



LOUVAIN
School of Management

Impact of Unified User-Story-Based Modeling on Agile Methods

Aspects on Requirements, Design and Life Cycle Management

Samedi HENG

Doctoral Thesis **01** | **2017**



Impact of Unified User-Story-Based Modeling on Agile Methods: Aspects on Requirements, Design and Life Cycle Management

by **Samedi Heng**

A thesis submitted in fulfillment of the requirements for the degree of

Doctor in Economics and Management Sciences

of the Université catholique de Louvain

Examination Committee:

Prof. Manuel Kolp (UCLouvain), Advisor

Prof. Yves Wautelet (KULeuven), Co-Advisor

Prof. Jean Vanderdonckt (UCLouvain), Examiner

Prof. Isabelle Mirbel (Université Nice Sophia Antipolis), Examiner

Prof. Vincent Englebert (UNamur), Reader

Prof. Per Agrell (UCLouvain), President of the jury

February 2017

To my parents,

*for your hard work and sacrifices to support our family and, more importantly,
your vision that only better education can improve our standards of living to
increased enjoyment and happiness.*

Acknowledgements

As long as I remember, writing a PhD thesis has always been a dream. It has been a unique experience comparable to no others I have had before. It took me six years to finally made it to the end. This would not have been possible without the precious help and encouragements from the people I thank hereafter.

First, I would like to express my sincere gratitude to my supervisors Profs. Manuel Kolp and Yves Wautelet. Yves once told me that the path to the PhD is made of ups and downs; downs can be the periods when most of the improvements are there to put yourself into question which is, by nature, the essence of any scientific work. My personal path has also been made of these with a down peak in 2013. Both of you helping me back on my feet in a hard but fair manner made of interventions and coaching but also encouragements, advices, and amusement.

Manuel, many thanks for the opportunities you have offered me: the teaching and research assistant position at the *Louvain School of Management (LSM)* and *Louvain Research Institute in Management and Organizations (LouRIM)* which led me to the honor of being one of your PhD students. You have provided me with the chance, support and liberty to achieve this work. This, of course, has also impacted my private life because I not only found a PhD supervisor in you but also a true and sincere friend. I would like also to extend my acknowledgements to your wife, Ai Vi, for her hospitality and kindness.

I deeply respect and admire Yves! You have provided me continuous and constructive feedback on my research to ultimately build this thesis. It would truly not have been possible to achieve and finish this work without your help, Yves; the contents, approach and writings have tremendously evolved because of your rigorous guidance. Many thanks for your time, coaching and the many discussions we have had along the way.

Next, I thank the other members of the jury, Profs. Isabelle Mirbel, Jean Vanderdonckt, Vincent Englebert and Per Agrell for accepting to participate in the jury of this thesis and their valuable feedback. Jean in particular, thank you for your gentleness, encouragement, and simple and clear explanations.

My sincere thanks also go to Prof. Christelle Scharff for allowing me to participate in the global software development project from 2011 to 2013 from which I have learnt about user stories, agile methods and the case study used in this thesis, and Prof. Stephan Poelmans and Velghe Mattijs for the help building the contributions with respect to the experimentations on user stories included in this thesis.

A warm regard also goes to Sandrine Delhaye for helping and facilitating my PhD process.

My gratefulness also goes to my colleagues in *Center of Management Information Systems (CEMIS)* at LouRIM. I especially want to thank Prof. Marco Saerens, Thanh-Diane Nguyen, Soreangsey Kiv, Sodany Kiv, Sylvie Baudine, Mathieu Zen, Iyad Khaddam and Jorge Luis Perez Medina for their helpful discussions regarding teaching, research and also social life.

I am grateful for the help of Thavorac Chun for implementing the CASE-Tool during his research stay at CEMIS in 2014 during 2 months.

My sincere appreciation also goes to Vietnamese, Taiwanese, Pilipino and, of course, Belgian friends for being friendly and sharing fun with our family.

I am very thankful to Cambodian friends in Belgium and the Netherlands who have helped our family and have made our life joyful during all these years. In particular, Samnang Nary, thank you for your assistance and care to our family.

Last but not least, I would like to thank my family: my parents, parents-in-law, wife and children, for their support and encouragement, and above all, their care and love. Thank you, Rachana, for accepting to be part of my PhD journey. I know you have had a hard time adapting to the live in a country with a culture and language you were not familiar with but in the end you were able to adapt and enjoy Belgian life, so far away from where we were born. It took me longer than I expected, but I made it for you and our family. Therefore, this PhD thesis is also dedicated to you! Marissa and Matisse, thank you for your everyday smile. *Love you all!*

Samedi Heng

Abstract

User Stories (US) are the most commonly used requirements artifacts within agile methods such as XP and Scrum. They are written in the form of text of maximum two lines in natural language using prose or following a specific template. Traditionally, they are written down on an index card and posted on a wall or whiteboard for analyzing and monitoring the progress of the project. This performs well when the number of US is limited; but not so well when the number of US is large. In practice, many templates have been proposed with no semantic associated to each syntax used in the templates. This leads to series of issues looking alike different stakeholders interpreting the purpose of the US differently and results in making the whole set difficult to structure and analyze. In the end, this translates in misinterpretation of requirements.

This thesis starts with studying existing US templates in order to build a (consistent) unified model for their use with, for each keyword, a syntax and associated semantic. Particularly, this thesis studies US templates following the structure: *As [WHO], I want [WHAT], so that [WHY]*. Additionally, we also build a derived graphical representation for representing a US set to better analyze dependencies and alternatives of US. This notably allows building a view of the system-to-be on multiple aggregation level. A coarse-grained view can help the development team to better manage the life cycle of the application development by selecting relevant US to be developed within the coming iteration(s). Additionally, a fine-grained view allows dealing with the details of the functions to be developed.

This thesis is organized as follows. **Part I** introduces the work. **Part II** presents the related literature review—i.e., agile software development, requirements engineering and requirements engineering in agile methods. **Part III** exposes the unified model for US template which is the pillar of the thesis and is used as a foundation for the other contributions of the thesis. The unified model was constructed based on empirical data collected from the web and scientific literature. The adopted concepts are the *Role* in the WHO dimension; the *Capability*, *Task*, *Hard-goal* and *Soft-goal* in the WHAT dimension; and, finally, the *Task*, *Hard-goal* and *Soft-goal* in the WHY dimension. We also provide a well-defined semantic associated to each of these syntaxes; most of them come for the i* framework. **Part IV** presents the *Rationale Tree (RT)*, a goal-based graphical representation for representing US sets. It indeed allows analyzing the dependencies between US set in a US set—this allows us to better understand the implications in terms of business value offered by their development as well as to get a multiple level view of the system-to-be. Basically, a RT is built out

of a US set tagged following the proposed unified model. This part also exposes an empirical study on the ability of understanding the constructs of the unified model for tagging a US set and ultimately building a RT out of a US set. The experimentations were conducted with 3 groups: *IT Students*, *Business Students* and *Researchers*. The results have shown that, when tagging a US, identifying the right concept—especially for functional elements where granularity plays an important role—can only be done when evaluating a consistent US set as a whole and thus not on an individual basis. **Part V** provides an alternative approach for graphically representing a US set based on the (industry adopted) Use-Case model. The latter model is nevertheless intended to only provide a coarse-grained representation of the system-to-be. Finally, **Part VI** concludes the thesis and discusses future work.

Table of contents

List of figures	xiii
List of tables	xv
I Introduction	1
1 Introduction	3
1.1 Research Context	3
1.2 Reading Map and Contributions	5
1.3 Limitations	6
II Literature Review	9
2 Agile Software Development	11
2.1 The Emergence of Agile Methods	11
2.2 Agile Methods	13
2.2.1 The <i>Manifesto</i> for Agile Software Development	13
2.2.2 Agile Methods in Practice	15
2.3 Overview of the Main Agile Methods	18
2.3.1 eXtreme Programming	18
2.3.2 Scrum	22
2.4 Conclusion	25
3 Requirements Engineering: an Overview	27
3.1 Requirements Engineering Basic Notions	27
3.1.1 Requirement: Definition	27
3.1.2 From Requirement to Requirements Engineering	28
3.2 Abstraction Levels for Requirements Representation and Management	28
3.2.1 Dimensions in Requirements Engineering	28
3.2.2 Modeling Requirements	29
3.3 Requirements Engineering: Dynamic Perspective	30
3.3.1 Basic Stages and Areas in Requirements Engineering	30
3.3.2 Towards a Requirements Engineering Process	33
3.4 Categories of Requirements	34

3.4.1	Functional Requirements	34
3.4.2	Non-functional Requirements	34
3.4.3	Quality Requirements	35
3.4.4	Features of High-Quality Requirements	35
3.5	Using Natural Language for Requirements	36
3.5.1	Natural Language: Pros and Cons	36
3.5.2	Use of Structured Natural Language	37
3.6	Using a Graphical Model for Representation Requirements	38
3.6.1	Overview	39
3.6.2	The UML Use-Case Model	40
3.6.3	The i* Framework	43
3.7	Conclusion	48
4	Requirements Engineering in Agile Methods	49
4.1	Requirements Engineering Activities in Agile Methods	49
4.2	User Stories: the Requirements Artifacts of Agile Methods	50
4.2.1	User Story Overview	51
4.2.2	Features of High-Quality User Stories	51
4.2.3	User Story Templates	54
4.2.4	User Story, Epic, and Theme	56
4.2.5	User Story versus others User Requirements Artifacts	56
4.2.6	Pros and Cons of using User Stories	61
4.3	Visualizing and Modeling Requirements with User Stories	62
4.3.1	User Role Modeling	63
4.3.2	The Product Backlog	63
4.3.3	User Story Mapping	65
4.3.4	Models and User Stories	66
4.4	User Story Based Planning in Agile Methods	68
4.4.1	Iterative Planning with User Stories	69
4.4.2	Selecting User Stories for an Iteration	69
4.5	Conclusion	70
III	Towards More Formality in Agile Methods'	
	Requirements Engineering	73
5	Unifying and Extending User Story Models	75
5.1	Research Context	75
5.2	Related Work	76
5.3	Research Method	77
5.3.1	Building the Dataset	77
5.3.2	Descriptive_Concepts in User Stories	79
5.3.3	Building the Candidate Model	79
5.3.4	Validation	80
5.4	Selected Semantic Associated to the <i>D_C</i> Class Instances	80
5.4.1	The WHO Dimension	81
5.4.2	The WHAT Dimension	82
5.4.3	The WHY Dimension	85

5.5	A Unified Model for User Story Templates	86
5.6	Validation	87
5.7	Threats to Validity	88
5.8	Conclusion	88

IV Graphically Representing User Story Elements: Identifying Granularity, Interdependencies and Scope of Requirements 91

6	Building a Rationale Tree for Evaluating User Story Sets	93
6.1	Research Context	94
6.2	Related Work	94
6.3	Research Method	96
6.3.1	Macro-level: Ways to Organize User Stories	96
6.3.2	Micro-level: Decomposing a User Story in Descriptive_Concepts	97
6.4	Graphical Notation for US Dependency Analysis: Micro-Level .	98
6.4.1	The WHO Dimension: Graphical Notation	98
6.4.2	The WHAT and WHY Dimensions: Graphical Notation	99
6.4.3	Linking Descriptive_Concepts of the Unified User Story Model	99
6.5	Towards a Rationale Analysis for User Stories Hierarchy and Grouping: From Micro to Macro Level	100
6.5.1	A Top Level Hard-goal (End), One Mean	101
6.5.2	A Top Level Hard-goal (End), Several Means	103
6.5.3	A Top Level Task, a Direct Decomposition	105
6.6	Impact on the Agile Software Process	106
6.6.1	Impact of Changing Requirements	106
6.6.2	Impact Iterative Planning	106
6.6.3	Generic Iterative Planning Template	107
6.7	A CASE-Tool for Automating the Approach and Round-Tripping Between Views	108
6.8	Validity, Threats to Validity, Scalability of the Approach and Future Work	109
6.9	Conclusion	111
7	On the Interpretation of Granularity and Interdependencies of User Story Elements with the Rationale Tree	113
7.1	Research Context	113
7.2	Research Method	114
7.3	Feasibility Study Design	114
7.3.1	Process for Building the Feasibility Study	114
7.3.2	Assignment and Measured Variables	115
7.3.3	Case Studies	116
7.4	Data Collection	119
7.5	Analyzing the Results	120
7.5.1	The Knowledge of Participants in Modeling	120

7.5.2	The Tagging of User Story Elements	122
7.5.3	Analyzing the User Story Model with Rationale Tree . .	125
7.5.4	Analyzing the Experience of Test Subjects	132
7.6	Limitations of the Feasibility Study	137
7.7	Conclusion	137
 V An Alternative Graphical Representation for User Story Elements: Suitability of the Industry-Adopted Use-Case Model		139
8	Bridging User Story Sets with the Use-Case Model	141
8.1	Research Context	141
8.2	Related Work	142
8.3	Running Example	143
8.4	User Stories Integration through a Use-Case Diagram	143
8.4.1	The Role	143
8.4.2	Hard-goal, Task and Capability	144
8.4.3	The Soft-goal	145
8.5	Automating the Approach and Round-Tripping Between Views	147
8.6	Impact on Produced Software: Future Work	148
8.7	Conclusion	149
 VI Conclusion		151
9	Conclusion	153
9.1	General conclusions	153
9.2	Summary of the main Contributions	153
9.3	Future Work	154
9.3.1	Improvement on User Story Model	154
9.3.2	Using Rationale Tree in Software Development Process	156
 References		159
 Appendix A List of Publications		171
 Appendix B User Story Templates Dataset		173
B.1	Introduction	173
B.2	User Story Templates Set	173
B.3	Summary of User Story Templates' Elements	183
 Appendix C Descriptive_Concept's Definitions		185
 Appendix D Feasibility Study User Stories		191

List of figures

1.1	Reading map of the thesis.	6
2.1	Agile methods timeline.	13
2.2	Agile manifesto values.	14
2.3	The 2015 CHAOS report conducted by Sandish Group	16
2.4	Survey of most adapted agile methods	17
2.5	Survey of most adapted agile practice	17
2.6	XP’s project lifecycle	21
2.7	The burndown chart	23
2.8	Scrum’s project lifecycle	25
3.1	The three dimensions of requirements engineering	29
3.2	Level of requirements and their relationships.	29
3.3	Requirements engineering disciplines	31
3.4	Requirements in software development	32
3.5	Spiral requirements engineering process	34
3.6	A taxonomy of non-functional requirements	35
3.7	IEEE Std 830 document structure	37
3.8	Requirements template	38
3.9	Essential Use-Case elements	41
3.10	An example using modeling elements of Use-Case diagrams	41
3.11	The i* elements.	44
3.12	Strategic Dependency model for Meeting Scheduling	46
3.13	Rationale Dependency model for Meeting Scheduling	47
4.1	User story index card	51
4.2	Quality user story framework	53
4.3	Connextra user story card.	54
4.4	Requirements in IEEE-830 style	57
4.5	Scenario in Human-Computer Interaction Design	58
4.6	An example of a Persona	59
4.7	An example of Hierarchical Task Analysis	60
4.8	User Stories on Scrum Board.	64
4.9	Prioritized product backlog	64
4.10	User Story Mapping template.	66

4.11	Possible modeling techniques used as a complementing view to user stories.	68
4.12	General process for iterative planning in agile methods	70
5.1	Followed research process.	77
5.2	The Descriptive_Concept class.	79
5.3	Unified model for user story Descriptive_Concepts.	86
5.4	Elements coverage in the Carpooling and CalCentral case studies.	88
6.1	US as Macro-Level structures: Meta-Model.	97
6.2	US as Macro- and Micro-Level structures: Meta-Model.	98
6.3	Icons used within the representation of the user story elements using the Strategic Rationale reasoning.	99
6.4	Top-Level Hard-goal, One Means-End decomposition.	103
6.5	Top-Level Hard-goal, several Means-End decompositions.	105
6.6	Portfolio optimization problem.	107
6.7	The supporting CASE-Tool.	109
7.1	Research method for feasibility study.	114
7.2	Type solution of Case 1 in the feasibility study.	118
7.3	Possible solution of Case 2 in the feasibility study.	119
7.4	Average understandability score of the different elements.	124
7.5	Average scores on Case 1 and Case 2.	132
7.6	Understandability of the theory.	133
7.7	Average general perceived difficulty to model both cases.	135
7.8	Perceived difficulty by the test subjects.	136
7.9	Graph difficulty Case 1 versus Case 2.	137
8.1	Use-Case diagram: Canonical form and carpooling example.	146
8.2	The supporting CASE-Tool.	148
9.1	Process fragment for integrating Agent-Oriented development in agile methods.	156

List of tables

2.1	Dependencies of agile manifesto values and principles	15
3.1	Template for textual Use-Case documentation	43
4.1	Requirements engineering implementation in XP and Scrum . .	50
4.2	The five templates for writing user stories.	55
4.3	User story coverage and complexity.	56
4.4	The comparison between user story and other user requirements artifacts	60
5.1	Instances for Descriptive_Concept and related syntax.	81
6.1	US set 1 sample issued of the ClubCar application development.	102
6.2	US set 2 sample issued of the ClubCar application development.	104
7.1	US set in Case 1 of the feasibility study.	117
7.2	US set in Case 2 of the feasibility study.	118
7.3	Expertise of participants with i* framework.	121
7.4	Expertise of participants with user story.	121
7.5	Tagging of the US elements in Case 1.	122
7.6	Tagging of the US elements in Case 2.	123
7.7	Understandability of the difference between the elements. . . .	124
7.8	Kruskal-Wallis test on the understandability scores of the different elements.	125
7.9	Descriptive statistics of the number of elements and links modeled.	128
7.10	Kruskal-Wallis test on the modeled elements and links.	128
7.11	Evaluation criteria in quoting the US models.	129
7.12	Descriptive statistics of the global score.	130
7.13	ANOVA test on the global performance scores.	130
7.14	Results of the post-hoc test of Bonferroni.	130
7.15	The paired-sample t-test on the score of both cases.	132
7.16	Kruskal-Wallis test on the global perceived difficulty.	135
7.17	Kruskal-Wallis on the perceived difficulty of step 1 to 5.	136
8.1	US sample of the ClubCar application development.	144
8.2	Mapping a user story set with the Use-Case diagram.	147

B.1	Keywords for searching user story templates.	173
B.2	User story templates found in formal source.	174
B.3	User story templates found in informal source.	176
B.4	Syntax used in user story template	183
C.1	Selected syntaxes of Descriptive_Concept	185
C.2	Definitions of Role.	185
C.3	Definitions of User.	186
C.4	Definitions of Actor.	186
C.5	Definitions of Goal.	186
C.6	Definitions of Feature	187
C.7	Definitions of Functionality.	188
C.8	Definitions of Capability.	188
C.9	Definitions of Task.	188
C.10	Definitions of Activity.	189

Part I

Introduction

Chapter 1

Introduction

This chapter introduces the whole thesis. This chapter is organized as follows. Section 1.1 overviews our research context in software development. Section 1.2 provides the reading map and contributions of this thesis. Finally, Section 1.3 discusses the limitations of the thesis.

1.1 Research Context

For decades software has been developed in order to make use of the full potential of hardware. With the evolution of the ideas and concepts in the *Information Technology (IT)* field, the potential use of software systems has drastically grown. It is now of vital importance for organizations to have adequate IT tools; these at best bring a competitive advantage and at worse allow to compete with the same tools as competitors. In parallel, the complexity of the working environment has significantly evolved and continues to evolve leading to the necessity of having the right methodological tools within a defined IT adoption situation. Large software developments can indeed hardly be developed in an ad-hoc manner. Therefore, professionals need to follow a defined set of practices grouped in a development methodology. Nevertheless, if the latter is too rigid in its *Project Management (PM)* or poorly defines adequate *Engineering Practices (EP)*, its application may lead to an incapacity to adapt to requirements changes coming directly from the user or from the software environment.

In order to properly deal with scalability without killing all the potential for changes and innovations along the development life cycle, a right balance has thus to be found within the amount of PM activities as well as the relevant EP and their level of formality. Too rigid PM practices hamper communication with stakeholders (such as the ability to detect changes that need to be made) as well as the management of the implementation of the changes identified. On the opposite, no PM at all leads to chaos. Similarly, well designed requirements, design, implementation, test, . . . models (part of the method's EP) allow to represent complex software problems and solutions in relevant manner. Ideally, these should allow an adequate and aligned representation of what users are truly expecting from the system. If such models are expressed in natural language, everything can be said or expressed. If they are expressed in a very formal manner, they lose part of their expressiveness but will be more

straightforwardly to use (or even possibly automated) for forward engineering activities, consistency checking, traceability and other EP. The right balance has thus to be found depending on the nature of the covered software problem.

In software problems of limited size where innovation is important, little or no PM can be performed in order to allow immediate adaptation of the software product when changes occur in the user requests or enterprise environment. The method can then essentially focus on EP. If the software problem is broad(er) in size, the software solution is likely to support (more) complex processes. In such a case, PM activities are necessary in order to coordinate the different parts of the system; formal EP including precise software modeling also increase in importance in order to coordinate (engineering) activities and be able to capture all of the aspects of the software problem and solution. In such a case, iterative and incremental development through sets of short iterations allows PM activities to guide the production of relevant artefacts yet allowing to make changes and learn through user experience. Finally, in huge software projects, PM activities are a must have and advanced EP should allow the representation and realization management of complex processes on multiple levels of granularity. Iterative and incremental development is here also from primary importance but since the software problem is larger in size, the scope elements for each iteration can be broader with a possible impact on the iterations' duration.

The initial aim of this doctoral thesis is to overview the potential improvements that can be made to agile methods on the basis of their requirement representation artifacts (which is part of its EP). We take the perspective that such improvements can only be identified/shown/proved if one or a set of dependable elements are changed at a time. Existing agile methods have reached various maturity levels and, within the improvements that we suggest in this thesis, the aim is not to evaluate this maturity or redefine entirely a method but rather to be able to anchor to existing methods to inherit its benefits and try to address some of its issues. This means that we intend to plug-in to existing agile software development methods to bring improvements on defined elements (starting from EP but also with an impact on PM).

More precisely, starting from the discussed findings and this research point of view, this doctoral thesis studies how we can improve an agile software development methodology starting from one of its core EP namely user story-based modeling. *User Stories (US)* are known for decades in the world of agile software development, they are simple sentences written in natural language possibly following a defined template. More specifically, we focus on the ones following a template that relates a WHO, WHAT and (possibly) WHY dimension to depict user requirements. These are typically found in the *eXtreme Programming (XP)* and Scrum agile methods. The willingness is to improve the requirements engineering stage by better managing US in order to deal with scalability issues often reported as problematic in the world of agile methods.

For this purpose, US models and their structure need to be carefully analyzed and rationalized in order to exploit them in a systematic manner. The opportunity for more structuring of US and their constituting elements make sense only if filtering can be done on the basis of elements granularity. This is

the purpose of defining precise semantics associated with accurate syntaxes for tagging US elements. This way we can distinguish the nature and granularity of US elements to reach the first objective of this thesis *unifying US-based modeling*.

When unified US-based modeling can be reached, the opportunity for a graphical representation of US can be studied. *Goal-Oriented Requirements Engineering (GORE)* with its advanced representation abilities is chosen, in the doctoral thesis, for building a graphical representation of US elements in the form of a *Rationale Tree (RT)*. The interpretation of the US elements nature and grain is tested among groups of students and researchers. In parallel of US modeling, the industry has also adopted UML Use-Case models for the representation of coarse-grained requirements. Bridging those views is promising on the condition that they may be kept consistent. Together, these contributions aim at fulfilling the second objective of the thesis *improving the requirements engineering stage of agile methods through graphical representation of structured US elements*.

Finally, all these contributions, once applied within an agile method, have an impact on the overall PM of the agile project. More specifically, the graphical requirements models (i.e., through the RT or the Use-Case approach) built-up can be used for life cycle management. Indeed, scope elements can be identified graphically. They then can be used as scope elements for iterative planning within the planning game. Along the contributions, we will thus discuss the third contribution of the thesis *improving the life cycle management of agile methods through requirements models*.

1.2 Reading Map and Contributions

This thesis consists of six parts and nine chapters. Figure 1.1 provides the structure and reading map of the thesis. Between the parentheses of some chapters is the name of the conference in which the content of the chapter is published.

This doctoral thesis starts with a general literature review; Chapter 2 positions the domain of agile software development methods, Chapter 3 provides a basic overview on requirements engineering, and Chapter 4 overviews the requirements engineering techniques used in agile methods. The purpose of each of these chapters is to develop a general overview of the subject without relating each element of theory to a specific scientific contribution of the thesis.

The dissertation has, indeed, been built as a set of individual contributions presented individually from Chapter 5 to 8. For each of these chapters, the specific contributions, related work and research method are systematically discussed in the chapters themselves.

Chapter 5 overviews the different US templates available in formal (i.e., scientific publications) and informal (e.g., blogs, web sites, ...) sources. It associates semantics from GORE framework to each of the keywords (syntaxes) found. After a selection process, a unified (meta-)model for US templates is built. This contribution is validated on sets of examples. It constitutes the main pillar (foundation) onto which all the other contributions are built.

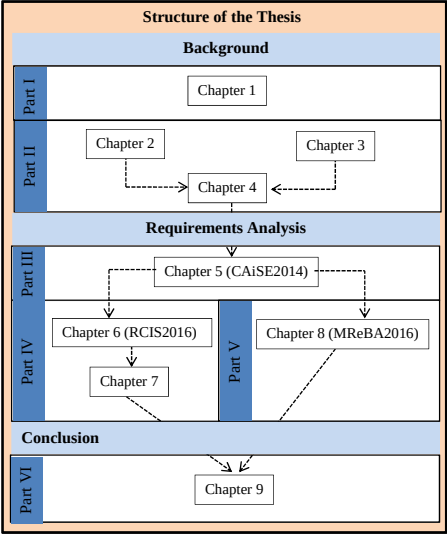


Fig. 1.1 Reading map of the thesis.

Chapter 6 overviews how a RT can be built using the US unified model. The tagging of US elements gives information about the nature and granularity of elements; with the use of that information a first graphical representation can be made. Then by domain analysis, US elements can be graphically linked for analysis. Consistency, alternatives, scope elements for PM, etc. can be studied from the graphical representation. The feasibility of the proposal is shown on an illustrative example in the field of carpooling.

Chapter 7 overviews the perception of software modelers of the RT built up in Chapter 6. Modelers are made of groups of students and researchers; their perception of the elements' granularity and the links they identify between the US elements is evaluated through their overall performance on two modeling exercises. The performance is then balanced through their modeling experience.

Chapter 8 overviews another graphical notation for US models. GORE models are, on the opposite of UML Use-Case models, far from being industry adopted. That is why we consider using the US template meta-model in order to build a UML Use-Case diagram serving as a complementary (and not concurrent) graphical view to the US set. The feasibility of the proposal is shown on an illustrative example in the field of carpooling.

1.3 Limitations

Once again, the limitations for each of the developed contributions are depicted on a case by case basis in the relevant chapters; we will here overview the limitations of model driven development based on US in the fashion of the contributions of the thesis.

The adoption of the contributions of the thesis in real life cases widely depends on the consistency of the input US sets as well of the adequate tagging and domain analysis performed on the basis of the US elements. The consistency in requirements can be evaluated using the RT but a completely inconsistent US set will require a lot of further domain analysis activities; this has a strong impact on the cost of applying that contribution.

We also highlight that:

- All the contributions developed in this thesis need to gain maturity through their use. In this dissertation, we indeed have applied most of the elements just on one to a few cases. However, it should be tested on more case studies so that the knowledge of the issues related to the practical application can be highlighted. This is particularly true for building the RT where only an accumulation of empirical data can lead to better guidelines and a higher level of automated support;
- We have several times identified scope elements for managing the iterative development life cycle but did not furnish a whole method in order to manage them. Custom work could be done in order to give ways to identify critical requirements to be prioritized first for prototyping in the iterative development life cycle;
- The contributions of this thesis are mostly designed for software problems not aiming to support heavy industrial processes with lots of realization paths. To such an end, other processes allowing workflow based modeling are often better suited.

Part II

Literature Review

Chapter 2

Agile Software Development

This chapter provides a brief and basic introduction to agile methods. The latter constitutes the software development methodology’s family in which the contributions of the thesis take place.

