracts, to express usage scenarios with specification drivers and to ascertain that the scenarios satisfy the contracts. It established that the UOOR approach does not require substantial training; applying the UOOR method helps to improve requirements specifications.

The results provide preliminary evidence that the UOOR method has a potential to be adopted by requirements engineering practitioners.

To draw definitive conclusions, the method should be evaluated in setting of larger scale, such as industrial projects and related user studies, which remains a prominent perspective for future work.

Part III

Conclusion

Chapter 8

Discussion and conclusion

8.1 Summary of contributions

This thesis explored the practical application of seamless requirements by addressing the following research questions:

- What should be the process of seamless software development from requirements to code?
- What tool support is required to ensure traceability between requirements and other project artifacts?

In answer to these questions, the thesis has devised the Unified Object-Oriented approach to Requirements (UOOR) and the supporting Traceability tool. It demonstrated that the approach is affordable and practically usable and that it facilitates producing unambiguous, verifiable, reusable and traceable requirements. The specific advances are the following:

1. A demonstration that the concept of class is general enough to describe not only "objects" in a narrow sense but also scenarios such as use cases and user stories and other important artifacts such as test cases and oracles.
2. A methodology that a requirements engineer may use as a guide for requirements specification process.
3. A partial formal model of requirements engineering through the definition of relations and their formal mathematical properties.
4. The notion of seamless requirements traceability, which relies on propagation of traceability links, themselves based on formal properties of relations between project artifacts.
5. The Traceability tool, which supports requirements traceability by creating typed links between project elements, and tracking changes from requirements to the affected code elements.
6. An evaluation of the approach by applying it to a significant real-world case study and conducting a controlled experiment.

8.2 Evaluation of the UOOR approach

Chapter 7 provided the initial evaluation of the approach; chapters 5 and 6 extended the approach by providing UOOR traceability information model supported with the Traceability tool. Summarising the advancements, we evaluate the UOOR approach and the characteristics of requirements formulated with the approach.

The UOOR approach is:

- **Explicit:** The approach provides a methodology and illustrative examples. The thesis serves as a guide for the process of producing requirements. The ongoing work on the book illustrating the approach [20] and online portal with the book's supporting materials may serve as a discussion forum for the adopters of the approach.

- **Not demanding:** The approach does not require the knowledge of formal models or mathematical notations. It does require, however, familiarity with a programming language and design by contract. The clarity of the Eiffel language, chosen as the requirements notation, significantly lowers this barrier: in order to be able to formulate contracts in Eiffel, a requirements engineer must learn only a few language constructs. The ability to rely on the mature IDE, EiffelStudio, makes the task even easier: since OO requirements are compilable code elements, the IDE provides error codes in case if OO requirements are formulated with errors.

- **Tool-supported:** The approach relies on the general-purpose IDE (EiffelStudio for the Eiffel language) and integrated Traceability tool. EiffelStudio provides facilities for static and dynamic verification of the developed system against the requirements; enables traceability links creation and management; provides syntactic check for OO requirements. The Traceability tool provides more advanced functionality for traceability links management: it allows assigning types to project elements, creating typed traceability links and tracking changes from requirements to the related project elements.

- **Seamless:** The UOOR approach provides a methodology of producing UOOR requirements, rather than translating requirements documents to a programming language. Thus, UOOR relies on a uniform process (based on refinement) and a uniform notation (Eiffel language).

Requirements, formulated with the UOOR approach, are:

- **Reusable.** Based on the capabilities, provided by the object-oriented technology, requirements in UOOR can be reused not but copy-pasting natural language texts, but as libraries of domain-specific components specifications in the form of contracted deferred software classes. Being implementation-independent, such specifications allow for different implementation.