This chapter is structured as follows. Section 2.1 provides historical background on agile methods and their evolution. Section 2.2 exposes the agile manifesto and a recent survey on most adopted agile methods. Section 2.3 exposes two agile methods—i.e., eXtreme Programming and Scrum. Finally, Section 2.4 concludes the chapter.

2.1 The Emergence of Agile Methods

Since the emergence of software in the 50s, many software development methodologies have been proposed and developed in order to reduce *software cost* and produce software satisfying user needs with an adequate level of *quality* [87, 97]. Agile Methods have emerged in 2001 as a reaction to traditional development methodologies lacks; AgileManifesto [98] acknowledges the “*need for an alternative to documentation driven, heavyweight software development processes*”. Generally speaking, the traditional methodologies are characterized as plan-driven, document-oriented, process-oriented and based on formal communication [2, 85, 122, 137]. In ‘traditional’ software development methodologies, the new software is built using a predefined and extensive plan that is strictly followed during the development cycle [90]. In addition, each phase finishes with well-defined documents so that they can be used in the next phase by the same or another team [85, 127]. Just to name a few, traditional methodologies include the V-model [55], the Spiral Model [18], the *Rapid Application Development (RAD)* [99], or the *Rational Unified Process (RUP)* [82].

The waterfall model proposed by Royce in the 70s is the oldest development life cycle in the software engineering history [85]; it provides the basis for software development activities. The waterfall model consists of a series of sequential activities and each activity must be finished before the next one can start. It begins with elicitation and documentation of the *complete* set of requirements followed by software design, implementation, and testing; the software is fully delivered at once at the end of the project. In addition, each

step is planned in detail with formal documentation and strictly followed during the development [137].

The waterfall model works well for projects where requirements can easily be determined in advance, but does not perform well for the development of user intensive software projects and/or for key innovative projects. Indeed according to Larman and Basili [85], many waterfall-based projects reported failure during the 80s and recommended that software should be developed iteratively and incrementally. The perceptions of requirements as well as the requirements themselves are likely to evolve [48, 85, 125]. In addition, customers have difficulty in stating their needs at the beginning of the project and expect more from their software [85, 137, 159].

Developing software iteratively induces producing an executable release at the end of each iteration. Customers are thus able to evaluate a concrete product in order to provide feedback leading to refining or add new requirements for the next iteration. Meanwhile, developing software incrementally allows the same requirement to be addressed into multiple iterations starting with basic support and progressively developing into full workable software. Therefore, strictly defining and fixing requirements early on into the project is not a relevant practice in user intensive software development. Several methodologies such as Spiral, V-model, RAD and RUP were later proposed to fix the problem of the waterfall model. However, these methodologies are still process-oriented, document-oriented and plan-driven [2].

Meanwhile, practitioners in the industry were also seeking for new ways for developing software due to the aforementioned flows of the waterfall approach. They made experimentations with real software development teams using iterative and incremental development and other best practices in engineering field [2, 35]. As the result, many initiatives such as Scrum [132], *eXtreme Programming (XP)* [14], *Dynamic Systems Development Method (DSDM)* [139], Crystal [34], *Feature-Driven Development (FDD)* [113], etc. have been proposed under the name *lightweight methodologies* and later known as agile methods after the creation of the *agile manifesto* (see Section 2.2.1). These are *iterative*, *incremental*, *less-documented* and *people-oriented*. The latter methods are in fact inspired from the best practices of Toyota [61] which are described in the paper entitled “*The new new product development game*” [141].

Since the creation of the agile manifesto, many methods have been proposed under the category of ‘agile’. There are different views on what agility and agile mean. Nonetheless, most of agile methods are, in practice, a collection of different practices (or techniques) that share the same values and basic principles [35]. Figure 2.1 provides an overview of the evolution of agile methods; other agile methods exist in literature and practice. This figure has built from the presentation of Gaetano Mazzanti [ref¹] and based on the research of Abrahamsson et al. [4].

¹<http://www.slideshare.net/mgaewsj/agile-principles-agile-people>

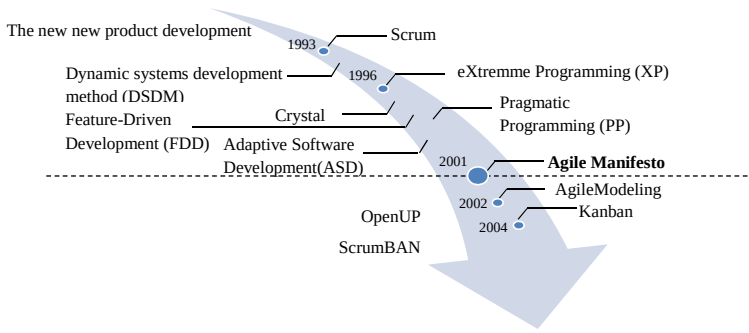


Fig. 2.1 Agile methods timeline.

2.2 Agile Methods

In this section, we explore the agile manifesto and discuss agile methods in practice. Section 2.2.1 exposes the agile manifesto and Section 2.2.2 provides the state of the art of agile methods in practice.

2.2.1 The *Manifesto* for Agile Software Development

The agile manifesto is the result of the meeting that took place at Utah, 2001 and made of seventeen software developers who are “*representatives from eXtreme Programming, Scrum, Dynamic Systems Development Method (DSDM), Adaptive Software Development (ASD), Crystal, Feature-Driven Development (FDD), Pragmatic Programming (PP), and others sympathetic*” [98]. The main goal of this meeting was to discuss the practices of each methodology that were successful in software development in the late 90s and to try to understand the common ground of each methodology. According to Sommerville [137], the manifesto is “*a set of principles encapsulating the ideas underlying agile methods of software development*”. The **manifesto for agile software development** consists of four values and twelve principles.

As shown in Figure 2.2, the manifesto is written in a structured sentence with the word *over* in the middle. The right parts are written in normal style, whereas the left parts are written in bold. This means that agile software development recognizes the values in the right part; additionally, it emphasizes more on the values in the left part [7].

As can be derived from Figure 2.2, agile methods focus on individuals and interactions more than on processes and tools. Hunt [68] states that project success relies mostly on the people involved and the way in which they communicate more than on the processes, methodologies and tools that are used. In these methodologies, producing a working piece of software is more important than producing comprehensive documentation. This second principle entails three main implications for agile approaches [64]. First of all, since the project team focuses on the development of a system, only those documents that are actually required for the development process are produced. Secondly, the actual development of the system (i.e., coding) is started as soon as possible



Fig. 2.2 Agile manifesto values.

to improve the understanding of both developers and clients concerning the end product that has to be developed. The third and final implication concerns the certainty for the users that they get a bugless software solution that meets all their needs. The two other values comprise the intensive customer involvement and participation and the ability to respond to change.

The twelve principles of the manifesto are listed below. These principles are used for guiding the software development to support the four values of the manifesto. Table 2.1 shows dependencies between agile manifesto values and principles.

1. *“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software;*
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage;*
3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale;*
4. *Business people and developers must work together daily throughout the project;*
5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done;*
6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation;*
7. *Working software is the primary measure of progress;*
8. *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely;*

9. *Continuous attention to technical excellence and good design enhances agility;*
10. *Simplicity—the art of maximizing the amount of work not done—is essential;*
11. *The best architectures, requirements, and designs emerge from self-organizing teams;*
12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly” [98].*

Table 2.1 Dependencies of agile manifesto values and principles (from [77]).

Value \ Principle	1	2	3	4	5	6	7	8	9	10	11	12
Individuals and interaction over process and tools	x			x	x	x		x				x
Working software over comprehensive documentation	x		x				x		x	x	x	
Customer collaboration over contract negotiation	x	x		x				X				
Responding to change over following a plan	x	x	x				x					(x)

2.2.2 Agile Methods in Practice

Methods for agile software development consist of a set of practices and principles for software development that have been created by experienced practitioners. Agile methods can be seen as a reaction to plan-based or traditional methods which emphasize “*a rationalized, engineering-based approach*” [49].

Since the creation of the agile manifesto in 2001, agile methods have gained a lot of popularity due to their success in software development. According to the 2015 CHAOS report [63] published by the Sandish Group, agile methods are more successful and less failing comparing to waterfall-based ones. Figure 2.3 exposes the result of the CHAOS report related to success and failure of using agile and waterfall methods in software project.

CHAOS RESOLUTION BY AGILE VERSUS WATERFALL				
SIZE	METHOD	SUCCESSFUL	CHALLENGED	FAILED
All Size Projects	Agile	39%	52%	9%
	Waterfall	11%	60%	29%
Large Size Projects	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Medium Size Projects	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Small Size Projects	Agile	58%	38%	4%
	Waterfall	44%	45%	11%

The resolution of all software projects from FY2011-2015 within the new CHAOS database, segmented by the agile process and waterfall method. The total number of software projects is over 10,000.

Fig. 2.3 The 2015 CHAOS report conducted by Sandish Group (from [63]).

According to the report conducted by the VersionOne in 2016 [149], Scrum has become the most popular agile methodology with a use of 58% in overall agile projects and XP has become unpopular with a use of only 1% (see Figure 2.4). XP has nevertheless been famous in the world of agile methods; it encompasses a lot of relevant engineering practices of software development like for example the *User Stories*, *Pair Programming*, *Planning Game*, *Open Workspace*, *Continuous Integration* and *Collective code ownership*. These techniques are still popular and used in many other agile methods. This can be seen in Figure 2.5; however, some names have been slightly changed but consist of the same practice. In addition, a practice as *user stories* does not appear but in fact, *user stories* are the main artifacts for performing *Story Mapping*. Therefore, *user stories* are also part of the most used techniques.

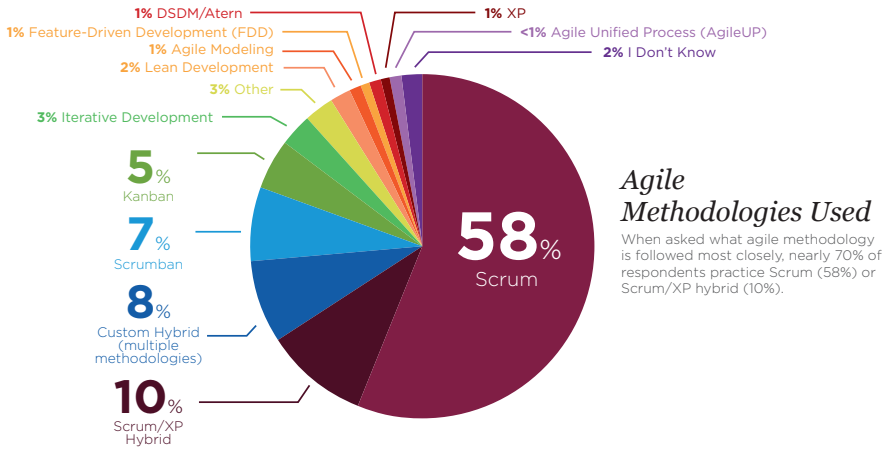


Fig. 2.4 Survey of most adapted agile methods (from [149]).

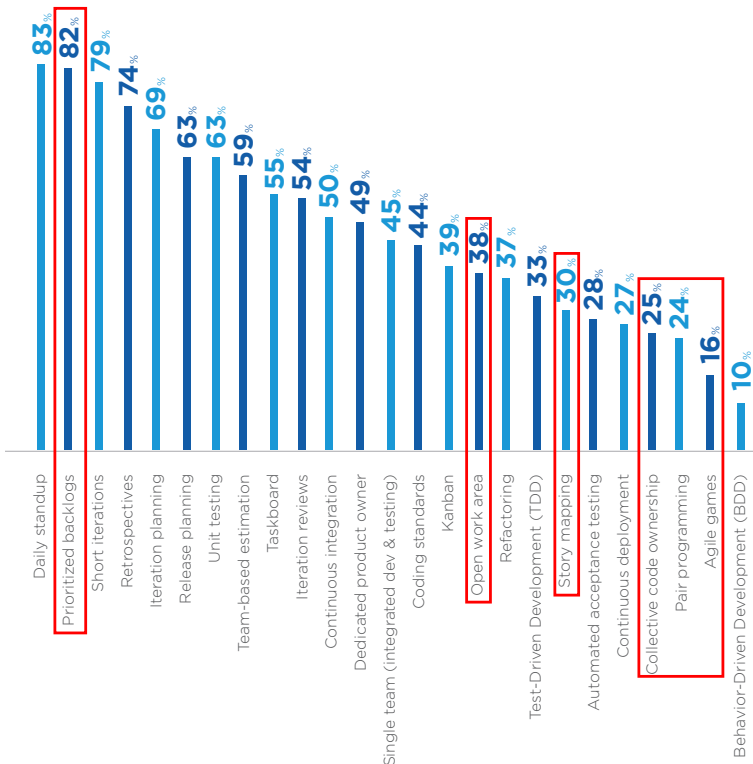


Fig. 2.5 Survey of most adapted agile practice (from [149]).

2.3 Overview of the Main Agile Methods

Within this section, we provide an overview on the two agile methods—i.e., *eXtreme Programming* (XP) and Scrum. Interested readers about other agile methods could refer to [3, 4, 35, 49, 61]. Section 2.3.1 reviews the XP methodology and Section 2.3.2 reviews the Scrum methodology. The XP has been chosen because it focuses on relevant engineering practices for agile development while the Scrum integrates project management in the same family of methods.

2.3.1 eXtreme Programming

XP could be considered as a primary methodology that made agile software development get public attention; it was a dominant methodology in the late 90s to 2000 before the rise of Scrum [ref²]. XP had been developed by Kent Beck during his consulting time with Smalltalk projects in the late 80s and early 90s [15, 24]. It was created as an answer to the long development cycle of the waterfall model. XP focuses on software quality improvement and responsiveness to vague or rapid changes in customer's requirements [4]; it has been built based on values of *simplicity, communication, feedback, courage and respect* [15]. The characteristics of XP are *short development cycles, incremental planning, continuous feedback, reliance on communication, and evolutionary design* [15]. It promises to reduce project risk, improve responsiveness to business changes, improve productivity throughout the life of the system, and add fun to building software in teams—i.e., all at the same time [24].

2.3.1.1 XP Roles

Following Beck and Andres [15], XP defines seven roles for running the XP project:

- *Programmer*: Programmers write the code and ensure it is simple to the best of their knowledge. They normally work in pair for writing the code;
- *Customer*: Customers write user stories, give the priority to each user story and set the scope of the implementation. Then, they write the test scenarios for each user story and make the acceptance tests;
- *Tester*: Testers help customers write the test scenarios and do the testing of the project progressively;
- *Tracker*: Trackers help programmers and customers better estimate their efforts for implementation by giving feedback on each iteration;
- *Coach*: Coach is the person who is responsible for the whole XP process. The coach guides programmers and customers to be in the XP process;
- *Consultant*: Consultant is an external person who possesses the specific knowledge that could help programmers when they encounter technical problems;

²<http://martinfowler.com/bliki/ExtremeProgramming.html>

- *Manager (Big Boss)*: Manager takes the final decisions during the XP process.

2.3.1.2 XP Artifacts

Following Beck and Andres [15], the artifacts produced within an XP project are:

- *User Story Card*: It is an index card that contains a requirement in form of a user story. It also contains the short description of requirements, priority, effort, and test scenarios. Programmers use it for implementation, discussion and planning;
- *Task List*: It is a listing task that programmers need to build in order to accomplish a user story. This helps programmers to estimate the effort and planning;
- *CRC Card*: CRC stands for Class-Responsibility-Collaboration. It contains the responsibilities and collaborators of classes (Object-Oriented) that allow the team to do the design of the system;
- *Customer Acceptance Test*: The test scenarios that customers write in order to validate the implementation. Normally, it is writing on the user story card;
- *Visible Wall Graphs*: It is a graphical board on which the progression of the team is displayed—e.g., *how many user stories are under development, tested and accepted?* It allows the team to communicate and see the progress of the project. It is displayed publicly to everybody in the team.

2.3.1.3 XP Practices

Following Beck and Andres [15], the 13 practices of XP are:

- *Planning*: This practice represents the meeting between customers and programmers for the planning; it happens once per iteration. The planning game consists of two steps. First, programmers estimate the effort needed for implementation of each user story, second customers decide on the scope of the iteration. Finally, they together determine the release date;
- *Small/short release*: This practice encourages programmers to build small and workable parts of the system for customers in a short period of time. As a result, customers are able to give feedback early. If the release is accepted, it is ready to *go-live*. If not, a new version will be built in the next iteration;
- *Metaphor*: A metaphor is “*an object, activity, or idea that is used as a symbol of something else*” [ref³]. Instead of a formal architecture, XP uses a metaphor as a simple common vision of how the system works. It is simple, so everybody understands it and can use it to guide their design;

³<http://www.merriam-webster.com/dictionary/metaphor>

- *Simple design*: The team is recommended to produce the simplest possible solution that is implementable for now. The complex and extra code is not necessary and it needs to be discarded immediately. Programmers regularly ask themselves, *is it the simplest solution they have got?*
- *Testing*: Every function needs to be tested. The test plans are written by customers before the implementation. The test of each function has to be conformed to the test scenarios;
- *Refactoring*: At the end of each iteration, the team needs to restructure the system by removing duplication, improving communication, simplifying and adding flexibility. However, the behaviors or functionalities of the system are kept unchanged;
- *Pair Programming*: Two programmers work together, sitting side by side, on the same code on a single computer. One writes the code and the other reviews it. This results in better design, better testing, and better code;
- *Collective ownership*: The code in XP is not belonging specifically to any pair programmers, but to everybody in the team. Anyone is able to change any part of the code. This practice could be dangerous to some extent if someone works blindly on code (s)he does not understand;
- *Continuous integration*: The new code is integrated into the current system as soon as it is ready. The system is thus built many times a day; the integration test must pass for each new build. This ensures that the system keeps running properly;
- *40-hours week*: XP limits the working hour for programmers to only 40 hours maximum per week. No two overtime weeks in a row are allowed;
- *On-site customer*: The XP methodology requires customers to sit with programmers to answer questions, resolve disputes, and set priorities. It is not necessary with the real costumers, but it could be proxy-customers;
- *Just rule*: Programmers need to respect several rules defined by the team. It is not a standard and rigorous rule. Teams can change rules to adapt to the culture of the team members. What is important is that everybody in the team understands and accepts to use these rules;
- *Open workspace*: It is preferable for the XP team to work in an open space. The configuration of the room should enable face-to-face communication of developers and direct discussion.

2.3.1.4 XP Process

Following Abrahamsson et al. [3], the XP process could be viewed as divided in six phases of which Figure 2.6 gives a graphical representation. These phases are:

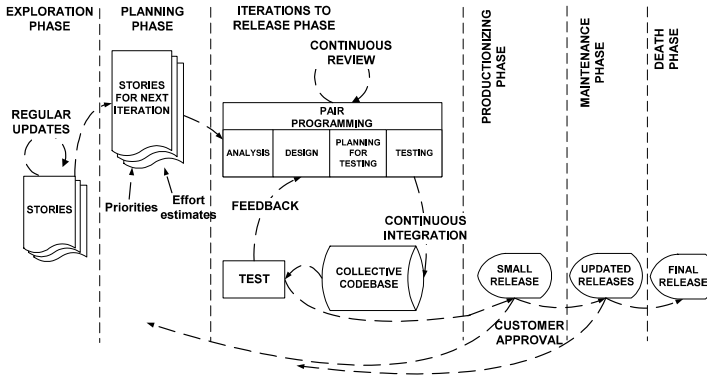


Fig. 2.6 XP's project lifecycle (from [3]).

- *Exploration:* During this phase, customers write requirements in the form of user story cards—i.e., function or functionality they want for the system. Each user story card consists of one and only one function or functionality. It is not necessary to have a complete list of the user stories in this phase, they can be added afterwards when the project evolves;
- *Planning:* Developers and customers, together, do the planning—i.e., the developers estimate the efforts for implementing each user story and customers, on their side, give the priority to user story and set the scope of the iteration;
- *Iterations to release:* Developers start to build the architecture of the system for the first iteration and then followed by a subsequent iteration of coding, testing and integration. Every new piece of function is tested and validated with customers before go-live. It is important in XP to keep the system running during the development;
- *Productionizing:* More integrated tests need to be done during this phase. The team members have to make sure that the system runs well with new functions. Developers have to repair the problem as soon as possible. Changes are still found during this phase and it could be treated in the next iteration;
- *Maintenance:* Customers use the system while developers keep developing and improving the system. If customers find a malfunction or unsatisfied function, they can inform developers team. It is possible for them to add new user stories. Developers, on their side, try to treat new user stories as early as possible in the following iteration;
- *Dead:* It is the final phase of the project. This phase could be happen in two different ways. If customers are happy with the system and have no more user stories to add. The system keeps running and generates business value for customers; it is a successful project. Therefore, the

team members produce some documentation for the system. If customers are not happy with the implementation they decide to stop the project; it is a failure project.

2.3.2 Scrum

The Scrum methodology was introduced for software development by Jeff Sutherland and later got the interests from Mike Beedle and Ken Schwaber in the early of the 90s. This methodology is inspired from the paper entitled “*The new new product development game*” [141]. The term ‘scrum’ comes from a strategy in the game of rugby where it denotes “*getting an out-of-play ball back into the game*” with teamwork [140].

Following Abrahamsson et al. [4], Scrum focuses on the project management aspects for software development. It does not provide software development techniques, methods, and practices for the implementation process; it is free to developers to use any specific practice. Scrum works well together with XP and also with other agile methods [ref⁴]. The three pillar concepts of Scrum are *transparency*, *inspection*, and *adaption* [142]. Another important aspect of Scrum is to have a common definition on what is defined as a *completed item* [142]. Basically, Scrum is an iterative and incremental method based on short time-box iterations (known as the black-box) of maximum 30 days which is known as a ‘sprint’; each sprint produces a piece of workable software.

Originally, Scrum only consists of practices. Later, the authors were inspired by the values of agile manifesto and finally came out with five values. Following Schwaber and Beedle [132], the five Scrum’s values are *commitment*, *focus*, *openness*, *respect* and *courage*.

2.3.2.1 Scrum Roles

Following Abrahamsson et al. [3], and Tsui et al. [142], the Scrum methodology principally consists of the three following roles:

- *Scrum Master*: Scrum Master is a new role in software development that was introduced by Scrum. This role ensures and keeps track of the team to follow the values and practices of Scrum, and facilitates every event of Scrum to keep the team productive. The Scrum Master is not a manager role, but it is rather a coach of the team; it is also part of the development team. The Scrum Master interacts with the development team, customers and also the management during the project;
- *Product Owner*: This role represents officially the voice of customers and ensures that the team delivers value to the business; the latter role is normally selected by the Scrum Master, customers, and management. The product owner is responsible for creating and prioritizing the Product Backlog, setting the scope of the sprint, and reviewing and accepting the software at the end of each sprint;

⁴<https://www.scrumalliance.org/community/spotlight/mike-cohn/april-2014/scrum-xp-better-together>

- *Development Team or Scrum Team*: It is the development team of the project; it is usually constituted of 3 to 10 developers. It is a self-organizing team that is involved in estimating, creating the Product Backlog (described in the next section), decomposing task for implementation of each Product Backlog item, and reviewing Product Backlog, etc.

2.3.2.2 Scrum Artifacts

Scrum defines three main artifacts to be used for controlling the project [142, 162]; sometimes, they are also called as practices of Scrum [3]. These artifacts are openly accessible by everybody in the team [162] and are the following:

- *Product Backlog*: It is a list that stores the current requirements of the product; it does not constitute the complete requirements set of the software because the items that have already been implemented are discarded from the Product Backlog and other items can be added afterwards. An item of the Product Backlog normally consists of a feature, a functionality of the system and its priority; sometimes, it is a user story card;
- *Sprint Backlog*: It is like to the Product Backlog, but it is for a sprint; it is the subset of the Product Backlog. The Product Owner decides and selects which Product Backlog item should go into the current Sprint Backlog for the implementation; normally, it will produce a workable software at the end of the sprint. Every item in the Sprint Backlog is further decomposed in a set small task and each task is assigned to specific developer;
- *Burndown Chart*: It is the chart that displays the remaining efforts of the project. It is a tool for measuring the progress in software development [ref⁵]. It is updated at the end of each sprint by the Scrum Master. Figure 2.7 exposes the Burndown Chart.

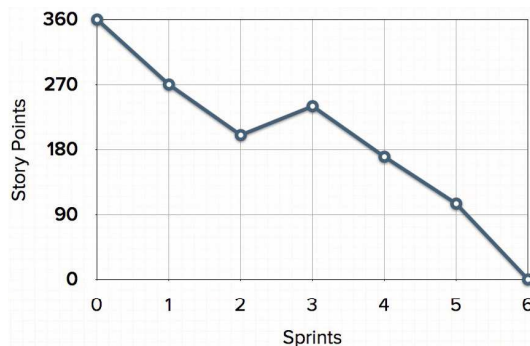


Fig. 2.7 The burndown chart (from [ref⁶]).

⁵<http://www.mountaingoatsoftware.com/agile/scrum/release-burndown>

2.3.2.3 Scrum Practices

Following Tsui et al. [142], the Scrum consists of four practices, sometimes called Scrum Events. These practices are:

- *The Sprint Planning Meeting*: It is happened before each new sprint starts; it involves Product Owner, Scrum team and Scrum Master. They define a general goal of each sprint which is a high-level success criteria for that sprint. The Product Owner gives the priority to the items in the Product Backlog and selects the items that are valuable for the current business and driven by the sprint goal; these items are later stored in the Sprint Backlog. The Scrum Team decomposes the Sprint Backlog items into small tasks and reorganizes the team for accomplishing the Sprint Backlog. Requirements changes and re-prioritizations are not allowed during the sprint;
- *Daily Scrum*: It is a short meeting (maximum 15 minutes) that occurs every day during the sprint. It is often called ‘Scrum’. The goal of the meeting is to make sure that the team members are on the track. It is a session of asking three famous questions: 1) *What have you done since the last Scrum?* 2) *What will you do between now and the next Scrum?* 3) *What got in your way of doing work?* If a member encounters a problem, (s)he can ask for help from others. This meeting is normally organized by the Scrum Master;
- *Sprint Review*: It is held one or two days before the end of the sprint. It is achieved for presenting the result of the sprint to the Product Owner, and customers; however, this event is informal. The participants review and assess the new release. During this review, participants determine what is finished and what is not; this results in productivity of the team which is measured as velocity, then the Scrum Master can update the Burndown Chart. The result of this review could bring out the new requirements—i.e., the Backlog items, and sometimes change the direction of the system being built;
- *The Sprint Retrospective*: It is held immediately after the Sprint Review; it does not take a long time. The Scrum Master and Product Owner are recommended to join this meeting. The team considers three things: *What went well?* *What didn’t?* and *What improvements could be made for the next sprint?* It is essential for the team to have this meeting for the self-evaluation and self-improvement.

2.3.2.4 Scrum Process

Following Abrahamsson et al. [3], the Scrum process could be viewed as divided in three phases of which Figure 2.8 gives a graphical representation. These phases are:

- *PreGame*: Scrum starts with the requirements gathering by creating the Product Backlog. It is not required to have a complete Product

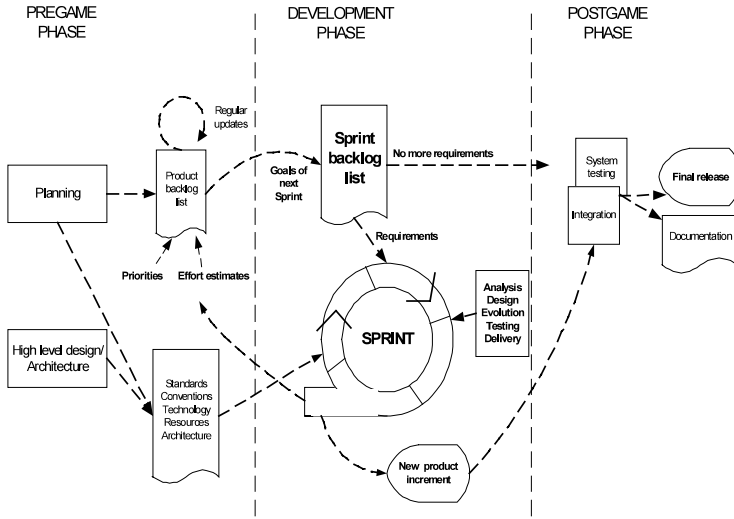


Fig. 2.8 Scrum's project lifecycle (from [3]).

Backlog; the Product Backlog can be updated after each sprint. This phase is separated into two sequential steps: *Planning* and *Architecture*. During the planning, each item in the Product Backlog is estimated with priority and efforts for implementing that item, and the scope of the sprint is set—i.e., moving the selected items from Product Backlog to Sprint Backlog, with schedule and cost. The architecture defines how the items in the Sprint Backlog are implemented; it involves the system architecture modification and high level-design;

- *Development:* This phase is also called *Game Phase*. It is the phase that is known as the 'black-box'; the team commits to the implementation of the Sprint Backlog. Any change including requirements, priority, efforts, etc. are not allowed;
- *PostGame:* It is the end of the project. The project can enter into this phase when there are no new requirements and clients agree on the system. It is time for the integration the whole system, global testing and documentation.

2.4 Conclusion

This chapter has presented the state of the art of the agile methods. Agile methods are a collection of methodologies that follow the four values and twelve principles of the agile manifesto. The manifesto aims at guiding agile methods rather than being a methodology on its own. Statistically speaking, agile methods are more successful than traditional methods. As a consequence, many agile methods have been proposed over the years. The agile methods are

generally characterized as *iterative*, *incremental*, *less-documented* and *people-oriented*. In addition, this chapter has reviewed two agile methods—i.e., XP and Scrum—which are relevant to this thesis.

Chapter 3

Requirements Engineering: an Overview

This chapter defines, positions, and overviews the different aspects of the *Requirements Engineering (RE)* discipline. Basically, it focuses on relevant concepts of RE related to our research context.

This chapter is structured as follows. Section 3.1 positions RE with respect to the overall software engineering field. Section 3.2 discusses different levels of abstraction of requirements. Section 3.3 explores the dynamic dimension of requirements—i.e., the RE process. Section 3.4 exposes different categories of requirements. Section 3.5 explains the advantage of using natural language for writing requirements. Section 3.6 overviews the modeling approach for documenting requirements. Finally, Section 3.7 concludes the chapter.

3.1 Requirements Engineering Basic Notions

This section aims at providing general concepts related to RE. Section 3.1.1 defines the *requirement* concept, while Section 3.1.2 discusses different definitions of RE.

3.1.1 Requirement: Definition

The Merriam-Webster defines a *requirement* as “*something that is needed or that must be done*” or “*something that is necessary for something else to happen or be done*” [ref¹]. This general definition of requirement highlights the notions of need and necessity. We will further overview the notion in the field of software development in the rest of the section.

The standard definition of requirement is provided by the Institute of Electrical and Electronics Engineers (IEEE standard 610–Glossary of Software Engineering Terminology) [71]. It is defined as “(1) *A condition or capability needed by a user to solve a problem or achieve an objective.* (2) *A condition or capability that must be met/possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documents.* (3) *A documented representation of a condition or a capability as in (1) or (2) above*”.

¹<http://www.merriam-webster.com/dictionary/requirement>

Following Hull et al. [67], IEEE24765:2010 [73], Pohl [119] and Van Lamsweerde [145], requirements can be defined as *unambiguous, testable and measurable statements*—based on the system-as-is and new technologies—that comprise a set of *objectives, conditions, properties, capabilities, qualities, constraints and assumptions* concerning the system-to-be. Within this context, the system-as-is comprises the current system that (partially) needs to be replaced by a new one (i.e., the system-to-be) for several possible reasons.

3.1.2 From Requirement to Requirements Engineering

RE is an emerging field that becomes a subarea of software engineering. The goal of RE is to make the requirements stage in software system development more systematic and disciplined [165]. The concerns of RE are the elicitation, modeling, evaluation, specification, analysis, communication, documenting and evolution of the objectives, functionalities, qualities and constraints of the system to be developed [65, 145]. Nowadays, a lot of definitions for RE are available in literature.

One of the oldest definition of RE is provided in [167]; RE is defined as “... the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families”.

Van Lamsweerde [145] defines RE as “a coordinated set of activities for exploring, evaluating, documenting, consolidating, revising and adopting the objectives, capabilities, qualities, constraints and assumptions that the system-to-be should meet based on problems raised by the system-as-is and opportunities provided by new technologies”.

Based on a wide range of other definitions in literature, RE is a process where stakeholders and their needs concerning the system-to-be (i.e., requirements) are discovered, defined and that subsequently are modeled, analyzed, negotiated and documented.

3.2 Abstraction Levels for Requirements Representation and Management

This section focuses on the abstraction of requirements. Section 3.2.1 exposes the three dimension of RE. Section 3.2.2 overviews the three level abstraction of requirement.

3.2.1 Dimensions in Requirements Engineering

According to Van Lamsweerde [145], the problem world can be divided into three different dimensions. RE is responsible for finding out why a system-to-be is necessary (i.e., the WHY dimension), what problems need to be solved (i.e., the WHAT dimension) and who has an interest in the system-to-be (i.e., the WHO dimension). These different dimensions are represented in Figure 3.1.

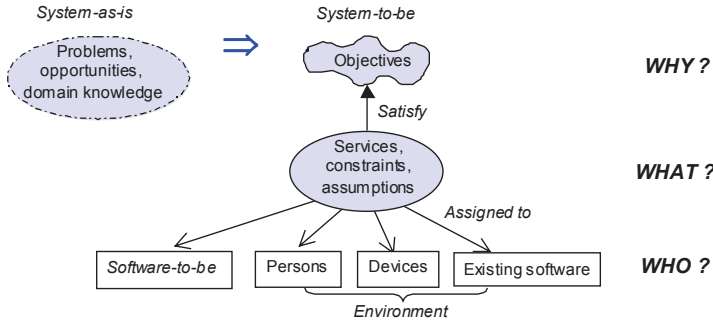


Fig. 3.1 The three dimensions of requirements engineering (from [145]).

3.2.2 Modeling Requirements

Requirements in software development can be represented at various levels of abstraction—i.e., different levels of abstraction and detail [137]. Good separation of requirements abstraction provides a better mean for communicating among various types of stakeholders [137]. Following Sommerville [137], and Wiegiers and Beatty [161], software requirements could be classified into three levels of abstraction: *business requirements*, *user requirements*, and *system requirements*. This is shown in Figure 3.2.

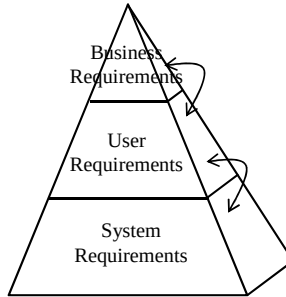


Fig. 3.2 Level of requirements and their relationships.

3.2.2.1 Business Requirements

Business requirements are high-level requirements. They describe why the organization is implementing the system—i.e., “*the business benefits the organization hopes to achieve*” [161]. Requirements are normally expressed in term of visions, goals, and business’ objectives.

3.2.2.2 User Requirements

User requirements specify what services the system is expected to provide and the constrains under which it must operate so that end-users can perform their

tasks or archive their *goals* [137, 161]. They are the result of the requirements elicitation task with clients and end-users. User requirements can be materialized as statements written in natural language with formal and informal digrams [83, 137]. In practice, the IEEE-830, Use-Cases, User Stories, and event-response tables are used for writing user requirements [161].

3.2.2.3 System Requirements

System requirements are derived from user requirements; they are therefore more detailed than user requirements and describe what exactly has to be implemented [137]. The system requirement's documents (sometimes called functional specification [137]) are the result of the analysis of developers or software analyst during the design phase. A mathematical model or a graphical model such as data-flow diagrams, object class hierarchies, etc. are generally used for system requirements [137]. An appropriate analysis allows a good derivation of system requirements from the user ones which is important to ensure that the design of the system always fulfills client's needs [96].

The terms *system* in this sense is not just any information system. It can be referred to all software or it can include both software and hardware subsystems. Van Lamsweerde [145] further distinguishes this concept into **system requirement** and **software requirement**. A **system requirement** is a statement formulated in terms of environmental phenomena that prescribes something that has to be implemented in the system-to-be in a cooperation with other system components. For example, "*All train doors shall always remain closed while a train is moving*" is a **system requirement** and "*The doors' state output variable shall always have the value 'closed' when the measured speed input variable has a non-null value*" is a **software requirement**. The latter kind of requirements comprise statements that are to be enforced solely by the system-to-be and these requirements are only formulated in terms of phenomena that are shared between the software and its environment. In this thesis, we do not distinguish both concepts and we use them interchangeably.

3.3 Requirements Engineering: Dynamic Perspective

This section focuses on the activities involved in RE and its process. Section 3.3.1 exposes the activities involved in RE. Section 3.3.2 presents the RE process.

3.3.1 Basic Stages and Areas in Requirements Engineering

Following Wiegers and Beatty [161], RE could be divided into two main areas: *requirements development* and *requirements management*. The purpose of requirements development is to identify, analyze, agree upon, and record requirements. This discipline is further composed of four activities: *requirements elicitation*, *requirements analysis*, *requirements specification*, and *requirements validation*. The purpose of requirements management is simply to manage changes in requirements. Figure 3.3 depicts the relationship between different RE activities; these activities are explained in this section.

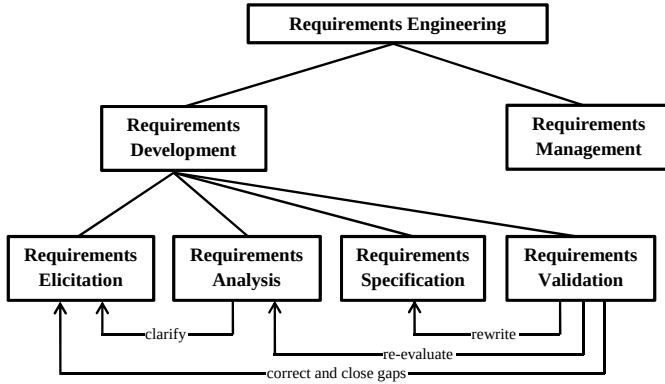


Fig. 3.3 Requirements engineering disciplines (from [160]).

3.3.1.1 Requirements Elicitation

The requirements elicitation activity is concerned with understanding the application domain, services the system should provide, system performance, hardware constraints, etc. [137]. The clear understanding of the needs of stakeholders, the system constraints as well as the limitations and deficiencies of the system-as-is are of critical importance [13, 32, 52, 108]. Information that has been gathered during this activity often has to be interpreted, analyzed, modeled and validated. That is also the reason why this activity of eliciting requirements is often related to the other activities of the RE process [108]. According to Chemuturi [31], Paetsch et al. [112], Pohl [119], Van Lamsweerde [145], and Wiegiers and Beatty [161], there are, in practice, many techniques that can be used for eliciting requirements from stakeholders; these include interviews, Use-Cases, scenarios, user-task elicitation, user stories, observation and social analysis, focus groups, brainstorming sessions, prototyping.

Many authors describe the elicitation of requirements as being the start of the RE process, Van Lamsweerde [145] nonetheless describes another task prior to this activity of eliciting requirements as being the first RE activity: *the domain understanding*. This task consists of analyzing the strengths and weaknesses of the system-as-is and the identification and understanding of the ‘problem domain’. Consequently, the domain understanding is concerned with the rationale and motives for developing and implementing a new system. During this activity it is of primary importance to get a clear view on the relevant stakeholders that need to be involved in obtaining a good understanding of the business environment in which the system-as-is is situated. The importance of a good domain understanding in RE is actually also recognized by [32, 52, 112], but, unlike Van Lamsweerde, these authors describe the domain analysis as being part of the elicitation activity.

3.3.1.2 Requirements Analysis

During the requirements analysis activity, requirements gathered in the elicitation activity are *analyzed* and *modeled*. It consists of checking and analyzing requirements for necessity, consistency, completeness and feasibility. Furthermore, conflicting, overlapping and omitted requirements are identified. Conflicting stakeholder concerns, that result in inconsistent requirement specifications, are solved through prioritization. Through dialogue with customers, a priority is attributed to the requirements. An analysis of both risk and impact of requirements is also included in this activity [52, 112].

Within this activity, modeling tasks are also performed. Models make requirements easier to understand because they imply the possibility of collecting, processing, organizing and analyzing smaller amounts of relevant information. In practice, there are many techniques, modeling notations and languages. The most popular ones are *Unified Modeling Language (UML)* [44], *Specification and Description Language (SDL)* [138], *Structured Analysis Structured Design (SASD)* [163], Petri Nets [104], Goal-based techniques [145, 165], etc.

3.3.1.3 Requirements Specification

Requirements specification is aimed at communicating the requirements between stakeholders and developers by means of the requirements document which directly results from activities performed within the different processes in the RE cycle. In this activity, requirements and features of the system-to-be, which are the result of the previous activities, are detailed, structured and documented [161]. The resulting requirements document is used in different activities of software development as shown in Figure 3.4.

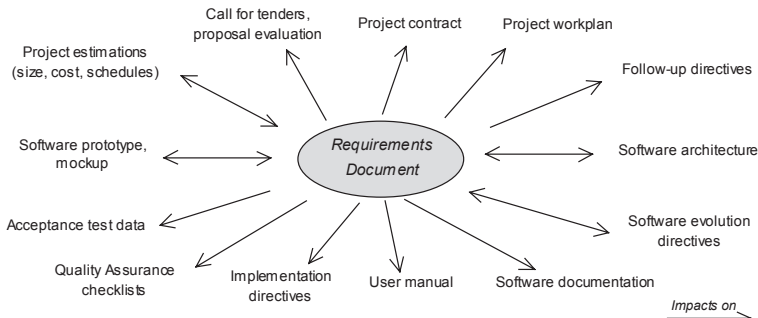


Fig. 3.4 Requirements in software development (from [145]).

In practice, requirements could be documented in three forms: *informal*, *semi-formal* and *formal* [120, 137, 145]. The *informal* form consists of using natural language or ad-hoc diagrams for documenting requirements. The *formal* form consists of using a rigorous mathematical basis or well-defined graphical notation with well-defined semantic for documenting requirements. Finally, the *semi-formal* consists of using natural language augmented with some (graphical) notations in formal approach [83]. Following Van Lamsweerde [145], the semi-

formal approach remains the most used technique for documenting requirements in software projects.

3.3.1.4 Requirements Validation

Requirements validation aims at validating and verifying requirements. It ensures that all gathered, modeled and documented requirements actually fulfill the need of all different stakeholders in an accurate and complete way. This activity is performed by using the requirements document, organizational standards and organizational knowledge as input [112]. According to Bourque and Fairley [22], techniques used for requirements validation are requirements reviews, prototyping, model validation and acceptance tests.

3.3.1.5 Requirements Management

Requirement management is a process in itself. It comprises all activities that are associated with change control, version control, requirements tracing, and requirements status tracking [161]. Traceability is a technique used to keep the relationships between requirements, design, and implementation of a system in order to manage changes.

3.3.2 Towards a Requirements Engineering Process

According to Van Lamsweerde [145], there are *data dependencies* among the four activities of *requirements development*. This means that requirements analysis requires input from requirements elicitation; requirements specification requires inputs from requirements analysis and requirements validation requires inputs from requirements specification [145]. In practice, these activities are very often intertwined and interrelated with possible overlap; therefore, these activities should not be applied in a strict sequence [137, 145, 161]. The process used for requirements development is widely dependent on the software process and methodology used for project [161]. In traditional software development with a waterfall life cycle, these activities are also categorized as *phases* since every activity has to be completed before the next one can be started.

In practice, the RE process is often iterative [137]. Van Lamsweerde [145] and Sommerville [137] represent the RE process with all its activities as being driven by a spiral model (see in Figure 3.5). Every iteration—as presented in Figure 3.5—is caused by a need of revising, adapting or extending the requirements that have already been identified, documented and validated. This trigger for a new iteration can either occur during the RE process itself, during other phases in the software development or even after the new system has been released and implemented.

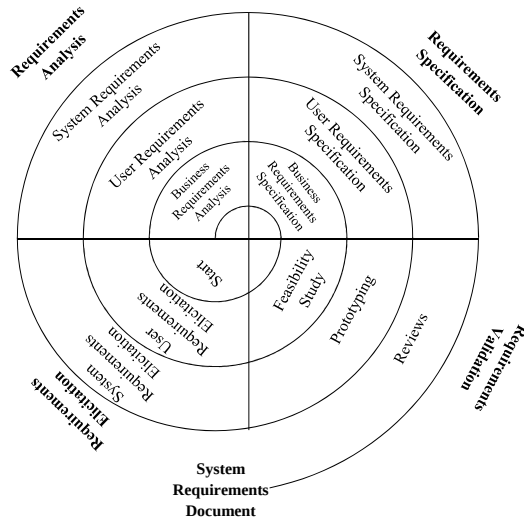


Fig. 3.5 Spiral requirements engineering process (adapted from [137]).

3.4 Categories of Requirements

Requirements are traditionally classified into two kinds: *Functional* and *Non-functional requirements* [83, 137]. Recently a new category has been added and studied in the RE research community: the *Quality requirements* [145]. This section is structured as follows. Section 3.4.1 overviews the functional requirements. Section 3.4.2 describes the Non-functional requirements. Section 3.4.3 provides descriptions of quality requirements. In addition, we provide the characteristics of high-quality of requirement in Section 3.4.4.

3.4.1 Functional Requirements

Functional requirements describe services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations [137]. These requirements address the *what* aspects depicted in Figure 3.1. They describe what the developers must implement to enable users to accomplish their tasks [161]. In addition, functional requirements can be high level and general (user requirements) or they can be detailed, expressing inputs, outputs, exceptions, and so on (system requirements) [83].

3.4.2 Non-functional Requirements

There are various definitions for non-functional requirements. The basic definition refers to them as statements for quality aspects of the system such as “-ilities” (e.g., usability) or “-ities” (e.g., integrity) or some others (e.g., performance, user-friendliness, coherence). Sommerville [137] defines non-functional requirements as “*constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and*

constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services”.

Van Lamsweerde [145] defines non-functional requirements as “constraints on the way the software-to-be should satisfy its functional requirements or on the way it should be developed”. He also provides the taxonomy classification for non-functional requirements (see Figure 3.6).

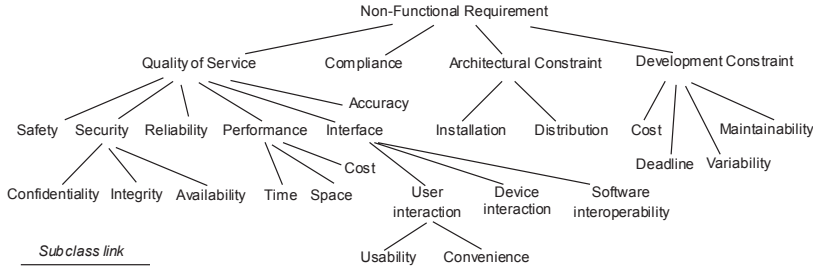


Fig. 3.6 A taxonomy of non-functional requirements (from [145]).

In addition, the difference between functional and non-functional requirements is that a functional requirement is completely satisfied and non-functional requirements are only satisfied up to a certain level.

3.4.3 Quality Requirements

According to Van Lamsweerde [145], quality requirements complement the *what* aspects with *how well* aspects of requirements. He defines quality requirements as “additional, quality-related properties that the functional effects of the software should have”. Quality requirements are previously known as ‘quality attributes’ of non-functional requirements; they appear on the left-hand side of Figure 3.6.

3.4.4 Features of High-Quality Requirements

Concerning high-quality requirements, there are various lists of criteria for writing good requirements provided by different authors; and most of them are similar. According to Bijan et al. [17], Hull et al. [67], IEEE29148:2011[74], Roman [126], and Van Lamsweerde[145], high quality requirements must be *complete*, *precise*, *pertinent* and *adequate* so that the system-to-be satisfies all objectives and actually tackles all problems that have been identified using the system-as-is as input. *Unambiguity* and *consistency* are two other important features of good requirements. It should not be possible to interpret a requirement in more than one way and all requirements should be compatible (i.e., non-contradictory). For the sake of sound project management, requirements have to be *measurable* and have to be implemented within a certain budget and schedule (i.e., *feasible*). Requirements also have to be *testable* since they are used to check whether or not the designed software product provides a solution for the defined problems and needs. Furthermore, requirements should be specified in a *structured* way so that they are *comprehensible* for all stakeholders involved in the software development project. Requirements should be written in *short*

sentences and short paragraphs in an active voice and only one process verb per sentence and *each sentence possesses only one requirement* [120].

However, it is not possible to fulfill all the aforementioned characteristics in practice. For example, it is hard to have complete requirements at the beginning of the project [54]. Stakeholders have their bounded rationality to express their needs during a limited time frame; new needs always appear when stakeholder see and use system [159].

3.5 Using Natural Language for Requirements

In this section, we provide an overview of using *Natural Language (NL)* in RE. Section 3.5.1 discusses the basic characteristics of NL in RE. Section 3.5.2 explains the use of structured NL to overcome some challenges of prose NL.

3.5.1 Natural Language: Pros and Cons

NL, particularly prose, has been wildly and commonly used for writing requirements since the beginning of software engineering [120, 137]. It consists of using our daily language such as English, French, Dutch, German, etc. for formulating and documenting requirements [120].

According to Pohl [120], Sommerville [137] and Van Lamsweerde [145], the prose provides several advantages over other approaches for expressing and writing requirements. It is *expressive*, *intuitive* and *universal*. The latter means that NL can be used for writing requirements in any area and domain. Intuitive refers to NL is basically an every-day language; therefore requirements can be understood by any stakeholder without additional training [120, 145]. Finally, expressive refers to the fact that NL provides no limitation for writing any kind of requirements [145].

On the other hand, NL are notoriously prone to many problems [120, 161, 145]; basically, it can be an obstacle to get good requirements (see Section 3.4.4). The drawback could result in risks and/or possible pitfalls in succeeding a software project [43]. The major disadvantage of NL is *ambiguity*—i.e., ambiguities are inherent to NL. Ambiguity in NL refers to the possibility to interpret them in more than one way; this can be harmful for a software project [101, 145]. When using NL for documenting requirements as a whole, additional drawbacks appear. The most frequent ones are *forward references* and *remorse*. Forward references refers to “*requirements document item making use of problem world features that not defined yet*” and remorse refers to a “*requirements document item stating a problem world feature too late or incidentally*” [101, 145]. Moreover, it is also hard to localize specific information in a long list of requirements. Last but not least, it is hard to build a supporting tool allowing to do automated analysis due to the absence of formalization.

Sommerville [137] recommends simple guidelines for writing requirements in order to reduce the aforementioned problem; they are described as follows:

- Build a standard format and ensure that all requirements definitions adhere to that format;

- Use language consistently to distinguish between mandatory and desirable requirements. For example, use ‘shall’ for mandatory requirements and ‘should’ for desirable requirements;
- Use text highlighting (e.g., bold, italic, or color) to pick out key parts of requirements;
- Do not assume that reader understands a technical software engineering language. The jargon, abbreviation and acronyms must be avoided for writing requirements;
- Try to associate a *rational* with each requirement. The rational should explain why the requirements have been included.

3.5.2 Use of Structured Natural Language

According to Sommerville [137], *Structured Natural Language (SNL)* “*is a way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way*”. Following Van Lamsweerde [145], there are two rules that have to be respected when writing and documenting requirements: the *global* and the *local* rule.

The global rule is concerned with how the requirements documents should be organized as a whole. Basically, these include rules for grouping related requirements and a global template for standardizing the requirements structure; this is commonly known as *Software Requirements Specification (SRS)*. In practice, we use the template proposed by the standardization institute such as IEEE-830 Standard [70]. Figure 3.7 is an example of requirement document template. However, this thesis is not concerned by the global rule; it is more concerned on the local rule.

Table of Contents
1. Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions, acronyms, and abbreviations
1.4 References
1.5 Overview
2. Overall description
2.1 Product perspective
2.2 Product functions
2.3 User characteristics
2.4 Constraints
2.5 Assumptions and dependencies
3. Specific requirements (See 5.3.1 through 5.3.8 for explanations of possible specific requirements. See also Annex A for several different ways of organizing this section of the SRS.)
Appendixes
Index

Fig. 3.7 IEEE Std 830 document structure (from [70]).

The local is concerned about how each requirements statement is written. There are several techniques allowing writing each requirements statements

[145, 161]. As an example, we can use table for writing complex requirements, programming language or even a predefined template. Our research is more concerned about using predefined template. Pohl [120] defines a requirement template as a “*blueprint for the syntactic structure of individual requirements*”. Figure 3.8 is an example of template of how each requirement should be strictly written; more templates could be found in [120].

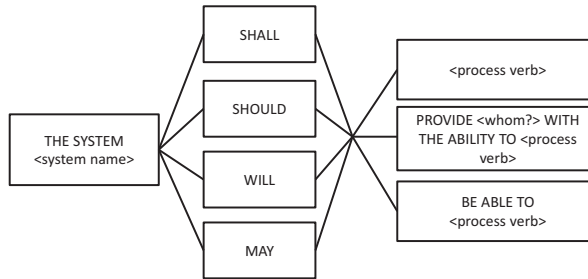


Fig. 3.8 Requirements template (from [120]).

A requirement template has several advantages to overcome the flows of NL described in Section 3.5.1. According to Pohl [120], Sommerville [137] and Van Lamsweerde [145], this technique is helpful for presenting requirements in a standardized form and so to build high quality requirements. The template increases the expressiveness, understandability, uniformity, and traceability. In addition, by using a template, the problem of ambiguity of requirements and variability of requirements can be reduced.

It is nevertheless sometimes difficult to write requirements by respecting a template for some particular problems [137]. For example, it is difficult to write requirements about how the system state change, how users interact with the system and how sequences of actions are performed. Other techniques such as a graphical model can better explain the problem. Following Pohl [120], using a template can be an obstacle to the creativity of the development team so that Pohl recommends to use requirement templates complementarily to other tools.

3.6 Using a Graphical Model for Representation Requirements

This section aims at reviewing the relevance of requirements graphical modeling languages. Specifically, we focus on the Use-Case model and the i^* framework. Section 3.6.1 provides background of graphical modeling languages. Specifically, we discuss on the advantages of *Goal-Oriented Requirements Engineering (GORE)* over the traditional RE. The next two sections focus on reviewing the basic notions of the both techniques. Section 3.6.2 briefly revises the Use-Case model. Finally, Section 3.6.3 provides the basic notions of the (GORE) i^* modeling framework.

3.6.1 Overview

Graphical modeling is a kind of modeling that uses a graphical notation in a model [137]. According to Wiegers [161], due to the limitation of short-term human memory, people have difficulty in analyzing long lists of requirements for inconsistencies, duplication, and extraneous. He argues that visual requirements models, in the sense of graphical models, can help us to identify missing, extraneous, and inconsistent requirements. Pohl [120] defines a model as “*an abstract representation of an existing reality or a reality to be created*”. Van Lamsweerde [145] defines a model as “*an abstract representation of the target system, where key features are highlighted, specified and inter-related to each other*”. The latter author states that a ‘good’ system model may act as a driving force for the RE process, because:

- *“It provides a comprehensive structure for what needs to be elicited, evaluated, specified, consolidated and modified;*
- *It allows us to abstract from multiple details of focus elicitation, evaluation, specification, quality assurance and evolution on key system aspects;*
- *It defines a common interface between those various RE activities, each acting as a producer or consumer of portion of it;*
- *It provides a basis for early detection and fixing of errors in requirements, assumptions and domain properties;*
- *It facilitates the understanding of complex systems and their explanation to stakeholders;*
- *It provides a basis for making decisions among multiple options and for documenting such decision;*
- *It defines the core RE artifact from which the requirements document can be generated”* [145].

According to Pohl [120], a graphical model is normally defined by two elements—i.e., *syntax* and *semantic*. Syntax refers to “*the syntax of a modeling language defines the modeling elements to be used and specifies the valid combinations thereof*” and semantic refers to “*the semantics defines the meaning of the individual modeling elements and serves therefore as a foundation for the interpretation of the models*”. The model could be informal, formal, and semi-formal depending on the degree of formalization of each model.

There are many graphical models existing in software engineering literature and requirements engineering literature [120, 137, 161]. Basically, we can classify those graphical models into two types: conventional and goal-oriented. Goal-oriented or GORE consist in using the goal concept as modeling element. The conventional model does not include the goal concept. Van Lamsweerde [145] define a goal as “*a prescriptive statement of intent that the system should satisfy through the cooperation of its agents*”. The most used conventional diagram are Use-Case model, activity diagram, state machine, data flow diagram, etc. Among

these most are industry adopted. GORE model nevertheless remain research frameworks that are not industry adopted. KAOS [145] and i* framework are examples of GORE framework.