- **Verifiable.** Since OO requirements are code elements, an IDE provides basic consistency checks at compile time. When contract checking is enabled at runtime, the IDE monitors contracts violations. Since every contract can have a unique tag, a developer can trace an exception to the violated contract. Contract specification serves as oracles for dynamic testing. Moreover, scenarios, implemented as specification drivers, serve as tests when passed actual arguments. A static verifier, such as Autoproof [122], can be used to ensure static verification of system's functional correctness.
- **Unambiguous.** Ambiguity, innate to natural language texts, can be eliminated by introducing formal notation. In UOOR contracts serve as the notation for requirements: being understandable due to readability and clarity of the Eiffel language, at the same time they remove ambiguity inherent to natural language texts.
- **Traceable.** The approach introduces the notion of seamless requirements traceability, which enables propagating traceability links based on formal properties of relations between project elements. A dedicated Traceability tool provides facilities for creating and managing traceability links and tracking changes from requirements to related code elements.

8.3 Limitations

The thesis has explored the practical applicability of the UOOR approach in Eiffel language. Although the approach is scalable to other declarative object-oriented languages, it should be noted that it relies on the Eiffel's readability and native support of contracts and on the rich functionality of EiffelStudio and related tools (such as Autoproof and AutoTest). Application of the approach for projects that use programming language not possessing such qualities, may substantially reduce its usability. Additional methodological and tool support may thus be required.

The thesis explored the application of the approach to the greenfield (i.e. developed from scratch) projects. Its application to the brownfield projects may be tedious in case of poor documentation practices.

The UOOR approach covers only functional requirements and constraints and thus does not cover non-functional requirements. We identify it as one of the future work directions.

8.4 Perspectives

The thesis devised a methodology of the UOOR approach and its initial evaluation. Nevertheless, many perspectives for future work can be observed.

First, the approach shall be evaluated in settings of larger scale, such as industrial projects and related user studies. The ongoing work on the book illustrating the approach [20] and a supporting web-site shall contribute to

creating a community of the users of the approach and recruiting projects for the approach's evaluation.

The second direction of work is the coverage of non-functional requirements. How non-functional requirements can be treated in the UOOR approach? Recent work on use-case-modelling of crosscutting concerns ([135]) can be a starting point in such a research.

The work on seamless requirements traceability can be continued in the following directions:

- Further exploration of relation propagation rules.
- Practical evaluation of seamless traceability on a significant project.
- Tool support.

The thesis presented the Traceability tool, which supports creating typed links between project elements and track changes from requirements to the related project artifacts. Further directions of the tool improvement are the following:

- Implementing link propagation, which will enable seamless traceability between software project artifacts.
- Ensuring the tool's usability in industry by enhancing its interface and fully integrating it into EiffelStudio.

These perspectives highlight the difficulty of the task of introducing the requirements engineering approach that would be embraced by the software engineering practitioners. Rather than a single effort, it is a continuous journey. This thesis provided a basis for such an endeavor and demonstrated the potential of its successful completion.

Bibliography

[1] IEEE 1220-2005. *IEEE Standard for Application and Management of the Systems Engineering Process*. 2005. DOI: 10.1109/IEEESTD.2005.96469.

[2] ISO/IEC/IEEE 24765: 2010. *Systems and software engineering – Vocabulary*. 2010.

[3] ISO/IEC/IEEE 29148: 2011. *Systems and software engineering – Requirements engineering*. 2011.

[4] ISO/IEC/IEEE 24765: 2017. *Systems and software engineering – Vocabulary*. 2010.

[5] ISO/IEC/IEEE 29148:2018. *Systems and Software Engineering – Life Cycle Processes – Requirements Engineering*. 2018.

[6] IEEE 830-1998. *IEEE Recommended Practice for Software Requirements Specifications*. 1998. DOI: 10.1109/IEEESTD.1998.88286.

[7] ISO 8807:1989. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour*. 1988.

[8] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[9] Daniel Amyot, Jean-François Roy, and Michael Weiss. "UCM-driven testing of web applications". In: Springer, 2005, pp. 247–264. DOI: 10.1007/11506843_18.

[10] Ludovic Apvrille, Pierre de Saqui-Sannes, and Rob Vingerhoeds. "An Educational Case Study of Using SysML and TTool for Unmanned Aerial Vehicles Design". In: *IEEE Journal on Miniaturization for Air and Space Systems* 1.2 (2020), pp. 117–129. DOI: 10.1109/JMASS.2020.3013325.

[11] Guillermo Francisco Arango. *Domain engineering for software reuse*. University of California, Irvine, 1988.