GORE framework have emerged due to the lacks of traditional RE [84, 146]; it has been gaining interest and attention in the literature during the last decades [144]. According to Anton [10], Lapouchnian [84], Pohl [120], and Van Lamsweerde [144, 145], GORE has several advantages over traditional RE. In the traditional approach we can only capture WHAT and WHO dimensions, while GORE also deals with the WHY dimension. This allows GORE to build a deeper rationale behind a requirement which leads to a better understanding of the system-to-be. In addition, GORE can also improve the requirements process, better identify irrelevant requirements, provide more stable requirements, assure the completeness of project, and ease of communication among stakeholders.

3.6.2 The UML Use-Case Model

Use-Case modeling was originally proposed by Jacobson et al. [75] in the early 90s for modeling requirements in Object-Oriented software development and has been widely used as a technique to support requirements elicitation and analysis [23, 137]. The model aims at capturing the interaction between actors and the system. Fundamentally, the Use-Case model is based on two concepts that are used in conjunction with one another: the *Use-Case diagram* and the *Use-Case specification*. The Use-Case diagram has become a fundamental feature of the *Unified Modeling Language (UML)* [110]. In this thesis, we specifically review the UML Use-Case [110].

3.6.2.1 The UML Use-Case Model Elements

Figure 3.9 exposes the most essential modeling elements of the UML Use-Case model which are related to this research; the interested reader can refer to the OMG Unified Modeling Language Specification [110] for more information.

According to Pohl [120], the most important Use-Case model elements are:

- *Use-Case*: A Use-Case specifies some behavior that a subject can perform in collaboration with one or more Actors;
- *Actor*: An Actor models a type of role played by an entity that interacts with the subjects of its associated Use-Case. Actor may represent roles played by human users, external hardware, or other systems;
- *System boundary*: System boundaries within a Use-Case diagram separate the parts of the Use-Case that are part of the system from the parts (people or systems) that are outside the system boundary;
- *Extend relation*: An extend is a relationship from an extending Use-Case to an extended Use-Case that specifies how and when the behavior defined in the extending Use-Case can be inserted into the behavior defined in the extended Use-Case;

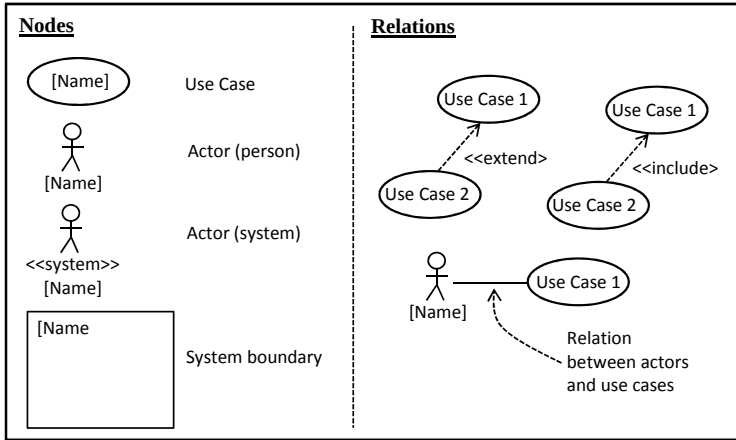


Fig. 3.9 Essential Use-Case elements (adapted from [120]).

- *Include relation*: Include is a directed relationship between two Use-Cases, indicating that the behavior of the included Use-Case (the addition) is inserted into the behavior of the including Use-Case.

3.6.2.2 The UML Use-Case Diagram

The Use-Case diagram is a composition of *actors* and *Use-Cases* with links to each others by the means of *relation* (see Figure 3.10). It allows to document the interrelations of the functionalities of a system and the relations between these functionalities and their environment from a user's perspective.

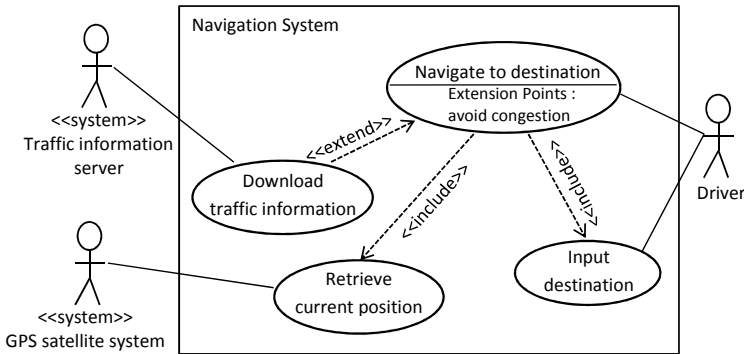


Fig. 3.10 An example using modeling elements of Use-Case diagrams (from [120]).

Figure 3.10 presents an example of Use-Case Diagram of a *Navigation System*. The model comprises of the Use-Cases **Navigation to destination**, **Download traffic information**, **Retrieve current position**, and **Input navigate**

to destination and the actors **Driver**, **Traffic information server**, and **GPS satellite system**.

Specifically, the *include* relations are used to link the Use-Case **Navigate to destination** with the Use-Cases **Input destination** and **Retrieve current position**. The relationship depicts that the interaction steps defined in the Use-Case **Navigate to destination** adhere to the interaction steps defined in the Use-Cases **Input destination** and **Retrieve current position**.

The Use-Case **Download traffic information** is related to the Use-Case **Navigate to destination** with the *extend relation*. The relationship depicts that the interaction steps defined in the Use-Case **Download traffic information** are included in the interaction steps of the Use-Case **Navigate to destination** when the condition **Avoid congestion** is attained. The extension point **Avoid congestion** depicts the steps in the Use-Case **Navigate to destination** at which the additional interaction steps are being executed.

3.6.2.3 Use-Case Specifications

Basically, the Use-Case diagram provides an overall view of the system—i.e., the system’s relevant functionalities from a user’s perspective and specific relationships between the functionalities of the system or between functionalities of the system and aspects in the system’s context. Nevertheless, the detail of interactions of a Use-Case are not documented in the Use-Case diagram; they are documented in other form, for example, by using a simple text description or a structured description in a table or a sequence diagram [137].

In general, the details of a Use-Case are documented in a textual specification with a defined template. There are many templates available in the literature; we specifically present, in Table 3.1, a reference template for writing a Use-Case Specification. An explicit example of Use-Case specification can be found in [120, 137, 161].

Essentially, the Use-Case Specification template contains the following attributes:

- Attributes for unique identification of Use-Cases (rows 1 and 2);
- Management attributes (rows 3 through 7);
- Attribute for the description of the Use-Case (row 8);
- Specific Use-Case attributes—e.g., the trigger event (row 9), actors (row 10), pre- and post-conditions (rows 11 and 12), the result of the Use-Case (row 13), the main scenario (row 14), alternative and exception scenarios (rows 15 and 16), and cross references to quality requirements (row 17).

Table 3.1 Template for textual Use-Case documentation (from [120]).

Template for Textual Use-Case Documentation		
N.	Section	Content / Explanation
1	Designation	Unique designation of the Use-Case.
2	Name	Unique name of the Use-Case.
3	Authors	Names of the authors that were involved in this Use-Case description.
4	Priority	Importance of the Use-Case according to the applied prioritization technique.
5	Criticality	Criticality of the Use-Case, e.g., with respect to how much damage a failure of the Use-Case may cause.
6	Source	Designation of the source from which the Use-Case was elicited ([stakeholder document system]).
7	Person responsible	The stakeholder who is responsible for this Use-Case.
8	Description	Brief description of the Use-Case.
9	Trigger event	Name of the event that triggers this Use-Case.
10	Actors	List of all actors that are involved in this Use-Case.
11	Pre-conditions	List with all necessary constraints that must be met before the Use-Case can begin execution.
12	Post-conditions	List of all states the system can be in immediately after the execution of the main scenario.
13	Result	Description of the results that are produced during Use-Case execution.
14	Main scenario	Description of the main scenario of the Use-Case.
15	Alternative scenarios	Description of the alternative scenarios of the Use-Case or list of the trigger events of alternative scenarios. Often, post-conditions different than those described in (12) may hold.
16	Exception scenarios	Description of the exception scenarios of the Use-Case or list of the trigger events of exception scenarios. Often, post-conditions different than those described in (12) may hold.
17	Qualities	Cross references to quality requirements.

3.6.3 The i* Framework

The i* framework is a GORE framework that was originally proposed in [165]. The framework attempts to articulate a notion of *distributed intentionality: actors depends on other actors for goals to be achieved*. This has shift the requirements modeling and analysis towards one that is based on analyzing and describing social relationships among actors [165]. In addition, the framework focuses on providing an answer to the *why-question* behind the *what* and *how* aspect of a project [166]. This framework has been adapted and adopted in many fields including requirements engineering, business redesign, business organization, security, etc. Typically, this framework has become the prime

requirements model for agent-oriented software engineering, though the Tropos methodology [29]. Since November 2008, the i* framework has become part of ITU Standard².

Fundamentally, the i* framework is composed of two models: the *Strategic Dependency (SD)* and the *Strategic Rationale model (SR)* ones [165]. The SD model is used for representing the dependency relationships between the different actors within the organization. The SR model represents the rationales of the different actors concerning the different organizational processes. We review both models in the following sections. We do not however review all of the feature of i* framework but only the basic features that are relevant to this thesis; the interested reader can find more information in [165].

3.6.3.1 The i* Framework Elements

This section provides the syntax and semantic of the i* framework elements used in both the SD and SR models. Figure 3.11 shows the graphical representation of the elements; following Yu et al. [165], the elements are:

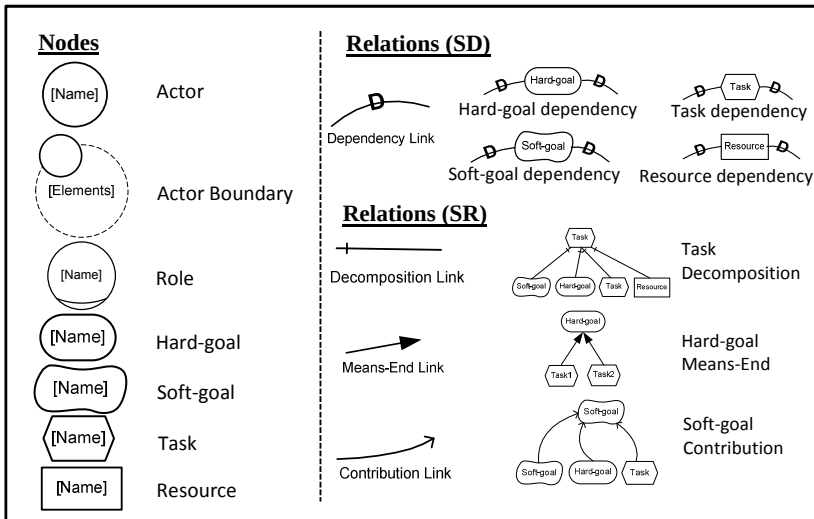


Fig. 3.11 The i* elements.

- *Actor*: “An actor is an active entity that carries out actions to achieve goals by exercising its know-how”;
- *Role*: “A role is an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor”;
- *Hard-goal*: “A hard-goal is a condition or state of affairs in the world that the actors would like to achieve”;

²<http://www.itu.int/rec/T-REC-Z.151/en>

- *Soft-goal*: “A *soft-goal* is a condition or state of affairs in the world that the actor would like to achieve. But unlike a *hard-goal*, there are no clear-cut criteria for whether the condition is achieved, and it is up to the developer to judge whether a particular state of affairs in fact achieves sufficiently the stated of *soft-goal*”;
- *Task*: “A *task* specifies a particular way of attaining a goal”;
- *Resource*: “A *resource* is the finished product of some deliberation-action process”;
- *Dependency Link*: It represents a dependency relation between two actors. There are four types of dependency within the i* framework: *Hard-goal*, *Soft-goal*, *Task*, and *Resource*; the detail of each dependency is described in the following section;
- *Decomposition Link*: A task element is linked to its component nodes by decomposition links. A task can be decomposed into four types of elements (i.e., *subgoal*, *subtask*, *resource*, and/or a *Soft-goal*) which correspond to the four types of elements. The task can be decomposed into one to many of these elements. These elements can also be part of dependency links in SD model(s) when the reasoning goes beyond an actor’s boundary;
- *Means-end Links*: The Means-end links indicate a relationship between an end and a mean for attaining it. The ‘mean’ is expressed in the form of a task, since the notion of task embodies how to do something, while the ‘end’ is expressed as a goal. In the graphical notation, the arrowhead points from the means to the end;
- *Contribution Links*: The Contribution link can be used to link any of the elements to a soft-goal to model the way any of these elements contribute to the satisfaction or fulfillment of the soft-goal.

3.6.3.2 Strategic Dependency Model

The SD model aims to capture the *intentional structure* of a process instead of the usual nonintentional and nonstrategic process models of activities and entities. The SD model is actually a graph representing network of actors with intentional *dependency* relationships among actors; the dependency is known as *strategic dependency*. A dependency (called *dependum*) describes the need of participation or agreement in order to accomplish a particular goal of the *dependor* actor from the *dependee* actor. This model particularly allows identification of stakeholders, the analysis of opportunities and vulnerabilities and the identification of relationship patterns.

Basically, there are four types of dependum: *Hard-goal*, *Soft-goal*, *Task*, and *Resource*. Therefore, Yu et al. [165] highlight that the SD model has four types of dependencies, these are:

- *Hard-goal dependency*: The *dependor* depends on the *dependee* to bring about a certain state in the world. The *dependee* is given the freedom to choose how to do it;

- *Soft-goal dependency*: A *depender* depends on the *dependee* to perform some task that meet a softgoal;
- *Task dependency*: The *depender* depends on the *dependee* to carry out an activity. A task dependency specifies *how* the task is to performed but not *why*;
- *Resource dependency*: One actor (the *depender*) depends on the other (the *dependee*) for the availability of an entity (physical or informational). By establishing the dependency, the depender gains the ability to use this entity as a resource.

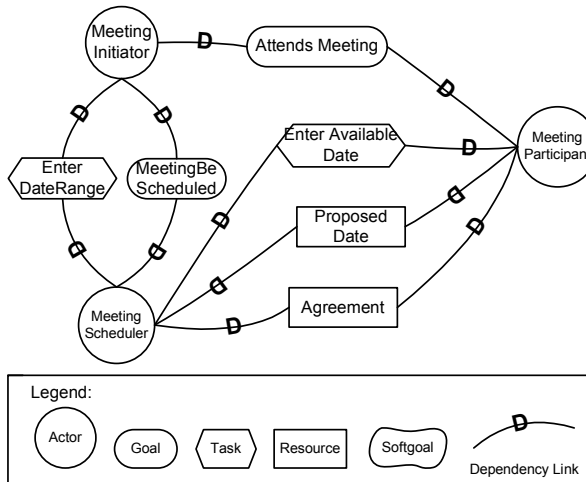


Fig. 3.12 Strategic Dependency model for Meeting Scheduling (adapted from [165]).

The Figure 3.12 presents an example of SD model for *Meeting Scheduling* with the support of *Meeting Scheduler* system. The main functionality of *Meeting Scheduler* is to find the best date in order to maximize the number of participants from ranges of date provided by the *Meeting Initiator* for organizing a meeting. The system systematically communicates with *Meeting Participants* for their availabilities on those dates range and the system chooses the best date for the meeting.

The *Meeting Scheduling*'s SD model consists of two human actors—i.e., the *Meeting Initiator* and the *Meeting Participant*; and one system actor—i.e., the *Meeting Scheduler*. The *Meeting Initiator* depends on the *Meeting Participant* for the goal *Attends Meeting*. The *Meeting Initiator* depends on the *Meeting Scheduler* for the goal *MeetingBe Scheduled*; while the *Meeting Scheduler* depends on the *Meeting Initiator* for the task *Enter DateRange*. Once the dates range are entered, the *Meeting Scheduler* depends on the *Meeting Participant* for the task *Enter Available Date*. Then, the *Meeting Scheduler* merges all availability dates provided by the participants

for the best date with maximum number of participants and communicate that date to participants and wait for their agreements. Therefore, the **Meeting Participant** depends on the **Meeting Scheduler** for the resource **Proposed Date** and the **Meeting Scheduler** depends on the **Meeting Participant** for the resource **Agreement**.

3.6.3.3 Strategic Rationale Model

The SR model aims at capturing *intentional* elements of actors and their rationale behind each dependency relationship. It allows visualizing the intentional elements into the boundary or an actor in order to refine the SD model to add reasoning ability. This model is based on the elements of SD model.

The SR model basically consists of four main elements—i.e., *hard-goal*, *soft-goal*, *task* and *resource*—and three type of links—i.e., *means-end* link, *decomposition* link and *contribution* link. Figure 3.13 exposes an example of SR model for the *Meeting Scheduler* described in the previous section.

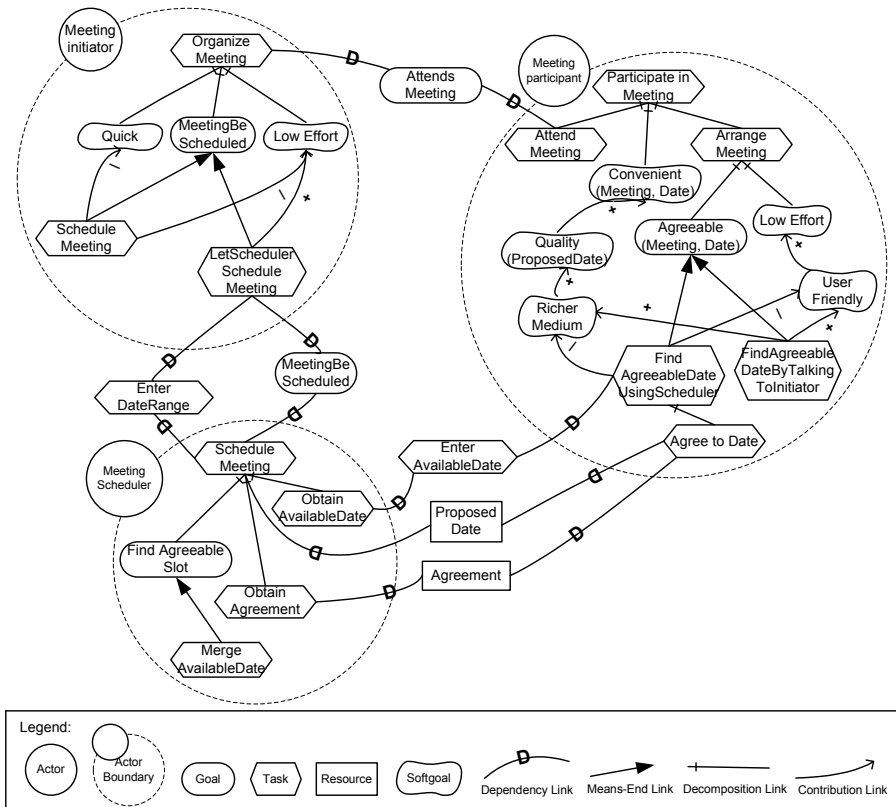


Fig. 3.13 Rationale Dependency model for Meeting Scheduling (adapted from [165]).

Figure 3.13 shows an example of an SR diagram refined from the SD diagram presented in Figure 3.12; we partially describe it here. We can see that the strategic dependencies are linked to the intentional elements. We observe that the *Meeting Scheduler* has a main task **ScheduleMeeting**. This task is decomposed with a *task decomposition link* into the hard-goal **Find Agreeable Slot** and two tasks **Obtain AvailableDate** and **ObtainAgreement**. The hard-goal **Find Agreeable Slot** is the *end* achieved by the *mean* task **MergeAvailableDate**. Likewise, the *Meeting Initiator* has the main task **OrganizeMeeting** which is decomposed into two soft-goal **Quick** and **Low Effort** and a hard-goal **MeetingBeScheduled**. The latter is the *end* for the task **ScheduleMeeting** and **LetSchedulerScheduleMeeting**. The task **ScheduleMeeting** contributes negatively to both soft-goals **Quick** and **Low Effort**. The task **LetSchedulerScheduleMeeting** contributes positively to the soft-goal **Low Effort**.

3.7 Conclusion

This chapter has presented a basic overview on RE. It focuses mainly on RE activities and requirements artifacts such as the use of NL for writing requirements as well as graphical models for modeling requirements. More precisely, it has overviewed two graphical models that are relevant to this thesis, the *i** framework and the Use-Case model.

Since its critical and fundamental importance has gradually been recognized, RE has gained a lot of interest in both literature and software industry during the last decades. *Missing, poorly communicated, inconsistent, incomplete and ambiguous* requirements are major threats for the success of a software project in that it increases the likelihood of developing a wrong system. This could potentially lead to huge losses. Without RE, software developers should not know what to develop, users should not know what to expect and, more importantly, project teams should be unable to verify if the designed system meets the initial business needs.

The complexity of (large) software systems leads to the necessity to use advanced development methodologies allowing to increase the level of understanding of the problem continuously over the *Software Development Life Cycle (SDLC)*. In this, waterfall based SDLC fail to furnish a constant reevaluation of the current software problem understanding as well as the capacity to embrace identified requirements changes. *RE should thus not be considered as a phase achieved once and for all in the SDLC but repeated iteratively along the software development like in agile methods.* This is studied into the next chapter.

Chapter 4

Requirements Engineering in Agile Methods

Requirements Engineering (RE) is an essential discipline for any software development methodology. However, the approach for modeling requirements of each methodology might be different from one to another. The waterfall model addresses RE as a single phase. Requirements are modeled and documented before the other phases of software development are tackled. In agile methods, on the other hand, RE activities are continuous along the development life cycle. RE activities take place just before each iteration starts. In addition, agile methods have proposed *User Stories (US)*, the main requirements artifact for conducting RE activities. Most of RE activities in agile methods that have been proposed adhere to US artifacts. The main focus of this chapter is to study the US within agile methods. Particularly, we concentrate on reviewing different techniques for visualizing US, but also models that are often used by agile practitioners next to US. Furthermore, we briefly overview planning within agile methods.

This chapter is structured as follows. Section 4.1 discusses the relationship between RE activities and agile practices for conducting requirements. Section 4.2 discusses requirements artifact proposed by agile methods—i.e., the US. Section 4.3 reviews different visualizing techniques for US and models used by agile practitioners next to US. Section 4.4 provides a brief overview on planning within agile methods based on US artifacts. Finally, Section 4.5 concludes the chapter.

4.1 Requirements Engineering Activities in Agile Methods

RE and agile methods are often seen as being incompatible since RE is frequently conceived as a phase heavily relying on documentation for sharing knowledge [112]. However, the study in [152] reveals that RE is critical for project success in agile methods. According to Ramesh et al. [123], agile RE practices generally do not follow the RE principles. Nonetheless, agile methods do address all the RE activities (see Section 3.3.1) [13, 95, 123]. Table 4.1 provides the mapping of XP's and Scrum's practices with RE activities.

Table 4.1 Requirements engineering implementation in XP and Scrum (from [95]).

RE activity	XP's practice	Scrum's practice
Requirements Elicitation	• Requirements elicited as stories; • Customers write user stories.	• Product Owner formulates the Product Backlog; • Any stakeholder can participate in the Product Backlog.
Requirements Analysis	• Not a separate phase; • Analyze while developing; • Customer prioritizes the user stories.	• Backlog Refinement Meeting; • Product Owner prioritizes the Product Backlog; • Product Owner analyzes the feasibility of requirements.
Requirements Specification	• User stories & Acceptance tests as requirements documents; • Software products as persistence information; • Face-to-face communication.	• Face-to-face communication.
Requirements Validation	• Test Driven Development (TDD); • Run acceptance tests; • Frequent feedback.	• Review meetings.
Requirements Management	• Short planning iteration; • User stories for tracking; • Refactor as needed.	• Sprint Planning Meeting; • Items in Product Backlog for tracking; • Change requirements are added/deleted to/from Product Backlog.

Agile methods are often applied in an environment where requirements are unstable or unknown and change is the norm. In such situations, it is hard to build high-quality requirements documents as recommended by RE (see Section 3.4.4). These have lead to some agile RE practices. Particularly, XP has proposed US for writing requirements. US have become fundamental in agile methods. We will discuss about the latter in the next section.

4.2 User Stories: the Requirements Artifacts of Agile Methods

This section studies US in detail. Section 4.2.1 provides an overview of US. Section 4.2.2 provides different frameworks for writing quality US. Section 4.2.3 explores the US templates. Section 4.2.4 presents several concepts used

next to US. Section 4.2.5 provides the differences between US and some other requirements artifacts. Finally, Section 4.2.6 provides the advantages and disadvantages of using US in an agile project.

4.2.1 User Story Overview

US are the main requirements artifacts of agile methods. US have initially been proposed by Kent Beck in *eXtreme Programming (XP)* for the value of *simplicity* [16]. It has, since then, been adopted by other agile methods (e.g., Scrum, AgileModeling [7], ...). Beck and Fowler [16] argue that “*the [user story] is the unit of functionality in an XP project. We demonstrate progress by delivering tested, integrated code that implements a [user story]. A [user story] should be understandable to customers and developers, testable, valuable to the customer, and small enough so that the programmers can build half a dozen in an iteration*”. In addition, they describe US as “*a chunk of functionality (some people use the word feature) that is of value to the customer*”. Cohn [38] describes US as “*a short, simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system*”.

Basically, US are written by end-users; this implies the end-user to be part of the software development process like in End-User Development (*EUD*) [88]. It is a text of maximum of two lines written in the everyday or business language (thus natural language) by the end-user of a system. According to Beck and Fowler [16], the best US should describe something important to the end-user, and, the shorter the US the best. Traditionally, US are written down on *index cards* (see Figure 4.1) but nowadays also through the use of specially designed *Computer Aided Software Engineering (CASE)* tools (i.e., software such as Excel, JIRA¹, etc.) allow us writing and managing US [48].

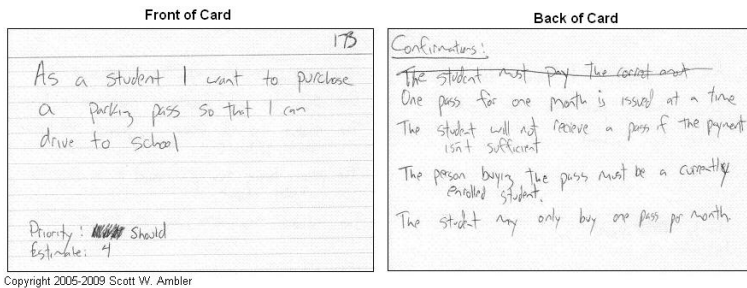


Fig. 4.1 User story index card (from [8]).

4.2.2 Features of High-Quality User Stories

Beck and Fowler have initially provided several principles of how US should be written in [16]. These are: (1) *US must be understandable to the customer*; (2) *Each US must provide something of value to the customer*; (3) *Developers do*

¹<https://www.atlassian.com/software/jira>

not write US; (4) US need to be of a size that several of them can be completed in each iteration; (5) US should be independent of each other, and finally; (6) Each story must be testable. Nevertheless, several explicit models for high-quality US have been developed for improving requirements in agile methods. We describe the INVEST model, the INSERT model and the Quality User Story Framework in the following subsections.

4.2.2.1 INVEST Model

The INVEST model was proposed by Bill Wake [36, 87, 151]. According to Wake, a good US must respect the six attributes of the INVEST model. These attributes are *Independent*, *Negotiable*, *Valuable*, *Estimable*, *Small*, and *Testable*; these attributes constitute the INVEST acronym. Following Cohn [36], Leffingwell [87], Patel and Ramachandran [114], and Wake [151], we describe these attributes as follows:

- *Independent*: This means that each US must not depend on any other US; it means that a US can be prioritized, developed, tested and, potentially, even delivered on its own. Dependency between US leads to prioritization, estimation and planning difficulty;
- *Negotiable*: US should be written to capture the essence of requirements and not their details. The details of US are later co-developed by developers and end-users. By doing this, we can keep US negotiable. According to Leffingwell [87], a US should not serve as a contract; it is rather a placeholder for requirements to be discussed, developed, tested, and accepted. Ideally, US contain one to two phrases that act as a reminder in order to have a conversation and possibly some notes about issues that are to be solved during the conversation with the end-user;
- *Valuable*: US are written in such a way that provides benefits and value to end-users. That is why, in agile methods, it is recommended that end-users write US instead of developers. It allows them to better prioritize the different US within the development schedule;
- *Estimable*: It is important that, for each US, the development team is able to provide an estimation of its complexity, amount of work and time required to transfer it into a working software code. The estimation is based on the team's experience; a good estimation has a positive impact on team's predictability;
- *Small*: US should be small enough so that they can be implemented in one iteration; therefore, they provide value to customer. When US are too big or too broad it becomes hard to estimate, so that they are useless in planning activities. These types of US are known as *Epic* US; they need to be spitted into smaller and estimable US [39]. Conversely, there can also be too small; these are also problematic for the planning. These US should be combined with each other to make a bigger one [39], *Theme* US;

- *Testable*: Each US should be written in such a way that it is possible to test whether or not the desired functionality is successfully implemented and transferred into working code.

4.2.2.2 INSERT Model

As for the INVEST model, Patel and Ramachandran [115] proposed another acronym for representing quality features of US in XP which is known as the INSERT model. INSERT stands for *I*ndependent, *N*egotiable, *S*mall, *E*stimable, *R*epresentation of user functionality, and *T*estable.

The differences between the INSERT and INVEST models are the *Representation of user functionality* and *Value* characteristics. Following Patel and Ramachandran [115], it is difficult to have tools or documents proving that US are valuable to the customer. In addition, the authors claim that their model improves the requirements elicitation process of US in XP and results in writing higher quality US.

4.2.2.3 Quality User Story Framework

Recently, Lucassen et al. [93] have proposed a framework for writing quality US. Figure 4.2 exposes this Quality User Story Framework. The qualities of US are defined at three levels: *Syntactic*, *Semantic*, and *Pragmatic*. Following Lucassen et al. [93], the syntactic quality level is concerned with “*the textual structure of a user story without considering its meaning*”. The semantic quality level is concerned with “*the relations and meaning of (parts of) the US text*”. Finally, the pragmatic quality level is concerned with “*choosing the most effective alternatives for communicating a given set of requirements*”.

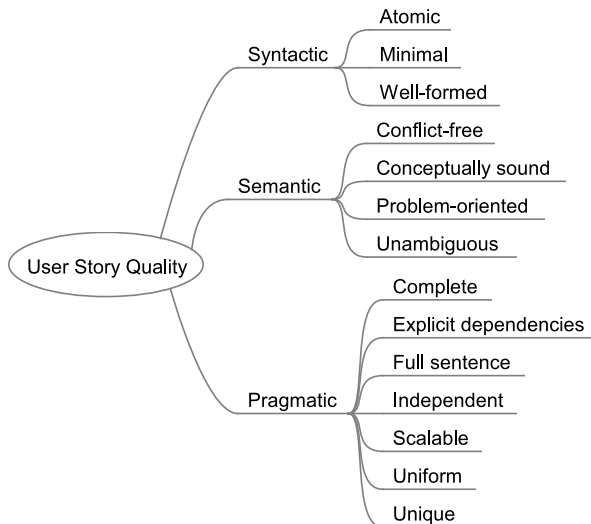


Fig. 4.2 Quality user story framework (from [93]).

4.2.3 User Story Templates

Kent Beck, the main artisan of US states that “the best user story is a sentence or two that describes something important to customer”. He adds that US should be written in “plain English” [16]. He also provided an example of a US—e.g., “The system should check the spelling of all words entered in the comments field.” This US structure is very similar to the IEEE-830 style (see Section 4.2.5). In [36], the way of writing US has been changed; Cohn writes it as “A user can post her resume”. We can observe that US are written from the perspective of the customer rather than of the system. In addition, he also suggests to include ‘user role’ into the US. Therefore, instead of writing a US as “A user can post her resume” this US can be written as “A Job Seeker can post her resume”. By respecting this way of writing, developers have the feeling of fulfilling real customer needs.

Later, Cohn took lesson from the work performed at Connextra and widespread the template used by that company. Figure 4.3 exposes the US that was written at Connextra [ref²]. According to Cohn [36], the template is structured as follows: *I as a <role> want <function> so that <business value>*. Cohn has proposed another template in [37], namely *As a <type of user> I want <capability> so that <business value>*. He has proposed many templates structured similarly but the key words of each segment are changed. His latest template is structured as follows: *As a <type of user>, I want <some goal> so that <some reason>* [ref³]. Following Cohn, the ‘so that’ clause is optional.

Connextra		A Connextra Story Card	
Perspective	Title	Reserved for priority	
	WRITING GOOD STORIES		
Reason	Requirements		
As a Connextra employee - I want to know how to write good stories so that I can submit cards to the planning game that are clear and will be accepted in the next iteration.			
Author	Date	Reserved for estimate	
Tim	8/Nov/01		

Fig. 4.3 Connextra user story card.

Other agile practitioners have suggested and published their own template. For example, Jeff Patton has proposed a US template structured as follows: *As a <type of user> I want to <perform some task> so that I can <achieve some goal>* [ref⁴]. Nonetheless, those templates are structured like those of Cohn. Generally, the US template is structured as follows: *As [WHO], I want [WHAT] so that [WHY]*. A collection of this kind of template is presented in Appendix B. Significantly, there are two templates which are structured differently. The first

²http://agilecoach.typepad.com/photos/connextra_user_story_2001/connextrastorycard.html
³<https://www.mountaingoatsoftware.com/agile/user-stories>
⁴<http://jpattonassociates.com/>

one was proposed by Chris Matts in 2011 [ref⁵]. The template is structured as follows: *In order to <receive benefit> as a <role>, I want <goal/desire>* (see [27]). This template allows us to emphasize on the ‘value’. The second one, is the 5Ws template and it is structured as follows: *As [WHO] [WHEN] [WHERE], I [WHAT] because [WHY]* [ref⁶].

According to our observation on the different ways for writing US, we would classify US templates into five categories. Table 4.2 provides the five templates with descriptions and examples.

Table 4.2 The five templates for writing user stories.

Template	Description and Example
Kent’s Template	It refers to the way of writing US proposed by Kent Beck in [16]. US are written in prose. <i>E.g., The system should check the spelling of all words entered in the comments field.</i>
Cohn’s Template 1	It refers to the way of writing US proposed by Cohn in [36]. US are written in prose but explicitly include the ‘user role’. <i>E.g., A Job Seeker can post her resume.</i>
Connextra’s Template or Cohn’s Template 2	It refers to the template proposed by Connextra and later made publicly available by Cohn. The US consists of three parts (i.e., [WHO], [WHAT] and [WHY]) and in general it is structured as follows: <i>As [WHO], I want [WHAT], so that [WHY]</i> . <i>E.g., As a creator, I want to upload a video so that any users can view it.</i>
Chris’s Template	It refers to the template proposed by Chirs in his blog. Similarly to Connextra’s template, this template also has the same three parts but they are structured differently. The template is structured as follows: <i>In order to <receive benefit> as a <role>, I want <goal/desire></i> . <i>E.g., In order to increase the number of sales of our print consumables, as a marketing manager, I want customers to register their e-mail addresses.</i>
5Ws Template	US is composed of five parts—i.e., [WHO], [WHAT], [WHERE], [WHEN], and [WHY]. The US can be structured as follows: <i>As [WHO] [WHEN] [WHERE], I [WHAT] because [WHY]</i> or <i>[WHO][WHAT][WHEN][WHERE][WHY]</i> [ref ⁷]. <i>E.g., WHO: As an Instructor</i> <i>WHAT: Selects multiple students (at once) and adds them to a discussion group</i> <i>WHEN: After determining what students go in which groups</i>

⁵http://antonymarcano.com/blog/2011/03/fi_stories/

⁶<http://www.ambitiousmanager.com/user-stories-explained/>

⁷<http://blog.agilejedi.com/2008/03/writing-user-stories-5-ws-way-writing.html>

	WHERE <i>From the student listing screen</i> WHY: <i>To get students into discussion groups</i>
--	--

Table 4.3 provides the coverage of each US template regarding the three dimensions of RE (see Section 3.2.1). The complexity of writing US using each template is also provided.

Table 4.3 User story coverage and complexity.

US Template	RE Dimension Coverage			Complexity
	WHO	WHAT	WHY	
Kent’s Template		✓		Easy
Cohn’s Template 1	✓	✓		Easy
Cohn’s Template 2	✓	✓	✓	Medium
Chris’ Template	✓	✓	✓	Medium
5Ws Template	✓	✓	✓	Difficult

4.2.4 User Story, Epic, and Theme

Within agile methods driven by US, the *Epic* and *Theme* concepts are often used to facilitate the planning, prioritizing and organizing of US [36, 128].

Some US are large in size and cannot be implemented in one iteration. Cohn [39] defines such US as Epics. Following Cohn [39], there is no clear defined size of US to be called Epic or not. An Epic US is simply referred to a US that cannot be implemented in one iteration. The Epic US needs to be decomposed into sets of smaller US. In some cases, some Epics are too large and they can be subdivided into sub Epics [39].

According to Cohn [39], Theme is a collection of logically related US. Unlike an Epic US that is not decomposed into enough detail yet, a Theme is a collection of multiple pieces of smaller US.

4.2.5 User Story versus others User Requirements Artifacts

There are several requirements artifacts in the RE literature. Some are seen as overlapping with US and some can be seen as complementary. This section aims at comparing US with IEEE-830 documents, Use-Cases, Scenarios, Personas, and finally, User-Task Models. According to Alexander and Maiden [6]; and Wiegers and Beatty [161] there are some confusions between US with IEEE-830 documents, Use-Cases, and Scenarios. However, US, Personas, and User-Task Models are commonly used within User-Centered Design [136]. Sections 4.2.5.1, 4.2.5.2, 4.2.5.3, 4.2.5.4, and 4.2.5.5 provide a brief description of each aforementioned models while Section 4.2.5.6 provides the comparison of US with all of these models.

4.2.5.1 IEEE-830 documents

The IEEE-830 is a template standardized by the *Institute of Electrical and Electronics Engineers (IEEE)* in 1998 for documenting *software requirements*

specification [70]. As briefly depicted in Section 3.5.2, it provides global rules for structuring requirements documents and local rules for specifying requirements. The latter recommends to write requirements following the template “*The system shall ...*”; Figure 4.4 provides examples of requirements written following IEEE-830 style.

1. The system **shall** allow a company to pay for a job posting with a credit card.
 - 1.1. The system **shall** accept Visa, MasterCard and American Express cards.
 - 1.2. The system **shall** charge the credit card before the job posting is placed on the site.
 - 1.3. The system **shall** give the user a unique confirmation number.

Fig. 4.4 Requirements in IEEE-830 style (from [36]).

According to Cohn [36], a US and an IEEE-830 statement are different in nature. Indeed, the IEEE-830 statement focuses on writing a software attribute whereas a US focuses on describing a user goal. By focusing on user goals for new software rather than on a list of attributes of new software, we are able to design a solution that better fulfills the user needs [36].

4.2.5.2 Use-Cases

As previously described in Section 3.6.2, the Use-Case model aims at capturing the interaction between actors and the system. According to Cohn [36], a Use-Case is “*a generalized description of a set of interactions between the system and one or more actors, where an actor is either a user or a system*”. Use-Cases can be either modelled in a diagram, the *Use-Case diagram*, or written as text in a template, the *Use-Case specification* (see Section 3.6.2). Within the latter, there are two sections for writing the *main* and *alternative scenario* of a Use-Case; these scenarios are referred to as *Use-Case scenario* and they are different from the scenario in *Human-Computer Interaction Design* (see next section).

According to Cohn [36], and Alexander and Maiden [6], the main difference between US and Use-Case is their scope. The scope of a US is smaller than the one of a Use-Case. A US is intentionally kept small so that it can be executed and developed in at most an iteration. However, a US can be compared or aligned with a Use-Case scenario [36].

Contrarily, customized Use-Cases like *essential Use-Cases* [40] are comparable to US. An essential Use-Case is “*a use case that has been stripped of hidden assumptions about technology and implementation of detail*” [36]. According to Cohn [36], the *user intentions* of the essential Use-Case could be directly interpreted as a US. Similarly, Wiegers and Beatty [161] argue that a US sometimes covers the same scope as an entire Use-Case but, in other cases, a US represents just a single Use-Case scenario. We share the same view of the latter. This subject is also studied in Chapter 8.

4.2.5.3 Scenario

We refer to the concept of scenario which is defined in *Human-Computer Interaction Design* (see [28] for details). According to Cohn [36], a scenario is “a detailed description of a user’s interaction with a computer”. Following Carroll [28], scenarios include the following characteristic: a *setting*, *actors*, *goals* or *objectives*, as well as *actions* and *events*. Figure 4.5 depicts an example of a scenario.

Maria is thinking about making a career change. Since the glory days of the dot-com boom she has worked as a tester at BigTechCo. A former high school math teacher, Maria decides she’ll be happier if she returns to teaching. Maria goes to the BigMoneyJobs.com website. She creates a new account with a user name and password. She then creates her resume. She wants to find a job as a math teacher anywhere in Idaho but preferably near her current job in Coeur d’Alene. Maria finds a handful of jobs that match her search criteria. The job that intrigues her most is with the NorthShore School, a private high school in Boise. Maria has a friend, Jessica, in Boise whom she hopes may know someone at NorthShore. Maria enters Jessica’s email address and forwards the job link to her with a note asking if she knows anyone at the school. The next morning Maria gets an email from Jessica saying that she doesn’t know anyone at the school, but she knows of the North Shore School and it has a wonderful reputation. Maria clicks on a button that submits her resume to North Shore.

Fig. 4.5 Scenario in Human-Computer Interaction Design (from [36]).

According to Cohn [36], the main difference between a US and a scenario lies in their scopes and details. Scenarios are more extensive and detailed than US. In fact, scenarios could be seen as large Epics containing a lot of possible US. In addition, a US is generally written in a structured sentence while a scenario is written in a free style.

4.2.5.4 Personas

Persona is a *User-Centered Design* technique proposed by Alan Cooper [41]. Personas are not users, nor roles and nor actors of the system [ref⁸], they are neither the real people. Personas are rather hypothetical archetypes of actual users; they represent real people throughout the design process and they are described as if they were real people—i.e., they are defined with significant rigor and precision [41]. Unlike user, role and actor, a persona is described by a name, a photo, a long narrative text describing its personalities, its needs, its goals, etc. in a single page. There are many ways for writing personas, some templates use US in personas for describing personas’ needs and vice-versa—i.e., use persona in the WHO dimension of US [66]. The latter refers to the persona stories which are not in the scope of this thesis. Figure 4.6 provides an example of a persona using US (but they do not follow any US templates) for describing their needs.

⁸<http://www.agilemodeling.com/artifacts/personas.htm>

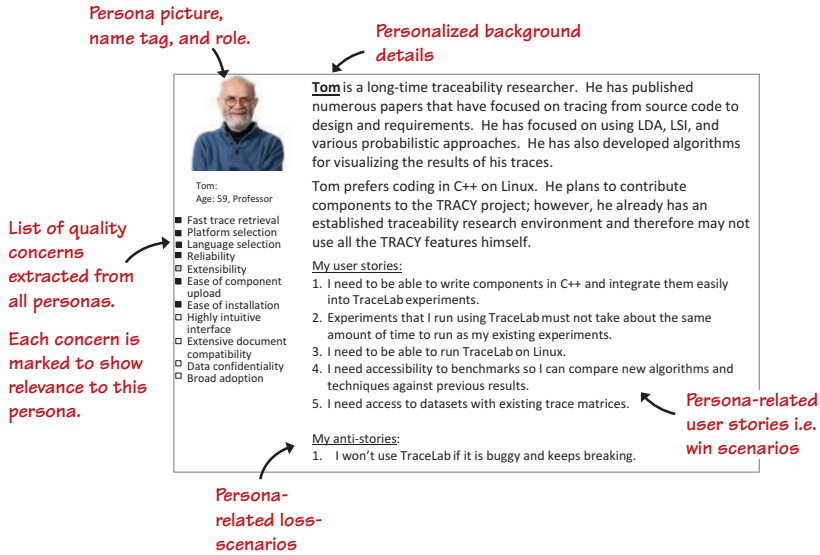


Fig. 4.6 An example of a Persona (from [33]).

Personas and US are different in nature. Personas are used to model real users of a system; whereas, US are used to express user requirements or needs. As mentioned before, US can be used to describe personas' requirements. Cohn [36] suggests to use personas for writing US; they allow to discover new requirements.

4.2.5.5 User-Task Models

User-Task Models, or task models, are commonly used to model the interaction between users and systems in *User-Centered Design* [89]. The task models are often used for describing the tasks that users can perform with a system. A task represents a specific action that a user can undertake in order to reach a goal; a goal “*is either a desired modification of state or inquiry to obtain information on the current state*” [102]. Many task models have been proposed in the literature [47]. Each task model addresses different aspects of user interface design, and has different formalities, expressivenesses and complexities [89]. In this thesis we rather compare US to the *Hierarchical Task Analysis (HTA)* [9]. Indeed, according to Limbourg and Vanderdonckt [89], the HTA task model has a lower complexity than other task models. Figure 4.7 exposes an example of an HTA task model.

The main difference between task models and US is their nature. US are rather requirements artifacts; whereas, task models are analysis artifacts. This means that they are built from others artifacts such as scenarios, US, etc. [136]. Another difference is the models themselves—i.e., US are text-based while task models are generally graphical hierarchical trees. However, US can be used as inputs for constructing task models [136]. A US can be transformed into a task of task models.

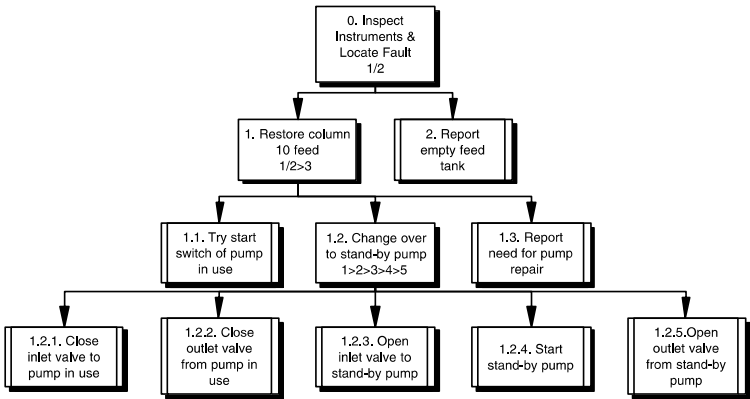


Fig. 4.7 An example of Hierarchical Task Analysis (from [9]).

4.2.5.6 Comparison between User Story and other User Requirements Artifacts

Despite the differences discussed in the previous sections, we further compare US with the aforementioned models regarding to five dimensions in order to justify our choice of studying in US for this thesis. The five dimensions are *Length*, *Scope*, *Lifespan*, *Expressiveness*, and *Degree of agility*. The length refers to the length of the text used for naming elements of the models. The scope refers to the scope of the model—i.e., business value, user requirements, software specification, etc. The lifespan refers to the usage of the models along the software development cycle. The expressiveness refers to the ability of representing or describing any information that users want to express within the requirements by using these models. Finally, the degree of agility refers to the ease and flexibility in changing the models. We give more priority to the latter.

We use US as the baseline for the comparison; we use this symbol “+/-” for the neutral. The symbols “-”, “- -”, “+”, and “++” for representing *lower*, *lowest*, *higher* and *highest* respectively. The given scores are based on our won interpretation.

Table 4.4 The comparison between user story and other user requirements artifacts

	Length	Scope	Level of Detail	Lifespan	Expressiveness	Degree of Agility
US	+/-	+/-	+/-	+/-	+/-	+/-
IEEE-830	+/-	-	+/-	-	+/-	-
Scenario	+	+	++	-	+	-
Use-Cases	-	++	++	+	+/-	- -
Persona	++	+	+	-	+	-
User Task	-	++	++	++	++	- -

Based on the results in Table 4.4, we can see that the scope, level of detail and expressiveness of US are lower than other models. The length of US, on the other hand, has a better score compared to some models. Importantly, the degree of agility of US is higher comparing to other models. We argue that by using graphical models for modeling US or using graphical models complementarity to a written US set allows to reduce some drawbacks in level of detail and expressiveness. These subjects will be addressed in Chapters 6 and 7.

4.2.6 Pros and Cons of using User Stories

This section provides the advantages and disadvantages of using US in software development. Section 4.2.6.1 exposes the advantages of US while Section 4.2.6.2 provides the drawbacks of using US in software development.

4.2.6.1 Benefits of using User Stories

The first advantage of US is the emphasizes on *verbal communication* for gathering requirements [36]. The goal of US consists in writing down desired pieces of functionality in only a few sentences. Consequently, the written US do not contain the required level of detail. In order to be able to develop the different US, the development team has to step into conversation with the customer in order to gather the remaining details that are missing on the story cards. This shifts from writing document in ‘traditional’ methods to verbal communication. Therefore, it allows and promotes rapid feedback cycles that result in a better understanding of the required functionalities of the system.

Since requirements are written by the customer rather than developers in daily language (i.e., natural language), a second advantage is the *comprehensibility of the US*. The absence of an abundant use of both technical and business jargon results in a better understanding of the different US for both developers and users/customers. Furthermore, Cohn [36] claims that US are *the right size for planning activities*. Another advantage of US is that they *encourage the team to defer detail* [36]. This can be beneficial for the project because a lot of high-level US can be written at the start of the project, what might give the customer and users the opportunity to get a clearer look and feel with the system to be developed. The remaining required level of detail can then be gathered subsequently. US also *stimulate the accumulation of tacit (i.e., hidden and intangible) knowledge across the team*. The more conversations that are held between developers and the on-site customer, the more knowledge that builds up within the development team.

US also contain the major advantage of *being compatible with iterative development*, what makes them ideal for usage in agile methods. This benefit is, among other things, reflected in the fact that not all US have to be written at the start of the project. Furthermore, the use of US allows to start with an Epic US that is split into smaller US later on the project. Since US heavily focus on conversation and contain the ability to be written and rewritten in various levels of detail at any moment during the project, they *support opportunistic and agile development*. By using US as requirements artifacts, project teams get

rid of the *I'll know it when I see it (IKIWISI)* syndrome [19] where users have difficulties in providing a clear view on what they want and expect from the system to be developed. Additionally, the US approach contains the possibility to embrace changing requirements during the project cycles [106]. Since US are used within agile methods, a US driven process *encourages participatory design* where one or more customer representatives become part of the development team. Several studies have shown that user involvement is an important success factor in software development [51, 148].

4.2.6.2 Drawbacks of User Stories

Opposed to the benefits of using US as requirements artifacts in agile methods, there are some critical drawbacks related to their usage [36]. First of all, *using US in large projects* can become difficult and complex. In such projects it can become hard to maintain a clear view on the relationships between the different US. Secondly, when *traceability is required, some additional documentation might be required*. Latter necessity in situations where traceability is required conflicts with the values and principles of the agile manifesto that encourages the use of a minimal amount of documentation within the project. Furthermore, US are not adequate artifacts to use with an organization in which management style and culture mandates more formal documentation [6]. Another set of drawbacks are *the ones that also correspond to those of natural language requirements* in general. Natural language (and thus also US) can be interpreted ambiguously [6, 59] what makes the project prone for communication errors. However, using US templates like that ones exposed previously could overcome the latter drawback (see Section 3.5.2). Following Pichler [ref⁹], US are “*not able to express relationships between different features and to describe workflows*”. Last but not least, US driven projects depend on the availability of the customer [106]; in some projects it is hard to always have the customer on ones side.

4.3 Visualizing and Modeling Requirements with User Stories

Traditionally, US are written on physical index cards (or sticker notes) and stored on a whiteboard. Nonetheless, with the CASE-Tool they are rather stored in the form of a list. As evoked requirements analysis in agile methods are based on these artifacts. A US is moved around the US set during the analysis. The understandability of the requirements depends on how the US set are physically distributed on the whiteboard or in the list. On the other hand, from the perspective of requirements engineering, modeling requirements visually plays an important role for better understanding requirements [161]; however, agile methods consider formal models as harmful to agile principles. In practice, we observe that some agile practitioners are using visual models to improve the analysis of US [152]. Therefore, this section aims at reviewing different techniques for visualizing US sets and more formal models used by agile practitioners for requirements analysis within agile methods. This section

⁹<http://www.romanpichler.com/blog/user-story-modelling/>

is structured as follows. Section 4.3.1 provides the overview on User Role Modeling. Section 4.3.2 presents the conventional representation of US in agile methods—i.e., the storyboard, product backlog, etc. Section 4.3.3 overviews the User Story Mapping technique. Finally, Section 4.3.4 exposes some modeling techniques for complementing US.

4.3.1 User Role Modeling

User Role Modeling is a simple practice suggested by Cohn [36]; it is aimed to be conducted prior to the US writing workshop. It is important to identify different user roles before writing US. According to Cohn, writing US from the perspective of a single user type can be harmful—i.e., the US of a specific user role are missing. Modeling different roles and writing US from the different perspectives of these roles can prevent the latter. Cohn defines user role as “*a collection of defining attributes that characterize a population of users and their intended interactions with the system*”. Modeling user role follows these following steps: (1) brainstorm an initial set of user roles; (2) organize the initial set; (3) consolidate roles; (4) refine the roles (for details see [36]). Additional techniques such as *Persona* and *Extreme Characters* can be also used in addition to the user role. Following Cohn [36], by using these techniques new US can be discovered.

4.3.2 The Product Backlog

The common and conventional practice for visualizing US with agile methods is to write US physical index cards as sticker notes and attach them on a wall or a whiteboard and make them visible to everyone in the team. Figure 4.8 shows the Scrum board (sometimes called storyboard) for visualizing US in Scrum. This board can also be served as a monitoring tool to see the progress of the US. The column on the very left of the board is used for storing the US to be implemented; it is known as the Product Backlog column. The column on the very right of the board is used for storing finished US; it is known as the *Done* column. In between the two columns, there are the *To Do* and *In Progress* columns; these constitute the basic elements and it is possible to have more columns depending on how detailed the team wants to monitor the progress. The US are moved from the left to the right on the board to indicate their progress. The order of implementing US depends on the priorities set by the customer. When there are new US, new sticker notes are added to the backlog; the team follows the same process until the backlog is empty. This could also be done by using list tools such as Excel [48].



Fig. 4.8 User Stories on Scrum Board.

This approach works well when the number of US is small; the customer always has the ability to provide priorities correctly to the US set. When the number of US is important, it becomes difficult for the customer to visualize the US set and often the customer (but also the developer) loses its vision on the backlog [117]. The prioritization of US set has turned to be hard and resulted in incorrect prioritization and it rapidly turns to chaos. Cohn [39] suggests to keep the number of US lower than 150 in the backlog to ensure they are kept manageable. To achieve this, Cohn proposes to keep some US in large size—i.e., Epic US, and sometimes we need to group some small US into Themes [39]. Only the US for next iteration have to be decomposed into an implementable size—i.e., in one iteration. The organization of the US is as follows. The small and high priority US are located at the top of the backlog and the large and low priority US are located at the bottom of the backlog; Cohn calls this the *product backlog iceberg*. This is also known as a *Prioritized Product Backlog*. Figure 4.9 exposes the structure of the prioritized backlog.

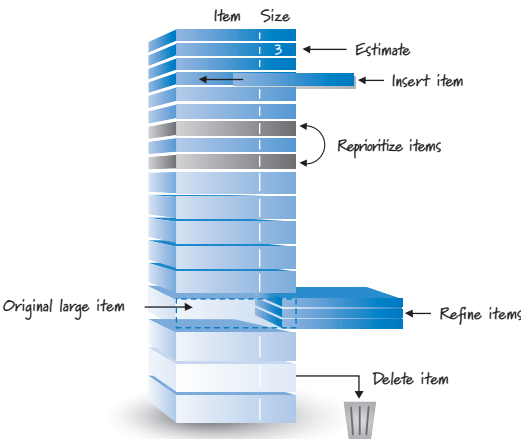


Fig. 4.9 Prioritized product backlog (from [128]).

In order to keep the backlog prioritised and structured, it requires additional efforts and practices for agile teams. Cohn [39] has proposed the *product backlog grooming* practice for maintaining the backlog. This practice is also known as *product backlog refinement*. The goals of product backlog grooming are (i) to ensure that the backlog remains populated with items that are relevant, detailed and estimated to a degree appropriate with their priority, and (ii) maintain the current understanding of the project or product and its objectives. Product backlog grooming is normally conducted before each iteration; it is also part of the planning game. Following AgileAlliance [ref¹⁰], the product backlog grooming consists of:

- removing US that no longer appear relevant;
- creating new US in response to newly discovered needs;
- re-assessing the relative priority of US;
- assigning estimates to US which have yet to received one;
- correcting estimates in light of newly discovered information;
- splitting US which are high priority but too coarse grained to fit in an upcoming iteration.