[12] Dave Arnold, Jean-Pierre Corriveau, and Wei Shi. "Modeling and Validating Requirements Using Executable Cotnracts and Scenarios". In: May 2010, pp. 311–320. DOI: 10.1109/SERA.2010.46.

[13] David Arnold. “An open framework for the specification and execution of a testable requirements model”. PhD thesis. Ottawa, Ontario, 2009. DOI: 10.22215/etd/2009-10147. URL: https://curve.carleton.ca/81f78a01-0190-4d0f-83c7-34be63120a61.

[14] S Baase. *Ethical Issues for Computing and the Internet*. 2008.

[15] Kent Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003. ISBN: 978-0-321-14653-3.

[16] Johannes Betz et al. “A software architecture for an autonomous racecar”. In: *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*. IEEE. 2019, pp. 1–6.

[17] Dines Bjørner. “Domain engineering”. In: *Formal Methods: State of the Art and New Directions* (2010), pp. 1–41. DOI: 10.1007/978-1-84882-736-3_1.

[18] Barry W. Boehm. “Software Engineering Economics”. In: *Software Pioneers: Contributions to Software Engineering*. Ed. by Manfred Broy and Ernst Denert. Berlin, Heidelberg: Springer, 2002, pp. 641–686. ISBN: 978-3-642-59412-0. DOI: 10.1007/978-3-642-59412-0_38.

[19] Grady Booch and et al et. “Object-oriented analysis and design with applications”. In: *ACM SIGSOFT software engineering notes* 33.5 (2008). publisher: ACM New York, NY, USA, pp. 29–29.

[20] Jean-Michel Bruel, Sophie Ebersold, and Mariya Naumcheva. *Applying Requirements and Business Analysis*. Springer, to appear.

[21] Ray JA Buhr and Ron S Casselman. *Use case maps for object-oriented systems*. Prentice-Hall, Inc., 1995.

[22] Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. “A model-driven approach for evaluating traceability information”. In: *Model-Driven Software Development* (2017), p. 436.

[23] International Institute of Business Analysis (IIBA). *A guide to the business analysis body of knowledge(BABOK Guide), v3*. 2015.

[24] Juan M. Carrillo de Gea et al. “Requirements Engineering Tools: An Evaluation”. In: *IEEE Software* 38.3 (2021), pp. 17–24. DOI: 10.1109/MS.2021.3058394.

[25] Jane Cleland-Huang. “Just Enough Requirements Traceability”. In: *30th Annual International Computer Software and Applications Conference (COMPSAC’06)*. Vol. 1. 2006, pp. 41–42. DOI: 10.1109/COMPSAC.2006.57.

[26] Jane Cleland-Huang. “Traceability in agile projects”. In: *Software and Systems Traceability*. Springer, 2011, pp. 265–275. DOI: 10.1007/978-1-4471-2239-5_12.

[27] Jane Cleland-Huang et al. “Software traceability: trends and future directions”. In: *FOSE 2014: Future of software engineering proceedings*. 2014, pp. 55–69. DOI: 10.1145/2593882.2593891.

[28] Alistair Cockburn. *Writing effective use cases*. Pearson Education India, 2001.

[29] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[30] Steve Cook et al. *Unified Modeling Language (UML) Version 2.5.1*. Tech. rep. Dec. 2017. URL: https://www.omg.org/spec/UML/2.5.1.

[31] Cucumber. [Online; accessed 2024-05-16]. URL: https://cucumber.io/tools/.

[32] Jennifer A Davis et al. "Study on the barriers to the industrial adoption of formal methods". In: *Formal Methods for Industrial Critical Systems: 18th International Workshop, FMICS 2013, Madrid, Spain, September 23-24, 2013. Proceedings 18*. Springer. 2013, pp. 63–77. DOI: 10.1007/978-3-642-41010-9_5.

[33] Jeremy Dick, Elizabeth Hull, and Ken Jackson. *Requirements engineering*. Springer, 2017.

[34] Xavier Franch et al. "The state-of-practice in requirements specification: an extended interview study at 12 companies". In: *Requirements Engineering* 28.3 (2023), pp. 377–409. DOI: 10.1007/s00766-023-00399-7.