4.3.3 User Story Mapping

User Story Mapping (USM) is a *user-centric* approach for visualizing and modeling US. It has been proposed by Jeff Patton [118]. USM consists of organizing US along two axes. The horizontal axis is used for arranging US with respect to the order of involvement of the user with the system. In other words, it refers to the process of the user's involvement with the system. The vertical axis is used for arranging US with respect to their priorities and abstractions. The US located at higher position have higher priority than the ones at the lower position. In addition, the vertical axis is divided into three layers; each layer refers to a level of the granularity of US. A US at the higher layer covers the scope of those under its hierarchical structure at the lower layer. The three layers are: the *Backbone*, it refers to the *User Activities*; the *Walking Skeleton*, it refers to the *User Task*; and finally, the lowest granularity is the US. Figure 4.10 exposes the template of the USM.

Following Patton [118] and Rogalsky [ref¹¹], using USM for representing product backlog provides several advantages over the normal and prioritized product backlog. The advantages are as follows:

- It provides the 'BIG PICTURE' of the product backlog; therefore, a better understanding of the system to be developed;
- It provides a better tool for making decisions about refining and prioritizing the product backlog;

¹⁰<https://www.agilealliance.org/glossary/backlog-grooming/>

¹¹<http://winnipegagilist.blogspot.be/2012/03/how-to-create-user-story-map.html>

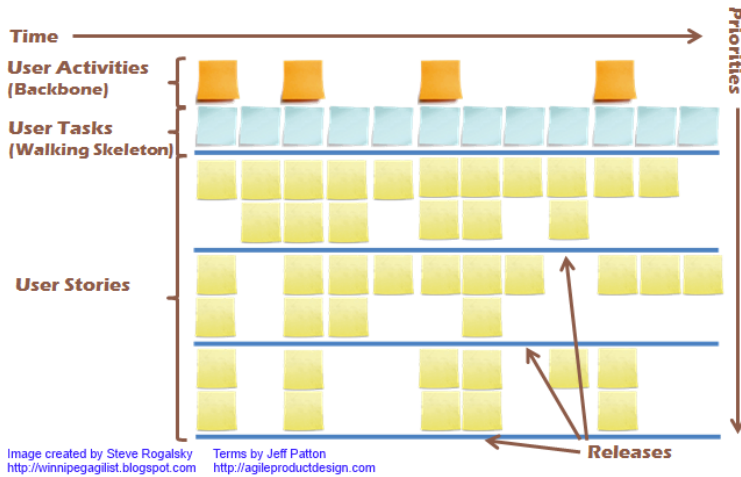


Fig. 4.10 User Story Mapping template.

- It promotes silent brainstorming and a collaborative approach to generating US;
- It provides a visual alternative to traditional project plans;
- It provides a useful model for discussing and managing scope;
- It provides a visual dimensional planning and real options for the project.

4.3.4 Models and User Stories

The shift from well documented requirements to the simple use of US has resulted in neglecting requirements modeling activities within agile methods [137]. This can be seen by examining the values and principles of the agile manifesto—i.e., there is no principle nor value of the agile manifesto focusing on modeling requirements. Conversely, from the perspective of RE, modeling is important; it allows developers to better and easier understand the problems [7, 13], and help the customer in prioritizing requirements [7, 137]—for example, we can identify the interrelated requirements that need to be implemented in the same iteration. Following Ambler [7], modeling activities provide positive impacts in discovering flows in the system; but it is a time consuming activity which is harmful to agile principles. In addition, modeling can slow down the development process.

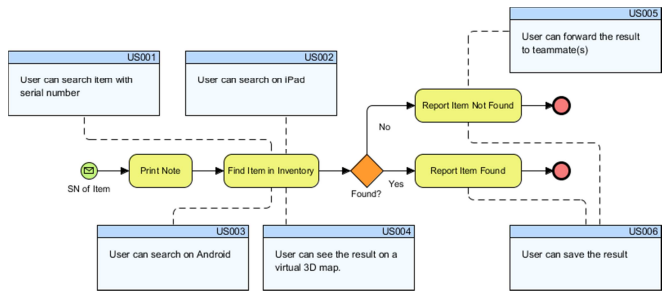
To the best of our knowledge, Ambler is the first one who has tried to include modeling activities in agile methods. He has proposed an agile method called *Agile Modeling* [7] in 2002 which is compatible with other agile methods. It provides practices and guidelines for other agile methods to elaborate the modeling activities without violating the agile principle. He proposes to model just enough to explore the system but not too much. The recent research conducted by Wang et al. [152] revealed that modeling is the second used method

for requirements elicitation after US within agile methods. “*Requirements Analysis preferred to use model as models can show their ideas easily and quickly*” [152].

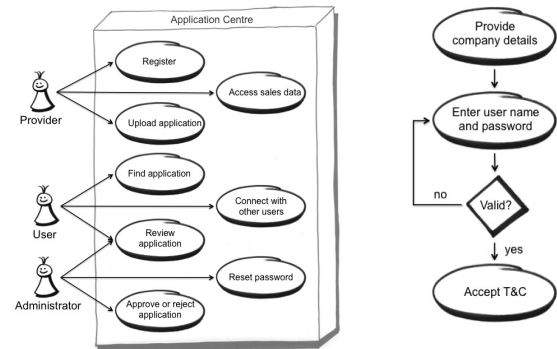
Following Wang et al. [152], Pichler [ref¹²] and Ambler [7], most of models used for requirements activities in agile methods are rather serving as a complementary view to US with the customer (in order to improve the US writing) than modeling US themselves. In other words, models are used for validating requirements or stimulate the latter to write missing US. This means that US are normally generated from the model but not inversely. In this thesis, *we attempt to produce models from a US set and the models themselves can also serve for validating requirements or discovering missing US*. This model can also be served as an input for the planning game.

Following Wang et al. [152], the possible modeling techniques used in requirements analysis within agile method are the *business process model*, the *goal model*, the *Use-Case model*, the *Role Card*, and the *organization model*. We also did an informal research. We used Google as a tool for searching. We used these two keywords ‘*modeling and agile methods*’ and ‘*modeling user stories*’ for doing the search; we only take into account the first one hundred pages for each keyword. Moreover, we only consider blogs and personal websites rather than forums. Our results informally confirm the research of Wang et al. Figure 4.11 provides some models used for/with US. For example, Pichler suggests to use a context diagram for depicting user roles and Epic US, large and coarse-grained US. He argues that the diagram can provide an overview of the product’s functionalities. In addition, he also suggests to use activity diagrams to capture sequences and workflows by connecting individual US. Furthermore, a modeling tool such as Visual Paradigm [111] also provides the possibility to describe US within a Use-Case and a business process diagram.

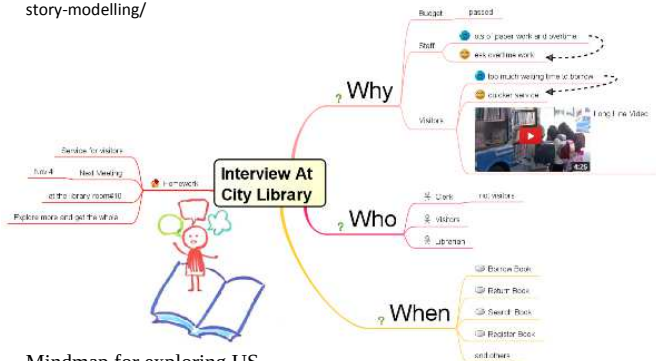
¹²<http://www.romanpichler.com/blog/user-story-modelling/>



Business Process and User Story. source : <https://www.visual-aradigm.com/tutorials/business-process-to-user-stories-mapping.jsp>



Context diagram for modeling user role and epic US and activity diagram for capturing sequence US. source : <http://www.romanpichler.com/blog/user-story-modelling/>



Mindmap for exploring US. source : <https://www.infoq.com/articles/kenji-modeling-agile>

Fig. 4.11 Possible modeling techniques used as a complementing view to user stories.

4.4 User Story Based Planning in Agile Methods

This section aims at presenting the basic US planning techniques found in agile methods. Section 4.4.1 exposes the two strategies for doing iterative planning.

Section 4.4.2 provides some practices for selecting US to be implemented in an iteration.

4.4.1 Iterative Planning with User Stories

Basically, there two types of planning withing the agile approach: *release planning* and *iteration planning*. Iteration planning consists of doing a plan for a selected US set to be developed during an iteration. Whereas, the release planning consists of determining how many US can be accomplished by what date [37]. This can be done by two strategies. In the first case, we set a date and we determine how many US can be accomplished by that date. In the second case, we can set the number of US and determine when these US can be accomplished. Most agile methods adopt the first when doing release planning.

Figure 4.12 provides a general planning process within agile methods. The process involves the following steps [37]:

- *Determine the conditions of satisfaction*: It consists of determining the criteria of success or failure (business value, the amount of money saved, ...) of each iteration;
- *Estimate the US*: It consists of estimating the size of the US in terms of efforts, duration of implementation, user story point, price, ...;
- *Select an iteration length*: It consists of selecting the length of the iteration. For example, the iteration length in Scrum is 2 to 4 weeks;
- *Estimate velocity*: The velocity refers to the amount of user story points (see in [37]) that a team is able to implement within an iteration. Thus, estimate velocity consists in determining the amount of user story points to implement based on the performance of last iteration;
- *Prioritize US*: It consists of prioritizing each US. Normally, the priorities are given by the customer;
- *Select US and release date*: It consists of selecting a set of US to be implemented for the next iteration and defining the release date.

Following Cohn [37], a release plan is important for a number of reasons. First, it helps the product owner and the whole team to decide how many US to be developed and how long will take to deliver a workable software. The sooner the software can be released the better; the organization can start earning a return on its investments. Second, a release plan conveys expectations about what is likely to be developed and in what timeframe. Many organizations need this information because it feeds other strategic planning activities. Third, a release plan serves as a monitoring tool to see the progress of the team.

4.4.2 Selecting User Stories for an Iteration

Selecting a US set to be implemented for the next iteration is somehow changeling; it can significantly impact when and with what functional and

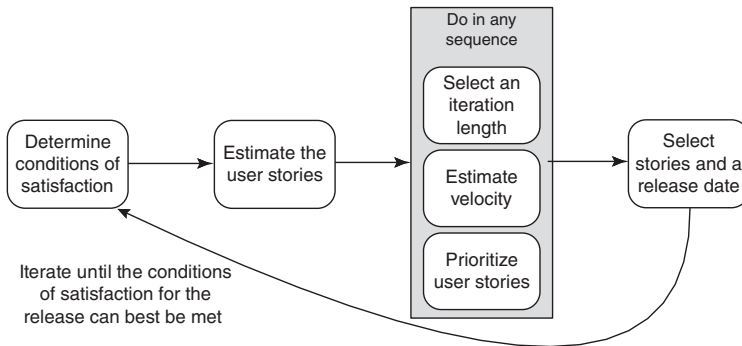


Fig. 4.12 General process for iterative planning in agile methods (from [37]).

nonfunctional quality levels software is delivered. The common strategy used for selecting US for an iteration is to select the set of US located at the top of the product backlog. The main idea is to implement the US with highest priority early in the project so that the customers can realize the most business value [123]. The highest priority means that those US are important for delivering business value.

The above strategy works well when all US in the product backlog are independent from each other—i.e., every US can be implemented on its own and can deliver business value. In some cases, the functionality of some implemented US cannot provide its value to the customer because they depend on other functionalities of other low priority US to make it fully valuable for the customer [27]. This leads to reprioritizing of US so that relevant US can be selected for the next iteration. This can delay the value delivery to the customer. To optimize such a situation, it is better to select a coherent set of US to be implemented in the same iteration. This means that it is better to select some high priority US and some low priority US that are functionally depending on each other to be implemented in the same iteration. Following Rubin [128], there are always dependency between US. Cardinal [27] suggests to use the visual aid like USM for selecting coherent US set. Nonetheless, from the requirements engineering perspective, modeling US can discover dependencies between US [161].

4.5 Conclusion

This chapter has presented the state of the art of RE activities in agile methods. It more specifically focuses on US—the most used requirements artifacts in agile methods.

RE and agile methods are often seen as being incompatible because the former induces producing a formal documentation through a model of the system-to-be and the latter specifically points to producing as few documentation as possible. The RE process of agile methods is continuous over the project life cycle so that it may lead to many changes and putting into question the already modeled requirements. At the inception of the project some RE activities are

conducted in order to get an understanding of the boundaries of the system as well as an evaluation of its critical features. Then, in later iterations, more details about the requirements are collected to build a finer representation of the system-to-be. Last but not least, since simplicity is one of the core agile principles, very operational and easy to understand requirements artifacts such as US have been adopted in agile methods.

US are indeed used as primary requirements artifacts in agile methods, especially in XP and Scrum. Within this chapter, the role of US in agile methods has been discussed. They are more than only small pieces of functionality represented on story cards and form the fundamental basis of communication between the development team and the customers. In addition, US differ from other requirements artifacts such as IEEE-830 documents, Use-Cases, Scenario, Personas, and User Task Models. The US gathered at the beginning of the project, at the beginning of a release or at the beginning of an iteration are used as guidance within several conversations and discussions between developers and customers/users. A US driven approach starts with gathering the different US and writing them on so-called story cards. Some US of the project backlog are then the scope elements of the coming iteration, and, at the end of each iteration agile practitioners proceed to a *re-prioritization of requirements* as well as a *re-evaluation of the criteria used in the prioritization*.

The way of writing US is diverse—i.e., agile practitioners create their own US templates following their sensibility and do not associate precise semantics to their practices. This leads to difficulty in interpreting US templates. The nature of the elements present in the US can be interpreted differently since readers of each US template can be misled when using the US template. This motivates the research of the next chapter which defines a unified model for US templates. That contribution is then further used as a supporting pillar for the other contributions of the thesis.

Part III

Towards More Formality in Agile Methods' Requirements Engineering

Chapter 5

Unifying and Extending User Story Models

Within Agile methods, *User Stories* (*US*) are mostly used as primary requirements artifacts and units of functionality of the project. The idea is to express requirements on a low abstraction basis using natural language. Most of them are exclusively centered on the final user as only stakeholder. Over the years, some templates (in the form of concepts relating the WHO, WHAT and WHY dimensions into a phrase) have been proposed by agile methods practitioners or academics to guide requirements gathering. Using these templates can be problematic. Indeed, none of them define any semantic related to a particular syntax precisely or formally leading to various possible interpretations of the concepts. Consequently, these templates are used in an ad-hoc manner, each modeler having idiosyncratic preferences. This can nevertheless lead to an underuse of representation mechanisms, misunderstanding of a concept use and poor communication between stakeholders. This chapter studies templates found in literature in order to reach unification in the concepts' syntax, an agreement in their semantics as well as methodological elements increasing inherent scalability of US-based projects.

The research exposed in this chapter has been realized in collaboration with Y. Wautelet, M. Kolp, and I. Mirbel. Results have been published in the proceeding of the 26th *International Conference on Advanced Information Systems Engineering* (*CAiSE 2014*, [154]).

This chapter is organized as follows. Section 5.1 provides the research context. Section 5.2 provides some related works. Section 5.3 explains the research methodology in detail. Section 5.4 exposes how each syntax and semantic are selected. Section 5.5 presents the meta-model of the US template. Section 5.6 provides the evaluation of the model based on two case studies. Section 5.7 discusses the validity of our model. Finally, Section 5.8 provides the conclusion of the chapter.

5.1 Research Context

US constitute the main key artifact serving for requirements engineering in agile methods; this is particularly the case in *eXtreme Programming* (*XP*)

[15]. US are very operational documents describing user functionalities on a low-level basis. Basically, a US is made to be written in natural language even if initially a few templates have been proposed. With the years, practitioners developed more templates which they used at their best convenience (e.g., [78, 87] for formal sources and [8, 38, 79, 107] for informal ones). The US template is structured in the following way: *As [the **WHO**], I want/want to/need/can/would like [the **WHAT**], so that [the **WHY**]*. In other words, US allow inherently to address the three following fundamental elements (called the *dimensions* in this research) of requirements engineering: *WHO wants the functionality, WHAT functionality end-users or stakeholders want the system to provide and the reason WHY the end-users or stakeholders need the system for*. These dimensions are materialized by a *syntax* in a US template, like the elements between angle brackets in: *As a <role>, I want <goal> so that <benefit>*.

With practically no definition (called *semantics* in the rest of the chapter) associated with the elements constituting the US templates, the interpretation is often hazardous. This leads to a need of accuracy, precision and unification. We consequently propose to build a unified model defining a set of US templates. We therefore started from frameworks issued of *Goal-Oriented Requirements Engineering (GORE)*. The use of GORE frameworks, inherently high-level, is our deliberate choice. Indeed, to the best of our knowledge, they include the richest sets of modeling constructs for system analysis. Alternative choices could have been made and the priority among GORE frameworks within the research could have been different leading to build a different model. The aim is, however, not to evaluate each possible unified model but to build one that could allow to associate a single (one option only) syntax/semantic to every (no lack) dimension of a US template. For this purpose, the unified model is evaluated empirically onto sets of US issued of real life projects.

Finally, we argue that a unified syntax coupled with precise semantics would allow to enhance the potential of these requirements artifacts. Indeed, the use of a well-defined set of non-redundant terms would:

- Reduce communication issues between the agile project stakeholders;
- Reduce scalability issues of US-based agile methods (see [116]). Indeed, by furnishing constructs that can be better structured, hierarchized and grouped on the basis of their nature, the (iterative) planning of the software development could be based on elements with a higher abstraction level (i.e., broader scope). It will allow to divide the software problem into pieces better manageable for huge software developments;
- Ease the querying and reasoning onto US.

5.2 Related Work

Patton [116] highlights the difficulties of agile practitioners to deal with a project involving a huge number of US for implementing the information system of a hospital. Indeed, when dealing with about 800 US, the hierarchy was rather

difficult to determine and the big picture of the system while performing the planning game. Scalability is thus definitely an issue in agile projects with US as poor requirements engineering artifacts. He decided to introduce the US template “As <User>, I want<Task>, so that <Goal>” in order to define a hierarchy in the form of Goal, Task and US but with no semantic associated.

Series of papers have proposed enhancements to handle the requirements engineering process into agile software development, most of them focus on scalability issues and granularity of elements; however, to the best of our knowledge none has proposed a unified model with associated semantics. Leffingwell [87], Vlaanderen et al. [150], and Vähäniitty and Rautiainen [143] identified issues in the usage of backlog items alone. Backlog items are US or non US-based but still a text based requirements sets. It does indeed not allow to represent the business strategy (or the long-term business goals). They highlight the project should thus include not only the backlog items, but also *Epics*, *Themes*, *Visions*. The backlog items are the lowest requirement items and respectively Epic, Theme and Vision are elements representing the software problem through higher level entities. They however did not provide any explanation on how to map the backlog items to the upper levels.

5.3 Research Method

Figure 5.1 illustrates the research process. First, the dataset has been built; on that basis we have defined a candidate model which has finally been validated on two real life case studies. These steps are depicted into this section.

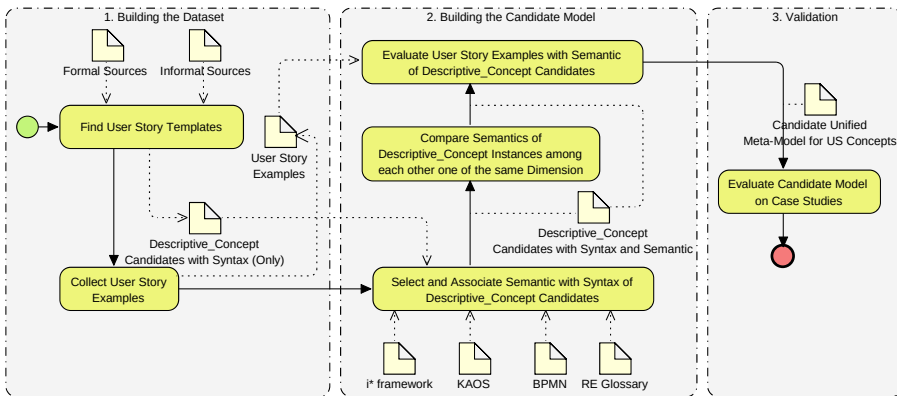


Fig. 5.1 Followed research process.

5.3.1 Building the Dataset

Initially, the way of writing US was rather fuzzy; it mostly consisted of a small text of maximum 2 lines to describe some functional expectation or scenario involving the final user. Even if Cohn [36] proposed an initial way of writing

US, many users of agile methods have been suffering from the lack of guidance in how to write an effective US [116]. Some of them thus proposed their own solutions and plenty of templates used in an ad-hoc manner appeared. In line with this, the research sources that have been taken into account are of two types:

- Academic: which consisted in overviewing the US templates found in ‘formal’ or semi-formal sources (published scientific articles or books);
- Practice: which consisted in overviewing the US templates found in ‘informal’ sources (mostly websites and blogs).

The aim was to list and basically classify the US templates that are used in practice. No higher importance was given to formal sources, even if Table 5.1 explicitly shows the number of syntaxes issued of both types of sources.

Basic references for agile development as well as sources found using scientific publications search engines were primarily taken into account. Then, a web search on Google¹ allowed us to fill the set of US templates with others issued of the daily use of agile methods practitioners. The search included informal sources like websites, blogs, etc. (i.e., html pages); only sources considered relevant (i.e., referring to a practical use) were taken into account. The search included: (1) ‘User Story Template’; (2) ‘User Story’ $\wedge$ ‘XP’; (3) ‘User Story’ $\wedge$ ‘eXtreme programming’; (4) ‘User Story’ $\wedge$ ‘Agile’; (5) ‘Agile Requirement’ $\wedge$ ‘User Story’. The first ten pages (i.e., 100 links since 10 pages multiplied must be multiplied by 10 sources by page) provided were taken into account. All pages were carefully scrutinized; relevant US templates were included². We finally included 20 US templates issued of formal sources and 65 of informal ones³ (see Appendix B); this constitutes our first research material.

Unfortunately, no semantic description associated with any of the syntax (except for the *Business Value* syntax for which we found a vague definition, see Section 5.4.3) was ever found in literature, this made any direct semantic evaluation impossible. Nevertheless, in nearly all the cases, examples associated to the proposed US templates were provided. We collected 237 examples; this constitutes our second research material (see Section 5.3.3).

¹The data was collected by a junior researcher (PhD candidate) onto the Belgian French Google version. Another local version or its consultation at another moment of time can lead to the collection of other templates. However, since we have collected a significant number of them, it would have impacted redundancy or brought genuine occurrences in a non-significant amount with no impact on the model.

²We only considered the US templates structured around a WHO, WHAT and (possibly) WHY dimension; other forms of US templates were not taken into account for this study. In addition, we only used basic search option of the search engine for doing search.

³The addition of the figures in Table 5.1 within one dimension surpasses the number of collected templates. Indeed, into one template we can find several syntaxes for a dimension; e.g., the US template *As a <type of user>, I want <capability or feature> so that <business value or benefit>* generates 2 occurrences for the WHAT and the WHY. The input material for our research was based on a Google search at the mid of 2013. Later generation of a Google search based on the same keywords is likely to produce a different research input with a possible consequence on the research output.

5.3.2 Descriptive_Concepts in User Stories

In order to be able to study the relevant concepts within US templates, we first decompose these to keep the syntaxes and their related dimensions only. Such an element is, for the sake of uniformity, characterized as a *Descriptive_Concept* (D_C) in the present study. Figure 5.2 shows the D_C in the form of a class. When building the dataset, each element that we find in a US template and that relates to one of the 3 dimensions will be an instance of that class. As an example, for the template *As a <role>, I want <goal> so that <benefit>*, we will have 3 instances of the D_C class, one for *role*, one for *goal* and one for *benefit*. The attribute *dimension* thus compulsorily takes one of the values *WHO* (e.g., for *role*), *WHAT* (e.g., for *goal*) or *WHY* (e.g., for *benefit*) and the attribute *syntax* takes the syntax found within the dimension. Finally, the attribute *semantic* will eventually be instantiated later through the use of GORE frameworks.

Descriptive_Concept
dimension : ENUM{WHO,WHAT,WHY}
syntax : String
semantic : String

Fig. 5.2 The Descriptive_Concept class.

5.3.3 Building the Candidate Model

Three sets of D_C instances can be distinguished: one for the WHO, one for the WHAT and one for the WHY dimension. For each dimension, a table was built including each of the D_C instances found and their number of occurrences.

Then, the syntax of each instance of the D_C class was associated to a semantic issued of GORE, business modeling or general requirements engineering literature. More precisely, we used the following sources:

- The i* modeling framework [165], an agent and goal-oriented framework which has been applied in many fields including requirement engineering, software process development, business redesign, business organization, security, etc. [164]. The framework was taken *at large* since it includes the contributions of the whole Tropos [29] methodology⁴. [165] was used as a main source since it encompasses significant work done around the framework;
- The KAOS framework [145], a requirements engineering framework based on goal modeling. [145] was used as a main source since it encompasses significant work done around the framework;
- The Business Process Model Notation (BPMN) framework [109], a well-known and industry adopted framework for representing business processes;

⁴Tropos is the software development methodology using i* in the requirements stages.

- A glossary of requirements engineering terminology [58], which collects, defines and translates most of the concepts used in software engineering.

In order to build the model, each instance of the D_C class leads to consult these sources in a sequential order. Indeed, the syntax of each instance was compared to the different syntaxes proposed in the frameworks. When a match was found between the syntax issued of a US template and one issued of the consulted framework, we proceeded to a preliminary adoption⁵. This means that the semantic issued of the framework was associated to the D_C so that we dispose of a couple syntax/semantic that can be further evaluated later. We can take the example of the D_C associated with the syntax $\langle \text{role} \rangle$: the i^* framework was firstly taken into account. Within this framework, the syntax *role* was present. A match was thus directly found and the semantic associated to the *role* in the i^* framework was adopted for the D_C attribute *semantic*. If it was not the case, we would have done the same process for the second framework (KAOS), and so on. If finally no match could be established in any of the evoked frameworks, it was left out of the study. This stage is referred to as *Syntax Included and Semantic Association* in Section 5.4. A list of definitions of each syntax are exposed in Appendix D.

After the process depicted in the previous paragraph was performed for each D_C instance of a particular dimension, each semantic was firstly compared to the other ones of the same dimension in order to evaluate possible redundancy/overlap/mismatch. When issues were identified, some of these instances were left out for the semantic evaluation onto US examples. This stage is referred to as *Comparison of Associated Semantic* in Section 5.4.

Each remaining D_C was then compared to each of the set of US examples (known as *second research material*). The goal was to find how many examples could be related to the semantics of each D_C to evaluate its relative importance. This stage is referred to as *Semantic Evaluation on Examples* in Section 5.4.

The model was built by a senior researcher (PhD graduate). After, it was evaluated by a junior and two other senior researchers. Elements that lead to discussions were carefully evaluated and discussed until a consensus was found.

5.3.4 Validation

Once the final model has been built, it has been evaluated onto two sets of US issued of two real-life projects. This has been done by a junior researcher then cross checked by a senior one. We started from the US sets and evaluated to what class of the unified model it belongs in order to determine coverage and completeness. Results are discussed in Section 5.6.

5.4 Selected Semantic Associated to the D_C Class Instances

Table 5.1 summarizes the different syntaxes that we have found for the WHO, WHAT and WHY dimensions. The reader can find between brackets next

⁵We always respected the defined framework hierarchy to ensure higher internal consistency of the produced model.

to each syntax, the respective number of occurrences found in US templates (*number of occurrences found in formal sources + number of occurrences found in informal sources*). The syntaxes in bold are the ones that were associated with a semantic; the other ones were left out of the process of building the candidate model before a semantic was associated. Full rationale for each of these dimensions is given in the rest of this section.

Table 5.1 Instances for Descriptive_Concept and related syntax.

WHO	WHAT	WHY
Role (13+31)	Goal (4+18)	Business Value (7+18)
Type of User (8+15)	Something (3 + 10)	Benefit (7+18)
User (0+10)	Action (4 + 7)	Reason (4+14)
Actor (0+6)	Feature (4+7)	Goal (3+6)
System Role (0+1)	Function (1 + 7)	Achievement (0+4)
Persona (0+1)	Desire (0+6)	Rationale (0+2)
'x' (0+1)	Functionality(1+4)	Desire (0+2)
	Capability (3+1)	Outcome (0+1)
	Task (1+2)	Result (0+1)
	Activity (1+2)	'z' (0+1)
	Outcome (0+2)	
	Behaviour (0+1)	
	Description (0+1)	
	What (0+1)	
	'y' (0+1)	

5.4.1 The WHO Dimension

5.4.1.1 Syntax Included and Semantic Association

As shown in Table 5.1, we found six different syntaxes into the WHO dimension. We decided to group *User* and *Type of User* into a single instance of the *D_C* class since a *User* inherently refers to a *Type of User* through its instantiation. In the same way, we have preliminarily left out the syntaxes *System Roles*, *Persona* and '*x*' because the number of their instances found was not significant and only supported by informal sources.

Using the method depicted in Section 5.3.3, the semantics associated to the remaining syntaxes were:

- Role: “*A role is an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor*” [165];
- User: “*A user is a person who uses the functionality provided by a system*” [58];
- Actor: “*An actor is an active entity that carries out actions to achieve goals by exercising its know-how*” [165].

5.4.1.2 Comparison of Associated Semantic

As explained in the research method, the semantics we have associated to the syntax of *Role*, *User* and *Actor* have been compared to each other. We first emphasize that the semantics associated to *Role* and *Actor* are issued of the *i** framework and, as such, can thus be evaluated complementarily. Concretely, *i** includes both concepts because the framework distinguishes *Actors* at a high-level of abstraction and, as mentioned in their definition, *Roles* are used into a specific context. Concretely, this high-level of abstraction is not present in the WHO entities into the US examples that we have studied and each of them always refer to a specific context. Consequently, none instances would be qualified as an *Actor* but rather as a *Role* with respect to the semantic issued of *i**. The *D_C* with syntax *Actor* is thus judged non-relevant and eliminated from the candidates to be integrated in the unified model.

5.4.1.3 Semantic Evaluation on Examples

The semantics associated to *Role* and *User* were further compared to the list of examples built-up as a research dataset. In every of the studied US examples, we found either the word *User* used as example (so not instance of *D_C*) within the WHO dimension, or a specific role played by a user of the system. At this stage, *User* can thus be the syntax of an instance of the *D_C* class or the syntax of a US example. This is misleading so that we suggest to only keep the instance of the *D_C* class associated to the syntax *Role*; the syntax *User* can be used in a US but only as an instance of *Role*. Figure 5.3 thus only owns one class related to the WHO dimension; the *Role*.

5.4.2 The WHAT Dimension

5.4.2.1 Syntax Included and Semantic Association

Something was directly left out of the study. Indeed, even if the number of occurrences is high (3 + 10), no semantic could be found in the source frameworks and it is inherently too vague/imprecise/broad to be taken into account. No semantic could be associated to *Action* and *Function*. The only semantic we found for *Action* (“... an auxiliary operation associated with a state transition” [145]) is in the context of UML [110] state chart diagrams, which are design diagrams documenting the states of the object so non relevant for the present purpose. Similarly, we also only found a semantic related to *Function* in the context of object-oriented design which was stated as non relevant for the present context. They were thus left out before evaluation. No direct semantic for *Desire* was found into the envisaged frameworks, we nevertheless can point to the *Belief-Desire-Intention* (BDI) model [45] for a semantic related to *Desire*. The BDI paradigm nevertheless refers to the design stage of an agent paradigm and, when integrated with *i** elements, the desire explicitly refers to an actor goal. So the concept is redundant with the goal concept and located specifically into a design context; we have thus decided to leave it out of the evaluation. Finally, we have preliminarily left out the syntaxes *Outcome*, *Behavior*, *What* and “*y*” because the number of their occurrences was

not significant (1 or 2 informal sources only). Using the method depicted in Section 5.3.3, the semantics associated to the remaining syntaxes were:

- Goal: we decided to include semantics of hard- and soft-goals to evaluate the opportunity of including both notions into our unified model for US. More particularly we associated the following semantics:
 - Hard-goal: “A hard-goal is a condition or state of affairs in the world that the stakeholders would like to achieve” [165];
 - Soft-goal: “A soft-goal is a condition or state of affairs in the world that the actor would like to achieve. But unlike a hard-goal, there are no clear-cut criteria for whether the condition is achieved, and it is up to the developer to judge whether a particular state of affairs in fact achieves sufficiently the stated soft-goal” [165].
- Feature: “A feature is a delimitable characteristic of a system that provides value for stakeholders” [58];
- Functionality: “Functionalities are the capabilities of a system as stated by its functional requirements” [58];
- Capability: “A capability represents the ability of an actor to define, choose, and execute a plan for the fulfilment of a goal, given certain world conditions and in the presence of a specific event” [165];
- Task: “A task specifies a particular way of attaining a goal” [165];
- Activity: “An activity represents work that a company or organization performs using business processes. An activity can be atomic or non-atomic (compound). The types of activities that are a part of a Process Model are: Process, Sub-Process, and Task” [133].

5.4.2.2 Comparison of Associated Semantic

The *Feature* and *Functionality* could be compared since they both refer to properties (characteristic or capability of a system). This similarity could be problematic. We first of all decided to look for enhancements in the semantics of *Feature* to gain confidence with interpretation/identification onto the set of examples. We notably consulted literature about Feature-Oriented Development. Apel and Kästner [11] define a feature as “a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option”. The *Feature* thus is unique when compared to the other semantics because it refers to part of the system that satisfies a functional or non-functional requirement [21] and thus inherently shapes part of the structure of the system-to-be. Due to the perceived similarity in semantic between the concepts of *Feature* and *Functionality*, we decided to only keep one. We have thus chosen to integrate the *Feature* as a candidate D_C because of the higher presence of the term in the US templates and in the context of agile development in general and the most precise semantics we have.

Then we can legitimately be willing to compare the *Capability* and *Task* since both semantics are issued from the i^* /Tropos framework and both point, in the chosen semantic, to the achievement of a goal. When comparing them, one can notice that they differ in the way they relate to a subject. While the *Task* is a particular way of attaining the *Goal* not related to any subject, the *Capability* explicitly refers to an ability (*define, choose and execute a plan*) of a particular actor to fulfill a *Goal*. Inherently, [165] defines the *Capability* on a design level rather than on an analysis one. We will nevertheless keep it into the evaluation in order to consider its relevance in the use of US where elements are often expressed at the frontier between analysis and design.

Finally, the *Task* and *Activity* must be semantically compared too because the application of these semantics can be conflicting. Within the definition of an *Activity*, we find an explicit reference to the *Task* syntax. In BPMN, the *Task* refers to some atomic behavior which is not compulsorily the case with i^* . Both the *Task* and the *Activity* refer to behavior in order to achieve a higher level element known as business process for BPMN and *Goal* for the i^* framework. We believe that these elements can thus be seen as overlapping but do not use the same terminology since they belong to different modeling paradigms. Since we give higher priority to i^* , we decide to eliminate the *Activity* from the candidate D_C instances and only keep the *Task* as candidate.

5.4.2.3 Semantic Evaluation on Examples

Empirically, we were able to find occurrences of each of the D_C instances selected within the studied examples.

Following the used semantics, there is nevertheless an over representation of the *Capability* (88% of the cases). Indeed, in a lot of cases, the US is expressed in its WHAT dimension as a *Capability* of a role instance.

We distinguish the *Task* from the *Capability* through the way they are expressed; the *Task* is expressed like a general intention constraining the system while the *Capability* is expressed as a direct system offering. The *Task* should thus be kept in order to be able to specify a way of acting to achieve a *Goal*; it can, in a sense, be seen as a way of constraining the system-to-be for achieving a goal during the requirements stage. For example, the US *As ..., I am required to log into the system so that ...* represents a *Task* for the WHAT dimension because it is expressed in the form of a constraint on the system, while *As a ..., I can start a new game* represents a *Capability* for the same dimension because it is expressed in a more direct manner.

Instances of the *Hard-goal* are also present, for example *As a borrower, I want to pay off my loan*. It nevertheless always concerns an Epic US—i.e., US that need to be refined. Similarly, occurrences of the *Soft-goal* in the WHAT dimension are also present but are even more rare; we for example find *As a player, I want nice looking background art that integrates with the game boards*. This shows that elements issued of GORE frameworks are envisaged in the context of US and we can make further use of it in the context of agile development.

We finally discuss the *Feature* element. Examples such as *Search for ...*, *Undo/redo move*, ... are sometimes considered as instances of *Feature* in the studied US but we do not believe this is in line with the semantic that we have associated to the *Feature*. It is rather aligned with the one that we have associated to *Capability*. A *Feature* is indeed, according to our semantics, a broader aspect of the system requiring to fulfill the 3 conditions set up in [11]. The US *As a salesman I want car to be equipped with GPS so that I can easily set my direction* could be considered as clearly integrating a *Feature* into the WHAT dimension. We nevertheless point to the interpretation as a *Goal* and leave the *Feature* concept not as a *D_C* instance but rather an element that can be used at higher level to group US around a central theme. This way of doing is in line with the use of the *Feature* element in the Scrum method.

5.4.3 The WHY Dimension

5.4.3.1 Syntax Included and Semantic Association

Some information was found into informal literature about agile development related to the syntax *Business Value*. [5] indeed points out that “*business value is a concept that describes the relative worth of any development effort to the business. Business value is often unquantifiable, but often relates to money ... The relative business value of stories can generally be determined by asking questions to get to the root value proposition of each*”. We do not really consider this as semantic for evaluation since it is more about the characteristics of *Business Value* than the description of how *Business Value* could be expressed in itself. It inherently refers to an umbrella term of a goal or objective to be attained. No *Business Value* syntax relating to semantics was found in the input frameworks. As we will see in the rest of this section, we do have a set of semantics directly referring to objectives to be attained that are defined in a much more precise manner. Even if the representation of the syntax is very high into the studied templates, we believe that leaving it out of the model to favor precise semantics is the best option.

No semantic associated to the syntax *Benefit* and *Reason* was found in the envisaged frameworks. For the same reason *Business Value* was left out, we decided not to consider them. We have also preliminarily left out the syntaxes *Achievement*, *Rational*, *Desire*, *Outcome*, *Result* and ‘*z*’ because the number of their instances found was not significant.

Using the method depicted in Section 5.3.3, the semantics associated to the remaining syntax, the goal which includes both the *Hard-goal* and the *Soft-goal*, are the same as in the WHAT dimension.

5.4.3.2 Comparison of Associated Semantic

The analysis of the WHY dimension will thus firstly be limited to the evaluation of examples with respect to i* hard-goal and soft-goal definitions.

5.4.3.3 Semantic Evaluation on Examples

On the evaluation, most of the examples could not be considered as goals because, even if they are expressed as objectives, they are onto a too low level to be considered as (abstract) i* goals. We indeed could only find alignment with the *Hard-goal* and *Soft-goal* semantics on 53% of the cases, leaving the rest without association.

We thus need the inclusion of other *D_C* instances to cover the WHY dimension. Nevertheless, as evoked earlier in this section, no satisfying instance was found among the collected syntaxes; we thus suggest relaxing the dimension characteristic of the *Task* element so that it can be used into the WHY dimension. Indeed, we intuitively believe that the semantics associated to the *Task* element would fit most of the examples found into the WHY dimension because they are expressed as a requisite on a lower level basis. If we take the following example ... *so that I know my recent deposit went through*. We assume that it cannot be considered as a *Hard-goal* because it is expressed on a too low-level basis, a corresponding *Hard-goal* could be for example *Make deposits* while *knowing that the recent deposit went through* is only a way of attaining the higher-level *Hard-goal* (could be to attain a soft-goal too). By including the *Task* element into the WHY dimension we were able to cover all of the examples.

5.5 A Unified Model for User Story Templates

Figure 5.3 represents the instances of the *D_C* class that have been selected on the basis of the previous study to be part of the unified model.

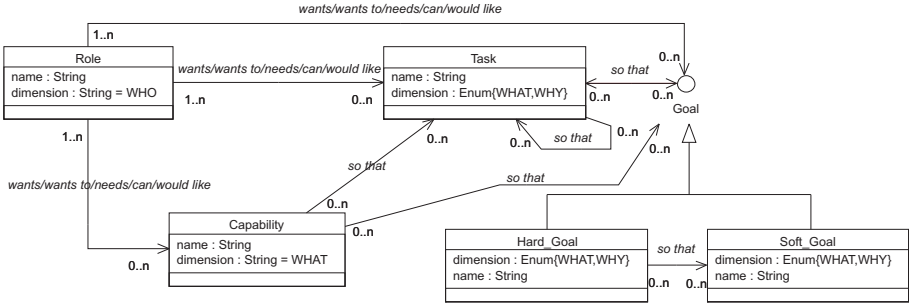


Fig. 5.3 Unified model for user story Descriptive_Concepts.

Each instance has become a class itself meant to be instantiated within the requirements gathering stage. The links between the classes represent the possible links between the elements into a US template issued of the model.

The link between the classes conceptually represents the link from one dimension to the other. Concretely, the unidirectional association from the *Role* to one of the class *Capability*, *Task* or *Goal* implies that the target class instantiates an element of the WHAT dimension (always tagged as *wants/wants*

to/needs/can/would like in the model). Then, the unidirectional association from one of these classes instantiating the WHAT dimension to one of the classes instantiating the WHY dimension (always tagged as *so that* into the model) implies that the target class eventually (0 as minimal cardinality) instantiates an element of the WHY dimension. An US template we can derive from the model is: *As a <Role>, I want/want to/need/can/would like <Task> so that to <Goal>.*

Let us finally note that the *Goal* class is represented as an interface because it cannot be instantiated as such; it is either a *Hard_Goal*, either a *Soft_Goal*. As shown within the research, the instance of *Hard_Goal* and *Soft_Goal* can be related to the WHAT or WHY dimensions. Also, if a *Hard_Goal* is related to the WHAT dimension, it can be linked to a *Soft_Goal* in the WHY dimension, not if the WHAT dimension is a *Soft_Goal*. This because it has never been found in any of the examples so we believe it is impossible to have a *Soft_Goal* as desired state to fulfill a *Hard_Goal*.

5.6 Validation

The unified US model has been applied to two different case studies in order to evaluate the coverage—i.e., *Is each of the elements required?*—and completeness—i.e., *Are more elements required?*—of the model.

The first case study is the development of an application to ease carpooling. Carpooling is the sharing of car journeys so that more than one person travels into the car; it takes increasing importance to save gas, reduce traffic, save driving time and control pollution. *ClubCar* is a multi-channel application available as an Android application, SMS service and IVR system. Users of *ClubCar* are riders and/or drivers, they can register by SMS, voice or through the Android Apps. Roughly speaking the software allows drivers to propose rides and submit their details with *dates*, *times*, *sources* and *destinations* while riders can search for available rides [134]. The project included a total of 28 US.

The second case study is called *CalCentral* has been developed by the University of Berkeley. *CalCentral* “... is an online system that delivers a unified and personalized experience to students, faculty and staff, facilitating the navigation of campus resources, delivering personal notifications from key campus systems, and supporting learning and the academic experience” [26]. US are used as requirement artefacts in the project; the list of 95 US that is available at [25].

Figure 5.4 summarizes results of the application of the conceptual model on the two case studies. It notably shows that each of the concepts included into the model has been required to characterize each of the US of the cases (even if the *Task* concept has not been required in the WHY dimension of the carpooling case). If we consider the two case studies, we thus have full coverage. In other words, each of the elements included in the unified model are necessary for concrete representation of the US found in these two real life projects. This means that, as far as these two projects are concerned, no element is superfluous. Similarly, all of the US of the case studies taken into account could be covered by using the set of elements considered in our model. We thus also have full

completeness. This concretely means that we are able to interpret each US of these projects with the set of US templates that we can derive from our unified model. As such, the US is thus associated with a well defined structure.

Finally, the reader should note that the WHY dimension was not present in each of the US; it was thus only evaluated when it was present in the US.

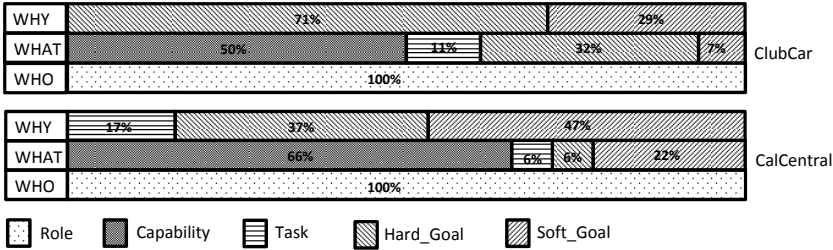


Fig. 5.4 Elements coverage in the Carpooling and CalCentral case studies.

5.7 Threats to Validity

One may argue that the choice made in the semantics was arbitrary. The choice has been made to start from GORE frameworks because these are the ones proposing the most advanced concepts for requirements representation and have a significant supporting community. Similarly, one may argue that each couple syntax/semantic of each of the frameworks used as source do form a whole and that we can hardly take a concept from one method and another from another. That is the reason why we proceeded sequentially and gave priority to the best ranked source. We nevertheless admit that the choices of the used framework and their respective priority could have been different leading to some differences in the proposed model and associated semantic.

The number of collected examples for each US template was unequal which could have biased the relative importance of each of the concepts found. We have always been aware of this reality but we do not consider it as an issue since we were studying the whole coverage of the model. Moreover, the validation is aimed to partially solve this issue by showing such a relative importance of the *D_C* instances onto real life case studies.

One may question about the level of abstraction of the US expressed in agile projects and question whether i* hard- and soft-goals semantics could be suitable to characterize elements in US by nature operationally oriented. The results that have been presented show that high-level elements are in a few occasions present notably in the form of Epic US.

5.8 Conclusion

Various syntaxes with no associated semantics have made the use of US ad-hoc and mostly operational only. This has lead to several problems in the use of agile methods notably in large development projects. This chapter has

provided a set of concepts with syntax and associated semantic for a more precise use of US. The objective is to be able to address larger projects with the same requirements artifacts and to include the overall benefits of GORE into agile methods use. The use of the framework could indeed help—through hierarchization and stepwise refinement—to enhance scalability possibilities of agile methods. Refinement of *Goals* using *Tasks* and *Capabilities* and grouping around Themes, the project can be divided easier into loosely coupled parts that can be developed rather independently within balanced iterations.

The research exposed in this chapter lead to building a unified (meta-)model for US templates but its real use and added value has not (yet) been established. Next chapter makes further use of the meta-model and builds up a graphical notation on its basis.

Part IV

Graphically Representing User Story Elements: Identifying Granularity, Interdependencies and Scope of Requirements

Chapter 6

Building a Rationale Tree for Evaluating User Story Sets

Requirements representation in agile methods is often done on the basis of *User Stories (US)* which are short sentences relating a WHO, WHAT and (possibly) WHY dimension. They are by nature very operational and simple to understand thus very efficient. The research exposed in the Chapter 5 allowed to build a unified model for US templates associating semantics to a set of keywords based on templates collected over the web and scientific literature. Since the semantic associated to these keywords is mostly taken from the i^* framework, we overview, in this chapter, how to build a custom *Rationale Tree (RT)* on the basis of a US set tagged using that unified template. The RT is strictly speaking not an i^* *Strategic Rationale (SR)* diagram but uses parts of its constructs and visual notation to build various trees of relating US elements in a single project. Indeed, the benefits of editing such a RT is to identify depending US, identifying Epic ones and group them around common Themes. This chapter shows the feasibility of building the RT, then points to the use of these consistent sets of US for iteration content planning. To ensure the US set and the RT constitute a consistent and not concurrent whole, an integrated *Computer-Aided Software Engineering (CASE)* tool supports the approach.

The research exposed in this chapter has been realized in collaboration with Y. Wautelet, M. Kolp, I. Mirbel and S. Poelmans. Results have been published in the proceeding of the 10th *International Conference on Research Challenges in Information Science (RCIS 2016, [155])*.

The chapter is structured as follows. Section 6.1 provides the research context of the chapter. Related work is discussed in Section 6.2 while Section 6.3 exposes the research design. Specifically, a meta-model of elements aiming at grouping US (macro-level) as well as decomposing US (micro-level). Section 6.4 explicitly maps the elements of the US template meta-model with the elements of the SR model to use its reasoning techniques with a project's US. Section 6.5 abstracts the different cases we can face within the edition of US using the reasoning approach of the SR model, how Epic US can be identified in these cases and how US can be grouped around Themes. Section 6.6 discusses its inclusion in the agile software process while Section 6.7 discusses the automation of the approach and its support through a CASE-Tool. Section 6.8 discusses

the validity, the threats to validity, the scalability of the approach and future works. Finally, Section 6.9 concludes the chapter.

6.1 Research Context

With respect to the model built in Chapter 5, one may indeed question about the utility of such a model for agile practitioners. In the end, why should US be ‘tagged’ to a certain template or keyword and not simply expressed following the WHO/WHAT and WHY structure without more refinement. The main advantage of tagging is that, if done respecting the semantics associated to the concepts, it gives information about the nature of the US element (and thus also its granularity if the element is functional). Such information could possibly be used later on for analysis or structuring of the problem as for example pointed out by [91]. Structuring of US is often done with the *User Story Mapping* (USM) technique (see [118]); the latter uses *Story Maps* (SM), which are hard to maintain and read, so that other techniques for visual representation could be welcome.

In this perspective, we suggest to explore the visual representation of US starting from a set of US tagged following the model exposed in Chapter 5. Since the latter unified model is largely inspired by i^* semantics, this chapter overviews how one can build a diagram in the form of a tree using the constructs of the i^* SR diagram with the elements contained in sets of US. The goal is thus *not* to use the SR as such, but to build a graphical notation largely inspired by the SR convenient for the representation of the US elements and studying their refinements, compositions and decompositions in order to group them consistently. In the requirements engineering process built out of our contribution, we point to keep up with agile principles and to build the set of US first then to generate a RT¹ on their basis. We indeed do not believe that starting from i^* modeling in agile development could, as such, be adopted as an alternative to USM because the approach is very abstract and often starts with as-is requirements representation so is not really in line with what agile modelers are expecting for requirements representation. Nevertheless, an i^* -like diagram furnishing a consistent visual representation of an existing US set thus providing a graphical representation of the system-to-be only provides added value and is in line with agile expectations.

Our proposal is illustrated through a running example about carpooling (see Section 5.6).

6.2 Related Work

Our work aims to organize US on the basis of a proper granularity analysis. Among other sources, the need for granularity levels in US-based modeling has been identified in [91]. The problem of poor scalability of agile methods

¹In this chapter, we refer to the RT as the diagram that we build in order to visually represent US elements issued of a US set and their links. It is strictly speaking not a SR diagram but uses close notation and constructs (this is built-up and motivated in this chapter). For a complex case this RT is made of several decomposition trees.

because of poor granularity identification in requirements representation has been identified in [86, 106].

The process of transformation from a set of US to RT can be compared to a more formal approach to USM (see [118]). USM is the industry adopted technique that relates the most to our approach. Within a project, USM is intended to produce a SM which is a map of a project's US according to (i) the level of abstraction, (ii) the sequence (horizontal dimension), and (iii) the priority (vertical dimension).

Basically, USM defines three layers: the *backbone* which represents an entire user activity (or process), the *walking skeleton* which represents a user task and the *slice US* which represents a small, implementable and concrete US [118]. We informally evaluate a possible alignment with elements present in our approach (see Chapter 5 for their definition). The USM *user activity*—which is an abstract objective—could then be compared to a US containing a *Goal* in its WHAT dimension while the USM *user task*—which makes the former objective more concrete—could be compared to a US containing a *Task* in its WHAT dimension and, finally, the USM *implementable US*—which is the most concrete and atomic one—could be compared to a US containing a *Capability* in its WHAT dimension. We could thus say that, by nature, granularity of elements is not what distinguishes our approach from USM. This is rather interesting since the model of Chapter 5 that we base our approach upon has been built from existing sets of US templates and examples and, empirically, also distinguished 3 required granularity levels. Nevertheless, our approach diverges from USM in the use of a graphical notation inducing the use of formal links between US elements while USM uses story cards with a color-coding technique. This allows only limited expressiveness and flexibility in US manipulation. Indeed, one finer-grained US can then only relate to one coarser-grained US where we could define multiple links.

When compared to USM, a bit more effort is required with the technique we propose in this chapter since we split a US in 2 or 3 dimensions and we use the graphical representation of i^* . Nevertheless, this leads to:

- A graphical representation of requirements. SM remain limited to post-its on a board or even on the ground;
- A structuring of requirements where we can:
 - Systematically eliminate redundant US elements. A SM is aimed to achieve a comparable process but with a refinement on the basis of the WHAT and WHY dimensions; our process offers finer possibilities;
 - Study the dependencies of US to other US (thus multiple possible dimensions) notably useful for the identification of Theme US. SM hierarchy is limited since a US can only be under the column (scope) of one Epic US.

We thus argue that with comparable modeling effort we make use of a more precise model to build the system-to-be.

Chapter 8 envisages a transformation approach from sets of US tagged with the same model (see Section 5.5) to a Use-Case diagram. Roughly speaking

we point to the transformation of goal elements as well as some task elements to Use-Cases in a Use-Case diagram. The approach delivers a coarse-grained representation of the system-to-be but fails to bring a decomposition approach necessary to study US inter-dependencies as we build-up in this chapter.

6.3 Research Method

This section exposes the ‘building blocks’ used within this research. Our main goal is to be able to group US on the basis of their interdependencies. A few concepts have been proposed in agile literature² from which we have built up a meta-model. At the macro-level, detailed in Section 6.3.1, relationships with US concepts are highlighted. US of various granularity levels require to be composed/decomposed into other US or be grouped with other US around common Themes. *Features* are also required for the proper fulfillment of US. We also propose to decompose US through their dimensions in so called *Descriptive Concepts* (see Section 5.5) to allow their analysis. Section 6.3.2 depicts this micro-level.

6.3.1 Macro-level: Ways to Organize User Stories

We first distinguish a US macro-level dealing with the grouping of US into depending (i.e., relating) sets. We thus envisage here the US as potential building blocks serving for project composition/decomposition. US are indeed subject to a process of sorting and dropping early on into the development project [92]; an initial identification of the US hierarchy using precise semantics could help (i) addressing priorities among coherent and non-redundant sets of US and (ii) manage changing requirements by understanding the impact of changes.

Figure 6.1 presents our meta-model of the US concepts in its project environment. At this macro-level, the aim is to identify what elements could be used (i) to group US, (ii) to abstract them or (iii) to see what technical elements should be provided by the system and that they are concerned with.

The *User_Story* class represents the US characteristics as a whole. Chronologically, *US* are written by the customer or product owner at the earliest stages of the project and put in the product backlog with an (implementation) *priority* and an *amountOfPoints* which refers to the number of *User Story Points (USP)*³ [36, 87]. These elements have thus been added as attributes to the *User_Story* class. Other attributes required for process management are included within the *User_Story* class. Indeed, US are written onto *User Story Cards (USC)*. To support their implementation, we enrich the *User_Story* class with a *status* attribute which contains the status of the US on the USC. The value of the *status* in the USC can be threefold: *Open User Story (O_US)*, *In Progress User Story (IP_US)* and *Completed User Story (C_US)* [116]. In addition, the *conversation* attribute contains the detailed discussion about the *US*.

²We have focused on the available sources describing the eXtreme Programming (XP) and SCRUM methods.

³The amount of USP represents the estimated effort required to implement the US.

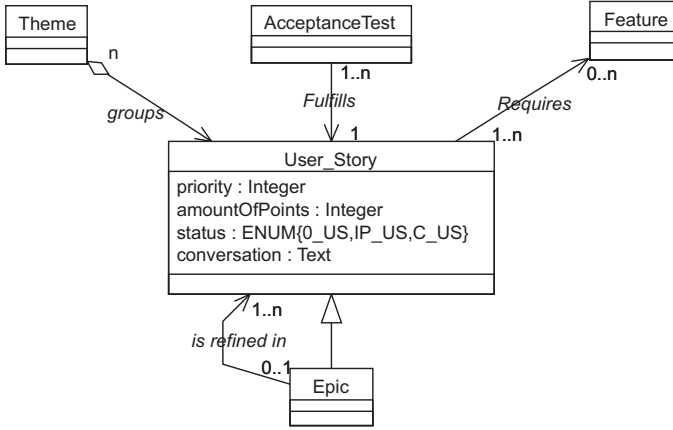


Fig. 6.1 US as Macro-Level structures: Meta-Model.