[35] Luiz Paulo Franz et al. "Um Editor para a Linguagem de Especificação de Requisitos RELAX". In: *Anais da I Escola Regional de Engenharia de Software*. SBC. 2017, pp. 33–40.

[36] Samuel A Fricker, Rainer Grau, and Adrian Zwingli. "Requirements engineering: best practice". In: *Requirements Engineering for Digital Health*. Springer, 2015, pp. 25–46. DOI: 10.1007/978-3-319-09798-5_2.

[37] Dirk Frosch-Wilke. *Using UML in Software Requirements Analysis - Experiences from Practical Student Project Work*. Jan. 1, 2003. DOI: 10.28945/2610.

[38] Florian Galinier. "Seamless development of complex systems: a multi-requirements approach". PhD thesis. Université Paul Sabatier-Toulouse III, 2021.

[39] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Boston MA: Addison-Wesley Professional, 1994. ISBN: 0-201-63361-2.

[40] Sarah Gatti et al. *COOL Relax Editor*. Tech. rep. Internal Report M2ICE, U. Toulouse II Le Mirail, 2011.

[41] M Glinz. *A glossary of requirements engineering terminology. Version 2.0.1*. Tech. rep. International Requirements Engineering Board (IREB), 2022.

[42] Orlena CZ Gotel and CW Finkelstein. "An analysis of the requirements traceability problem". In: *Proceedings of IEEE international conference on requirements engineering*. IEEE. 1994, pp. 94–101. DOI: 10.1109/ICRE.1994.292398.

[43] Zakaria Hachm. "Seamless Requirements Traceability Tool". Graduation Project Report. Mohammed V University in Rabat, 2024.

[44] Martin Höst, Björn Regnell, and Claes Wohlin. "Using students as subjectsa comparative study of students and professionals in lead-time impact assessment". In: *Empirical Software Engineering* 5 (2000), pp. 201–214. DOI: 10.1023/A:1026586415054.

[45] Li Huang and Bertrand Meyer. "A failed proof can yield a useful test". In: *Software Testing, Verification and Reliability* 33.7 (2023). DOI: 10.1002/stvr.1859.

[46] Li Huang et al. "Lessons from Formally Verified Deployed Software Systems (Extended version)". In: *arXiv preprint arXiv:2301.02206* (2023).

[47] Azham Hussain, Emmanuel OC Mkpojiogu, and Fazillah Mohmad Kamal. "The role of requirements in the success or failure of software projects". In: *International Review of Management and Marketing* 6.7 (2016), pp. 306–311.

[48] IBM. *DOORS Next*. [Online; accessed 2024-05-12]. URL: https://www.ibm.com/products/requirements-management-doors-next.

[49] IBM. *What is requirements management?* [Online; accessed 2024-08-30]. URL: https://www.ibm.com/topics/what-is-requirements-management/.

[50] Integranova. *Integranova M.E.S.* [Online; accessed 2023-05-11]. URL: https://www.integranova.com/.

[51] Mohsin Irshad, Kai Petersen, and Simon Poulding. "A systematic literature review of software requirements reuse approaches". In: *Information and Software Technology* 93 (2018), pp. 223–245. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2017.09.009.

[52] Daniel Jackson. "Alloy: A Language and Tool for Exploring Software Designs". In: *Commun. ACM* 62.9 (Aug. 2019). publisher-place: New York, NY, USA publisher: Association for Computing Machinery, pp. 66–76. ISSN: 0001-0782. DOI: 10.1145/3338843.

[53] Michael Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., 1995.

[54] Michael Jackson. "The meaning of requirements". In: *Annals of Software Engineering* 3.1 (1997), pp. 5–21.

[55] Michael Jackson and Pamela Zave. "Deriving Specifications from Requirements: an Example". In: *1995 17th International Conference on Software Engineering*. 1995, pp. 15–24. DOI: 10.1145/225014.225016.

[56] Ivar Jacobson, Ian Spence, and Kurt Bittner. *Use-case 2.0 The Guide to Succeeding with Use Cases*. Alexandria, Virginia: Ivar Jacobson International SA., Dec. 2011. URL: https://www.ivarjacobson.com/sites/default/files/field_iji_file/article/use-case_2_0_jan11.pdf.

[57] Ivar Jacobson, Ian Spence, and Brian Kerr. "Use-Case 2.0: The Hub of Software Development". In: *Queue* 14.1 (Jan. 2016). publisher-place: New York, NY, USA publisher: Association for Computing Machinery, pp. 94–123. ISSN: 1542-7730. DOI: 10.1145/2898442.2912151.

[58] Ivar Jacobson et al. *Object Oriented Software Engineering: A Use Case Driven Approach*. Boston MA: Addison-Wesley, 1992. ISBN: 978-0-201-54435-0.

[59] jUCMNav. *jUCMNav tool*. [Online; accessed 2023-05-12]. URL: https://github.com/JUCMNAV.

[60] JUnit. *JUnit 5*. [Online; accessed 2024-06-11]. URL: https://junit.org/junit5/.

[61] Mohamad Kassab. "The changing landscape of requirements engineering practices over the past decade". In: *2015 IEEE Fifth International Workshop on Empirical Requirements Engineering (EmpiRE)*. 2015, pp. 1–8. DOI: 10.1109/EmpiRE.2015.7431299.

[62] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques*. Wiley, 1998, p. 294.

[63] A. van Lamsweerde and L. Willemet. "Inferring declarative requirements specifications from operational scenarios". In: *IEEE Transactions on Software Engineering* 24.12 (Dec. 1998), pp. 1089–1114. ISSN: 00985589. DOI: 10.1109/32.738341.

[64] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. English. 1st. Wiley, 2009. ISBN: 978-81-265-4589-6.

[65] Phillip A. Laplante. *Requirements engineering for software and systems*. CRC Press, 2017. ISBN: 9781138196117. DOI: 10.1201/9781315303710.

[66] Craig Larman. *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education India, 2012.

[67] Gary T Leavens, Albert L Baker, and Clyde Ruby. "JML: a notation for detailed design". In: *Behavioral Specifications of Businesses and Systems*. Springer, 1999, pp. 175–188. DOI: 10.1007/978-1-4615-5229-1_12.

[68] Dean Leffingwell. *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison-Wesley, 2010.

[69] Yang Li and Walid Maalej. "Which traceability visualization is suitable in this context? a comparative study". In: *Requirements Engineering: Foundation for Software Quality: 18th International Working Conference, REFSQ 2012, Essen, Germany, March 19-22, 2012. Proceedings 18*. Springer. 2012, pp. 194–210. DOI: 10.1007/978-3-642-28714-5_17.

[70] Fangfei Lin and Hao Chen. "Comparative Study of Requirements Traceability in Facing Requirements Change: Systematic Literature Study and Survey". MS thesis. Karlskrona, Sweden: Blekinge Institute of Technology, 2019.

[71] Lin Liu, Tong Li, and Fei Peng. "Why Requirements Engineering Fails: A Survey Report from China". In: *2010 18th IEEE International Requirements Engineering Conference*. 2010, pp. 317–322. DOI: 10.1109/RE.2010.45.

[72] Oscar Pastor López, Isidro Ramos Salavert, and José H Canós Cerdá. "Oasis v2: A class definition language". In: *Database and Expert Systems Applications: 6th International Conference, DEXA'95 London, United Kingdom, September 4–8, 1995 Proceedings 6*. Springer. 1995, pp. 79–90. DOI: 10.1007/BFb0049107.

[73] Pericles Loucopoulos and Vassilios Karakostas. *System requirements engineering*. McGraw-Hill, Inc., 1995.

[74] Alistair Mavin et al. "Easy approach to requirements syntax (EARS)". In: IEEE, 2009, pp. 317–322. DOI: 10.1109/RE.2009.9.

[75] Kent J McDonald. *Beyond requirements: analysis with an agile mindset*. Addison-Wesley, 2015.

[76] Bertrand Meyer. "On Formalism in Specifications". In: *IEEE Software* 2.1 (1985), pp. 6–26. DOI: 10.1109/MS.1985.229776.

[77] Bertrand Meyer. *Eiffel: the language*. USA: Prentice-Hall, Inc., 1992. ISBN: 0132479257.

[78] Bertrand Meyer. *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs, 1997.

[79] Bertrand Meyer. "Multirequirements". In: [Online; accessed 2023-03-13]. MV Wissenschaft, 2013. URL: https://se.inf.ethz.ch/~meyer/publications/methodology/multirequirements.pdf.

[80] Bertrand Meyer. *The ABC of software engineering*. [Online; accessed 2023-03-02]. 2013. URL: https://bertrandmeyer.com/2013/03/25/the-abc-of-software-engineering/.

[81] Bertrand Meyer. *Handbook of Requirements and Business Analysis*. Springer, 2022.

[82] Bertrand Meyer et al. "Programs That Test Themselves". In: *Computer* 42.9 (2009), pp. 46–55. DOI: 10.1109/MC.2009.296.

[83] Bertrand Meyer et al. "Towards an anatomy of software requirements". In: *Software Technology: Methods and Tools: 51st International Conference, TOOLS 2019, Innopolis, Russia, October 15–17, 2019, Proceedings 51*. Springer. 2019, pp. 10–40. DOI: 10.1007/978-3-030-29852-4_2.

[84] Modeliosoft. *Modelio - UML/BPMN modeling tool.* [Online; accessed 2023-05-15]. URL: https://www.modeliosoft.com/en/.

[85] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[86] Alexandr Naumchev. *A library of reusable templates for constructing verifiable and expressive requirements.* [Online; accessed 2024-05-16]. Mar. 2018. URL: https://github.com/anaumchev/requirements_templates.

[87] Alexandr Naumchev. "Seamless Object-Oriented Requirements". In: IEEE, 2019, pp. 0743–0748. DOI: 10.1109/SIBIRCON48586.2019.8958211.

[88] Alexandr Naumchev and Bertrand Meyer. "Complete Contracts through Specification Drivers". In: July 2016, pp. 160–167. DOI: 10.1109/TASE.2016.13.

[89] Alexandr Naumchev et al. "AutoReq: Expressing and verifying requirements for control systems". In: *Journal of Computer Languages* 51 (Apr. 1, 2019), pp. 131–142. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.02.004.

[90] Maria Naumcheva. "Object-Oriented Approach for Requirements Specification". In: *Workshops, Doctoral Symposium, and Posters & Tools Track-REFSQ Co-Located Events 2022.* Vol. 3122. Doctoral Symposium. CEUR-WS.org. 2022, pp. 1–7. URL: https://ceur-ws.org/Vol-3122/DS-paper-3.pdf.

[91] Maria Naumcheva. *Roborace requirements repository.* [Online; accessed 2024-05-17]. Mar. 22, 2022. URL: https://github.com/mnaumcheva/Roborace_requirements_code.

[92] Maria Naumcheva. "Teaching Object-Oriented Requirements Techniques: An Experiment". In: *Agents and Multi-agent Systems: Technologies and Applications 2023.* Ed. by Gordan Jezic et al. Singapore: Springer Nature Singapore, 2023, pp. 347–353. ISBN: 978-981-99-3068-5. DOI: 10.1007/978-981-99-3068-5_32.

[93] Maria Naumcheva. *Library requirements repository.* [Online; accessed 2024-12-13]. 2024. URL: https://github.com/mnaumcheva/Library_requirements_code.

[94] Maria Naumcheva et al. "Object-Oriented Requirements: a Unified Framework for Specifications, Scenarios and Tests". In: *The Journal of Object Technology* 22.1 (2023), pp. 1–19. DOI: 10.5381/jot.2023.22.1.a3.

[95] Ildar Nigmatullin et al. “RQCODE–Towards Object-Oriented Requirements in the Software Security Domain”. In: *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2022, pp. 2–6. DOI: 10.1109/ICSTW55395.2022.00015.

[96] Ildar Nigmatullin et al. “RQCODE: Security Requirements Formalization with Testing”. In: *IFIP International Conference on Testing Software and Systems*. Springer. 2023, pp. 126–142. DOI: 10.1007/978-3-031-43240-8_9.

[97] *OMG Systems Modeling Language (OMG SysML), Version 2.0 beta*. Tech. rep. 2024. URL: https://www.omg.org/spec/SysML/2.0/Beta2/Language/PDF.

[98] Juan Carlos Molina Oscar Pastor. *Model-Driven Architecture in Practice*. Berlin, Heidelberg: Springer, 2007. ISBN: 978-3-540-71867-3. DOI: 10.1007/978-3-540-71868-0.

[99] Gunnar Overgaard and Karin Palmkvist. *Use Cases: Patterns and Blueprints*. Addison-Wesley Professional, 2004.

[100] Cristina Palomares, Carme Quer, and Xavier Franch. “Requirements reuse and requirement patterns: a state of the practice survey”. In: *Empirical Software Engineering* 22 (2017), pp. 2719–2762. DOI: 10.1007/s10664-016-9485-x.

[101] David Lorge Parnas and Jan Madey. “Functional documents for computer systems”. en. In: *Science of Computer Programming* 25.1 (Oct. 1, 1995), pp. 41–61. ISSN: 0167-6423. DOI: 10.1016/0167-6423(95)96871-J.

[102] Oscar Pastor. “Model-Driven Development in Practice: From Requirements to Code”. In: *SOFSEM 2017: Theory and Practice of Computer Science*. Ed. by Bernhard Steffen et al. Cham: Springer International Publishing, 2017, pp. 405–410. ISBN: 978-3-319-51963-0. DOI: 10.1007/978-3-319-51963-0_31.

[103] Oscar Pastor and Marcela Ruiz. “From Requirements to Code: A Conceptual Model-based Approach for Automating the Software Production Process”. In: *Enterprise Modelling and Information Systems Architectures (EMISAJ)* 13 (Feb. 27, 2018), pp. 274–280. ISSN: 1866-3621. DOI: 10.18417/emisa.si.hcm.21.

[104] Shari Lawrence Pfleeger and Joanne M Atlee. *Software engineering: theory and practice*. Pearson Education India, 2010.

[105] PlantUML. [Online; accessed 2024-05-15]. URL: https://plantuml.com/.

[106] Amir Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[107] Klaus Pohl. *Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam.* Rocky Nook, Inc., 2016.

[108] Respect-IT. *Objectiver.* [Online; accessed 2024-05-16]. URL: https://objectiver.com/.

[109] Suzanne Robertson and James Robertson. *Mastering the requirements process: Getting requirements right.* Addison-Wesley, 2012.

[110] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical software engineering* 14 (2009), pp. 131–164.

[111] Andrey Sadovykh et al. "Security Requirements Formalization with RQCODE". In: *CyberSecurity in a DevOps Environment: From Requirements to Monitoring.* Springer, 2023, pp. 65–92. DOI: 10.1007/978-3-031-42212-6_3.

[112] Siemens. *Polarion REQUIREMENTS.* [Online; accessed 2024-08-21]. URL: https://https://polarion.plm.automation.siemens.com/products/polarion-requirements.

[113] John Smart. *BDD in Action: Behavior-Driven Development for the whole software lifecycle.* Simon and Schuster, Sept. 29, 2014. ISBN: 978-1-63835-321-8.

[114] IEEE Computer Engineering Society. *SWEBOK, version 3.0: Guide to the Software Engineering Body of Knowledge.* 2014.

[115] Eiffel Software. *EiffelStudio.* [Online; accessed 2024-05-16]. URL: https://www.eiffel.com/eiffelstudio/.

[116] Jama Software. *Jama Connect.* [Online; accessed 2024-07-17]. URL: https://www.jamasoftware.com/platform/jama-connect/.

[117] Ian Sommerville and Pete Sawyer. "Requirements Engineering: a good practice guide". In: *John Wiley and Sons* 113 (1997), p. 114.

[118] SPILEn. *Inspecto.* [Online; accessed 2024-05-16]. URL: https://spilen.fr/.

[119] Sparx Systems. *Enterprise Architect.* [Online; accessed 2024-07-17]. URL: https://sparxsystems.com/platforms/requirements_management.html.

[120] Daniel Terhorst-North. *Introducing BDD.* [Online; accessed 2024-06-11]. Sept. 2006. URL: https://dannorth.net/introducing-bdd/.

[121] TestNG. [Online; accessed 2024-03-09]. URL: https://testng.org/doc/.

[122] Julian Tschannen et al. "Autoproof: Auto-active functional verification of object-oriented programs". In: Springer, 2015, pp. 566–580. DOI: 10.1007/s10009-016-0419-0.

[123] Iris Vessey and Sue Conger. “Learning to Specify Information Requirements: The Relationship between Application and Methodology”. In: *Journal of Management Information Systems* 10.2 (1993), pp. 177–201. DOI: 10.1080/07421222.1993.11518005.

[124] Wentao Wang et al. “Enhancing automated requirements traceability by resolving polysemy”. In: *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE. 2018, pp. 40–51. DOI: 10.1109/RE.2018.00-53.

[125] Muhammad Waseem and Muhammad Usman Sadiq. “Application of model-based systems engineering in small satellite conceptual design-A SysML approach”. In: *IEEE Aerospace and Electronic Systems Magazine* 33.4 (2018). publisher: IEEE, pp. 24–34. DOI: 10.1109/MAES.2017.180230.

[126] Tim Weilkiens. *Systems engineering with SysML/UML: modeling, analysis, design*. Elsevier, 2011.

[127] Jon Whittle et al. “RELAX: a language to address uncertainty in self-adaptive systems requirement”. In: *Requirements Engineering* 15.2 (June 2010), pp. 177–196. ISSN: 1432-010X. DOI: 10.1007/s00766-010-0101-0.

[128] Karl Wiegers and Joy Beatty. *Software Requirements*. 3rd. 2013.

[129] Claes Wohlin, Martin Höst, and Kennet Henningsson. “Empirical Research Methods in Web and Software Engineering”. In: *Web Engineering*. Ed. by Emilia Mendes and Nile Mosley. Berlin, Heidelberg: Springer, 2006, pp. 409–430. ISBN: 978-3-540-28218-1. DOI: 10.1007/3-540-28218-1_13.

[130] W. Eric Wong, Xuelin Li, and Philip A. Laplante. “Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures”. In: *Journal of Systems and Software* 133 (2017), pp. 68–94. ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.06.069.

[131] Jian Xie et al. “SysML-based compositional verification and safety analysis for safety-critical cyber-physical systems”. In: *Connection Science* 34.1 (2022). publisher: Taylor & Francis, pp. 911–941. DOI: 10.1080/09540091.2021.2017853.

[132] E.S.K. Yu. “Modeling organizations for information systems requirements engineering”. In: Jan. 1993, pp. 34–41. DOI: 10.1109/ISRE.1993.324839.

[133] Tao Yue, Lionel C. Briand, and Yvan Labiche. “Facilitating the Transition from Use Case Models to Analysis Models: Approach and Experiments”. In: *ACM Trans. Softw. Eng. Methodol.* 22.1 (Mar. 2013). publisher-place: New York, NY, USA publisher: Association for Computing Machinery, pp. 1–38. ISSN: 1049-331X. DOI: 10.1145/2430536.2430539.

[134] Tao Yue and et al et. “aToucan: an automated framework to derive UML analysis models from use case models”. In: *ACM Trans. on Soft. Eng. and Methodology (TOSEM)* 24.3 (2015). publisher: ACM New York, NY, USA, pp. 1–52. DOI: 10.1145/2699697.

[135] Tao Yue et al. “A practical use case modeling approach to specify crosscutting concerns”. In: *Software Reuse: Bridging with Social-Awareness: 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings 15*. Springer. 2016, pp. 89–105. DOI: 10.1007/978-3-319-35122-3_7.

[136] Pamela Zave. “Classification of Research Efforts in Requirements Engineering”. In: *ACM Comput. Surv.* 29.4 (1997), pp. 315–321. ISSN: 0360-0300. DOI: 10.1145/267580.267581.

[137] Pamela Zave and Michael Jackson. “Four dark corners of requirements engineering”. In: *ACM transactions on Software Engineering and Methodology (TOSEM)* 6.1 (1997), pp. 1–30. DOI: 10.1145/237432.237434.