The *AcceptanceTest* [36] class encapsulates the set of predefined tests for a US. This is used to validate whether the US satisfies stakeholders' requirement(s). This can be a normal/abnormal scenario and is defined by the tester.

Some US need to be refined into other ones since they are too abstract (coarse-grained) to be estimated, implemented and tested at once. These are called Epic US [36]. The latter are indeed US with a high-level of abstraction meaning that they must be refined/decomposed into smaller US to describe the requirement more precisely. These US are represented by the class *Epic* inheriting from the class *User_Story*.

A Theme is a collection of related US [36]. We model it using the Theme class as a grouping of a set of lower level US.

Finally, *Features* inherently relate to technical elements not expressed into US but that must be provided by the system. Indeed, a feature “*is a delimitable characteristic of a system that provides value for stakeholders*” [58]. This definition can be refined by “... *a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option*” [11]. As highlighted in [21], the feature is unique when compared to the other semantics because it refers to part of the system that satisfies a functional or non-functional requirement and thus shapes part of the structure of the system-to-be. The *Feature* class represents the concept.

6.3.2 Micro-level: Decomposing a User Story in Descriptive_Concepts

Within Figure 6.2, the meta-model of the previous section is enriched with the constituting elements of the US; we refer here to this US view as the micro-level.

Rather than using the US as a whole within the requirements analysis process, we suggest, in our research design, to decompose the US on the basis of their *WHO*, *WHAT* and, when available, *WHY* dimensions. For the sake of uniformity, these elements are all characterized as *Descriptive_Concepts* (D_C). When decomposed into a set of D_C , the dependency between D_C is

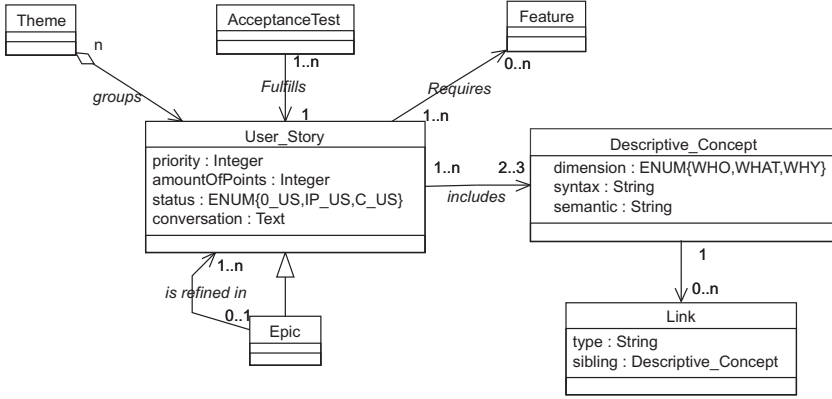


Fig. 6.2 US as Macro- and Micro-Level structures: Meta-Model.

intended to be further studied (see Section 6.4). Each element of a US template relating to one of the 3 dimensions is then an instance of the D_C class. For the template *As a <role>, I need a <task> so that <goal>*, we have 3 instances of the D_C class: one for *role*, one for *task* and one for *goal*. The *dimension* attribute thus compulsorily takes one of the values *WHO* (for *role*), *WHAT* (for *task*) or *WHY* (for *goal*) and the *syntax* attribute takes the syntax of the concept name (e.g., *role*, *task*, *goal*, ...). The *semantic* attribute relates to the definition of the D_C . The list of all the possible D_C is given in the form of a meta-model allowing to define US templates in Section 5.5.

Finally, since different D_C can be linked together, we introduce the *Link* class that represents the possible different types of links between two D_C . The possible instances of the *Link* class will be studied in Section 6.4.3.

6.4 A Graphical Notation for User Stories Dependency Analysis: Micro-Level Approach

The purpose of this section is to explicitly map the concepts of the unified model of US templates with the concepts taken from the SR model; this would allow to derive a relevant graphical notation to be used for reasoning around a project US.

6.4.1 The WHO Dimension: Graphical Notation

Within the WHO dimension, we only find, in the unified model, the *Role* concept.

The *Role* concept has semantics issued from the i^* framework and is thus ‘natively’ supported by the SR model with a defined icon (see Figure 6.3). Similarly, the boundary of the actor is defined as a circle associated to the role as within the SR model. This graphical notation is thus also adopted here within the graphical representation of WHO dimension US elements.

6.4.2 The WHAT and WHY Dimensions: Graphical Notation

Within the WHAT and WHY dimensions we find, in the unified model, the *Task*, *Capability* and *Goal* concepts. The latter must be a *Hard-goal* or *Soft-goal* so that we in total have 4 concepts that need to be represented in these two dimensions.

All of these concepts, except the *Capability*, have semantics taken from the i* SR model and are thus supported by a defined icon (see Figure 6.3). These graphical notations are thus also adopted here for these 3 concepts within the graphical representation of WHAT and WHY US elements.

The *Capability* concept with its associated semantics is not a requirements modeling concept but rather an agent-oriented design one not ‘natively’ supported by the SR model. The relevancy of its inclusion in the unified model has been discussed in Chapter 5 and, as evoked earlier, we keep it for the modeling of atomic *Tasks* (of course performed by a determined *Role*). As shown in Figure 6.3, we introduce a genuine icon for this concept. We also add a constraint on this element: *it cannot be used as an entire mean in a means-end decomposition* (see next section) *because it is atomic (i.e., low level and concrete)*.

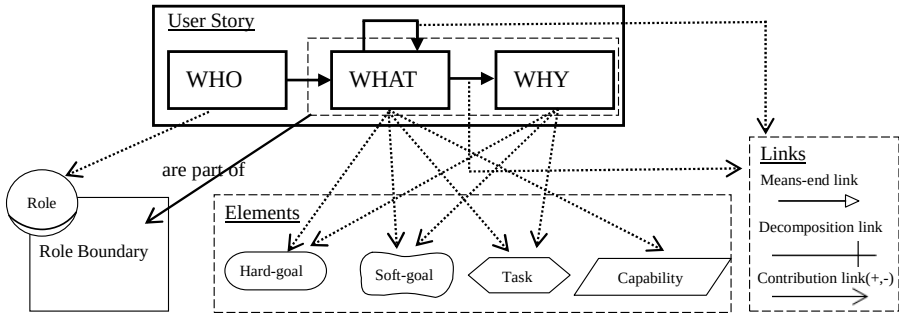


Fig. 6.3 Icons used within the representation of the user story elements using the Strategic Rationale reasoning.

6.4.3 Linking Descriptive_Concepts of the Unified User Story Model

Now that we have set-up the icons for the different elements, we need to study the possible types of links between elements of the WHAT and/or of the WHY dimension. Three types of links between elements are specifically defined for the SR model; these are:

- Means-end links which “indicate a relationship between an end, and a means for attaining it. The “means” is expressed in the form of a task, since the notion of task embodies how to do something, with the “end” is expressed as a goal. In the graphical notation, the arrowhead points from the means to the end” [165];

- Decomposition links are more specifically associated to tasks, indeed “*a task element is linked to its component nodes by decomposition links*” [165]. Moreover, “*a task can be decomposed into four types of elements: a subgoal, a subtask, a resource, and/or a softgoal – corresponding to the four types of elements. The task can be decomposed into one to many of these elements. These elements can also be part of dependency links in Strategic Dependency model(s) when the reasoning goes beyond an actor’s boundary*” [165];
- Contribution links for contributions to Soft-goals, indeed “*any of these Contribution Links can be used to link any of the elements to a Soft-goal to model the way any of these Elements contributes to the satisfaction or fulfillment of the Soft-goal*” [165].

When a WHY dimension is present into a US, we can deduce that there is a link between the elements of the WHAT and the WHY dimensions even if this link is not necessarily a direct one. Intuitively we could think that there is compulsorily a means-end link (the WHAT element is a mean to attain the WHY element) but this cannot be stated as a rule (thus nor automated). Indeed, empirically a lot of cases can be found where the element in the WHY dimension is very coarse-grained and the element in the WHAT dimension atomic. Then, the WHAT element is just a step in the realization scenario rather than an entire mean to achieve the element in the WHY dimension. This means that if the WHAT element is a *Capability* or a *Task* located on a low level basis, that is, a step or partial set of steps in the realization of an element expressed in a very coarse-grained manner in the WHY dimension (like a *Hard-goal* but could also be a *Task* or a *Soft-goal*), then there can be, in the diagram representing the set of US, elements between the elements of the WHAT and WHY dimensions of this single US.

The modeler has to create the links between the D_C in function of the requirements/domain analysis. The study and linking of elements lead to a tree hierarchy in a SR diagram fashion. That way an analysis of the alternatives (means-end) and of the possible redundancy (in the decompositions) could also be performed.

Note that a decomposition within our model can cross the boundaries of a single role. This is represented in the form of a dependency within a classical i* SR model. We here focus on decomposition only, so that we do not include dependency associations that would increase the complexity of the diagram (see examples in Table 6.2 and Figure 6.5).

6.5 Towards a Rationale Analysis for User Stories

Hierarchy and Grouping: From Micro to Macro Level

This section is aimed to study how the i* SR models’ constructs can be used to build a custom RT (so it is not strictly speaking an i* SR diagram but it is however largely inspired by it), aligned with a set of US in order to highlight Epic US and group US belonging to the scope of an Epic one (so sharing a same Theme). Indeed, after the graphical mapping of elements made within

the previous section, the remaining relevant question is to determine *how to characterize an Epic US and determine US relating to the same Theme on the basis of a rationale analysis?*

Intuitively, we envisage the Epic US as the ones containing elements at the highest level of the hierarchy of a decomposition model. *Capabilities* can thus not be considered for possible inclusion in the top-level elements category because they should be expressed as role decisions on a very low (atomic) level. Similarly, *Soft-goals* are by nature non-functional so that they are also not included in the category. Relevant top-level elements can thus be *Hard-goals* or *Tasks*. *Hard-goals* are abstract and need to be operationalized so that we choose to focus on elements with a concrete realization scenario and they will not be considered for being Epic US. Only the *Task* concept is then remaining and we define a **top-level Task as a Task element that is not issued of the refinement of another Task element but that itself needs to be refined in more elements**. Decomposition complexity and possible finer grained hierarchization of elements will be further discussed in Section 6.6.

Different possible cases for Epic US and Theme US identification are discussed in the rest of this section.

6.5.1 A Top Level Hard-goal (End), One Mean

6.5.1.1 Description

If, for a top-level *Hard-goal*, there is only one means-end decomposition (which represents a possible way of satisfying the *Hard-goal* through a *Task*), then the US containing the *Task*⁴ at the source of the means-end decomposition as *D_C* is an Epic US. The rest of the (lower level) elements in the scope of this means-end decomposition belong to US of the same Theme.

6.5.1.2 Example

Table 6.1 presents a set of US issued of the ClubCar application development. The scenario described in this section is represented in Figure 6.4 both in canonical form and instantiated to the US of Table 6.1. The US including the *Task* “Propose a ride from A to B with the price, location and time of departure, and number of seats available” is thus an Epic US because it is the top-level *Task* issued of the means-end decomposition of the *Hard-goal* “Propose a ride to go from A to B”. Moreover, the US containing all the elements refining the top-level *Task* are part of the same Theme⁵. Also note that the *Capability* “Select appropriate service” was not initially present into the US set but has been added to ensure the consistency of the rationale analysis. Also note that the presence of the *Soft-goal* “Rider satisfied of my driver service” has no impact on

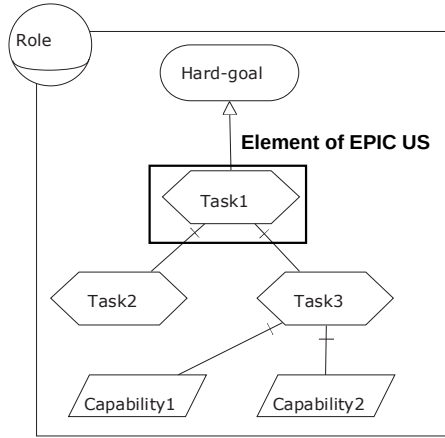
⁴Note that the *Task* could potentially be present in different US in the WHY dimension. We refer here to the US in the WHAT one. It may be that we need to create the Epic US (thus with the *Task* in the WHAT dimension) because it can be that it is not expressed as such in the set of US of the project.

⁵In our approach, a US containing a *Capability* element (necessarily in the WHAT dimension, see Section 5.5) can belong to different Theme groupings because these steps are relevant for the execution scenarios of different Epic US.

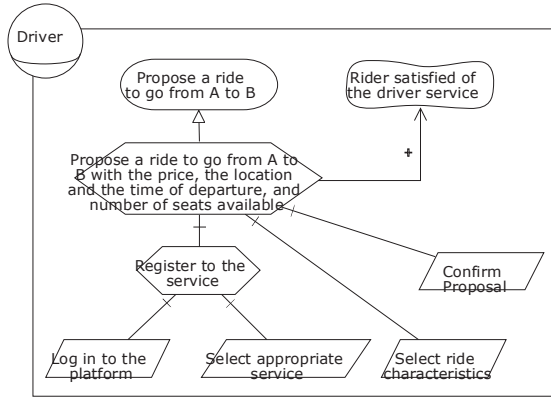
Table 6.1 US set 1 sample issued of the ClubCar application development.

<i>Dimension</i>	<i>Element</i>	<i>D_C Type</i>
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want to register to the service</i>	Task
WHY	<i>so that I can propose ride to go from A to B.</i>	Hard-goal
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want to propose a ride from A to B with the price location and time of departure, and number of seats available.</i>	Task
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want to log in to the platform</i>	Capability
WHY	<i>so that I can register to the service.</i>	Task
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want to select the ride characteristics.</i>	Capability
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want to confirm the proposal.</i>	Capability
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want the RIDER to be satisfied of my service.</i>	Soft-goal

the rest of the Epic and Theme identification process but its identification and representation can constitute a guidance for designers later on in the software engineering process (the use of *Soft-goals* in the software architecture and design is however outside the scope of this chapter).



(a) Canonical Model



(b) ClubCar Example

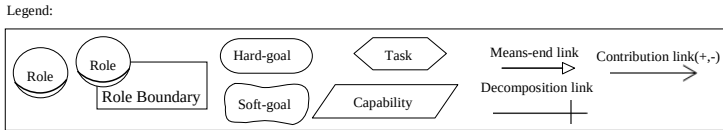


Fig. 6.4 Top-Level Hard-goal, One Means-End decomposition.

6.5.2 A Top Level Hard-goal (End), Several Means

6.5.2.1 Description

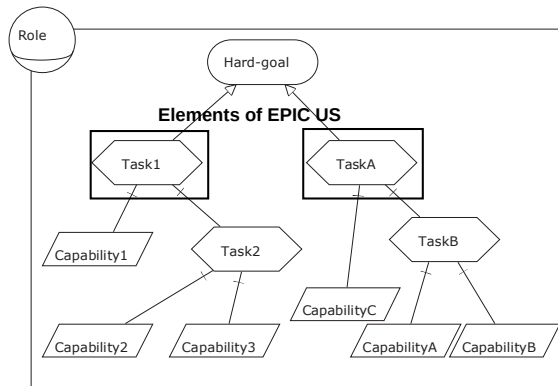
If, for a top-level *Hard-goal*, there are two or more means-end decompositions (which represent possible ways of satisfying the *Hard-goal* through *Tasks*), then the US containing the *Tasks* involved in the means-end decomposition as D_C are considered as Epic US. The rest of the (lower level) elements in the scope of each particular means-end decomposition belong to US of the same Themes.

Table 6.2 US set 2 sample issued of the ClubCar application development.

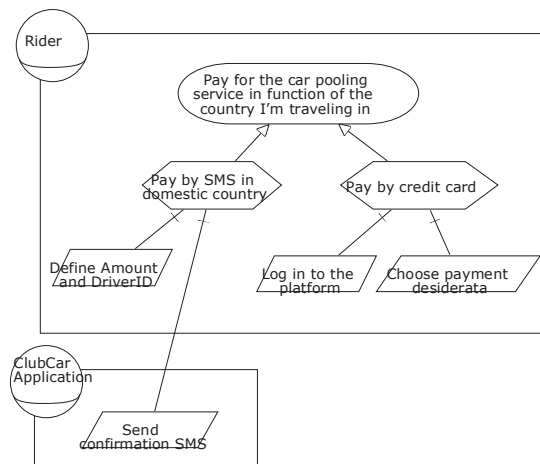
<i>Dimension</i>	<i>Element</i>	<i>D_C Type</i>
WHO WHAT	<i>As a RIDER</i> <i>I want to pay for the car pooling service in function of the country I'm traveling in.</i>	Role Hard-goal
WHO WHAT	<i>As a RIDER</i> <i>I want to pay by credit card.</i>	Role Task
WHO WHAT	<i>As a RIDER</i> <i>I want to pay by SMS in my domestic country.</i>	Role Task
WHO WHAT WHY	<i>As a RIDER</i> <i>I want to be able to select the payment desiderata so that I can pay by credit card.</i>	Role Capability Task
WHO WHAT	<i>As a RIDER</i> <i>I want to log in to the platform.</i>	Role Capability
WHO WHAT WHY	<i>As a RIDER</i> <i>I need to defined the amount and the driver so that I can pay by SMS.</i>	Role Capability Task
WHO WHAT WHY	<i>As the ClubCar Application</i> <i>I want to send a confirmation SMS so that when the payment by SMS has been performed.</i>	Role Capability Task

6.5.2.2 Example

Table 6.2 presents a set of US issued of the ClubCar application development. The scenario described in this section is represented in Figure 6.5 both in canonical form and instantiated to the US of Table 6.2. The US including the *Task* “Pay by SMS in domestic country” as well as the US including the *Task* “Pay by credit card” are thus Epic US because these are top-level *Tasks* issued of means-end decompositions of the *Hard-goal* “Pay for the car pooling service in function of the country he is traveling in”. Moreover, the US containing all the elements refining these two top-level *Tasks* are part of the same Themes (so we have two Themes represented in Figure 6.5).



(a) Canonical Model



(b) ClubCar Example

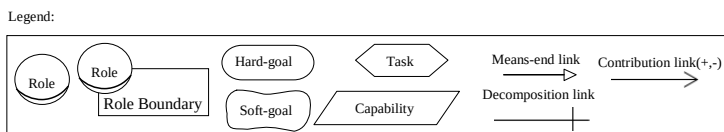


Fig. 6.5 Top-Level Hard-goal, several Means-End decompositions.

6.5.3 A Top Level Task, a Direct Decomposition

6.5.3.1 Description

For a top-level *Task* not linked with a *Hard-goal* through a means-end decomposition, the US containing this *Task* is considered as an Epic US. The rest of the (lower level) elements in the scope of this *Task* decomposition belong to US of the same Theme.

6.5.3.2 Example

This scenario is similar to scenario in Section 6.5.1 but without an upper *Hard-goal* of which the *Task* represents the means-end analysis. Because of this similarity, it is not illustrated here.

6.6 Impact on the Agile Software Process

This section studies the possible impact of integrating US analysis with the RT into a US-based agile method.

6.6.1 Impact of Changing Requirements

The transformation approach from a set of US to the RT using the constructs of an SR diagram as presented in this chapter is applied to a static set of US. Discovery and ability to deal with changing requirements is nevertheless one of the core willingness of agile methods. When requirements change US are adapted so that the RT is impacted. The impact of a change in a US on other requirements (i.e., US) can be studied on other US. In other words, the impact of a change of a US or several US can be studied on the RT by overviewing the links of changing US elements with other US elements. This cannot (or hardly) be achieved on the basis of the list of US only or with USM and could thus provide added value at this level. Consistency between the set of US and the RT is ensured by the use of a CASE-Tool (see Section 6.7).

6.6.2 Impact Iterative Planning

One of the main potential use of the RT built out the set of US is as input to the planning game⁶. Indeed, as evoked, building trees within the RT allows to distinguish Epic US and group US under a common Theme. Iterative and incremental development precisely requires such information for consistent iteration content planning.

Not all top-level *Tasks* found in a RT should necessarily be the exclusive focus of one iteration. Indeed, an Epic US grouping a set of common Theme US could require to be treated in multiple iterations (or the opposite). This depends on the couple time/effort that can be/or is willing to be deployed on a single iteration:

- The type of iterative method to be used may vary. An agile method like *eXtreme Programming (XP)* [15] or Scrum [128] tends to iterate more times than a method like the Rational Unified Process (RUP⁷) [56, 69, 82] which includes 8 or 9 iterations at maximum in the whole

⁶The planning game is the process (in agile methods like XP) of selecting the requirements on which the development team will focus during an iteration and aligning these requirements with the available development capacity (see for example [12, 15]).

⁷Note that the RUP is not US driven but Use-Case driven. It is also strictly speaking not considered as an agile method. Its life cycle template could nevertheless be driven by US and used in an agile fashion.

project. Consequently, the time spent on an iteration may vary in function of the used methodology or life-cycle template;

- The amount of available resources to work on a single iteration may vary;
- The preferences of the software development team may vary leading to be willing to do more or less iterations in various amounts of time.

Calibration in the planning of Epic US in the fashion developed in this chapter is thus required in function of the evoked parameters.

6.6.3 Generic Iterative Planning Template

With respect to the elements seen so far and the discussion that preceded in this section, we illustrate a possible decomposition of the i*-like RT for iterative planning. Figure 6.6 shows such a generic diagram in its canonical form. We refer to scope elements as elements that can be used as a basis for iterative planning. As shown in the picture, the scope element can be the entire Epic US meaning that the entire Theme should be prototyped/developed for the iteration or just a decomposed US meaning that we consider just part of the Theme US elements for prototyping/development into that iteration and that the entire Theme elements could be validated over multiple iterations. An approach to determine the right scope element could also be to evaluate the number of US points required for the development of the US containing the potential scope element.

Figure 6.6 presents a generic template that represents a possible approach for iterative content planning of the i*-like RT developed with the transformation method overviewed in this chapter. Prioritization of Epic US for iterative planning can be done on discussion with stakeholders or with a structured approach like in [158], this however remains outside the scope of this chapter.

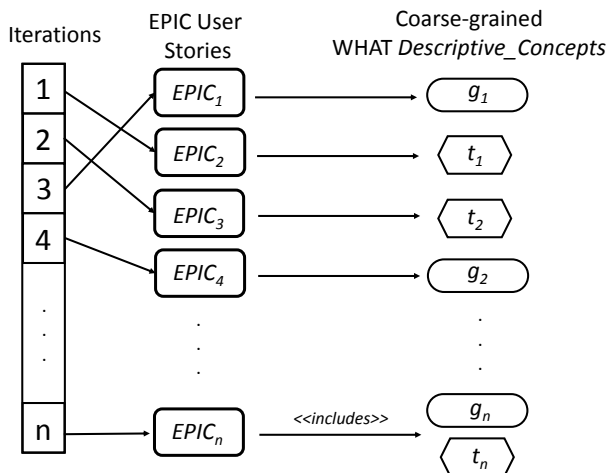


Fig. 6.6 Portfolio optimization problem.

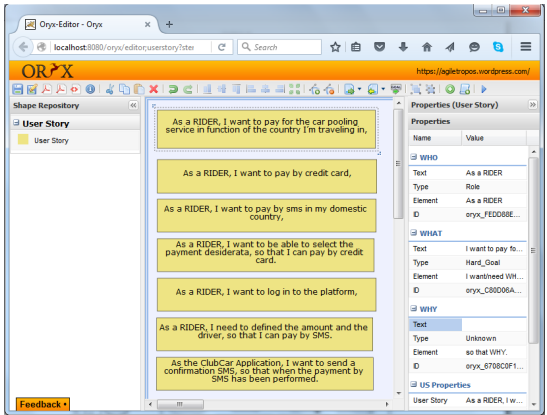
6.7 A CASE-Tool for Automating the Approach and Round-Tripping Between Views

In order to support the edition of US sets on US cards as well as the RT, we have build an add-on to the cloud version of the Descartes Architect CASE-Tool [46] that, for the present purpose, allows three views:

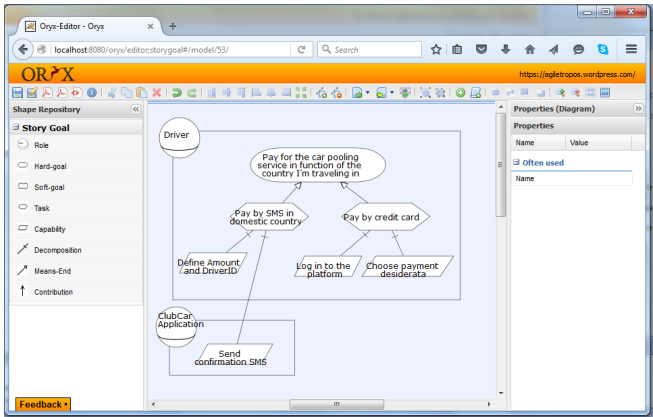
- The *User Story View (USV)* to edit US through virtual US cards. Each US element in a dimension must be tagged with a concept of the unified model;
- The *Rationale View (RV)* to edit RT following the specification made in this chapter. The graphical elements can be automatically generated from the US defined in the USV; the modeler is then in charge of further editing of the links between elements. When changes are made to graphical elements in the RV, the elements are automatically updated in the USV and vice-versa. These do indeed form the same logical element represented in different views;
- The *Structural View (SV)* to edit a structural agent diagram. This agent architecture is outside the scope of this chapter;
- Next to this, we can also edit classical UML diagrams.

Once again, as a prerequisite, the set of US needs to be tagged to start the transformation and round-trip between the views. The editing process is continuous and intensive over the requirements analysis stage⁸ (and to a certain extend over the entire project life cycle). This process is of course fully supported by the tool and leads to automatic updates of complementary views; consistency is ensured by separating the conceptual element in the CASE-Tool memory from its representation in a view.

⁸In practice, during requirements analysis some US elements are ‘retagged’ in an ad-hoc manner in later modeling stages. Indeed, the composition of the RT mostly leads to reconsider the nature of some elements (of which the granularity and structure was hard to determine when only seen in an isolated manner in the first stages) like in any modeling method.



(a) User Story View



(b) Rationale View

Fig. 6.7 The supporting CASE-Tool.

6.8 Validity, Threats to Validity, Scalability of the Approach and Future Work

As already evoked, the prerequisite to the use of our approach is to tag the US when setting them up. Nevertheless, in terms of time, the investment is very limited; at maximum a few minutes per US, encoding them in the CASE-Tool included. More time would then be necessary to create and edit the links between US elements in the RV. This is, however, similar to classical modeling.

A few threats to validity could also be evoked and should be clarified in later validation of the work:

- *Accuracy in US tagging could impact the relevancy of the RV.* A study on the perception of US elements' granularity using the unified model is explained into the next chapter. The study distinguished different groups of users from students to researchers. The results were better with

experienced modelers, but identifying granularity did not lead to major issues in any group with the condition that the set of US was taken as a consistent whole. This indeed allows to evaluate the relative links and hierarchy of US elements leading to adequate granularity identification and thus US elements tagging. As ‘stand alone’ elements, granularity identification makes no sense and is nearly random;

- *The accuracy of the RT with respect to the system-to-be.* In order to assess the validity of the RT in the RV, we will proceed to the following experience. At first, subjects (issued of 3 groups: students, researchers, and business analysts) will be informed about a case and asked to carefully read and tag a set of US. At second, these same subjects will be asked to rank their perceived relevancy of 3 RTs:
 - RT 1 generated and built from their own tagging of US set (so from the first part of the experiment);
 - RT 2 generated and built by the tagging of the US set by researchers participated in this research;
 - RT 3 purely randomly generated and built out of the US set.

The perceived relevancy/validity of the RT will then be evaluated by the subjects. Other metrics for the evaluation of RT will also be envisaged.

Future work also includes the application of the full validation of the technique on more real-life cases. We will notably proceed to a statistical analysis of the stakeholders perception of the relevancy and contribution of the RT for a project they have worked on. Also, they will be interviewed about the value of the use of the RT complimentary to the US sets in the agile requirements process.

The technique of transforming a set of US into a RT is virtually applicable to all sizes of projects. The complexity of the reasoning trees making part of the RT may then vary from project to project and the larger the number of US, the larger the modeling effort required. The tricky question of scalability can thus legitimately be posed. As evoked previously, our technique could be compared to USM; the latter is applied to projects of any size. Splitting US through their 3 dimensions will induce more modeling effort but using the CASE-Tool will save effort by adding flexibility in model management comparing to build a USM on a board or on the ground as it is the case for larger projects. The tradeoff of using our *transformation* technique with respect to USM must thus be balanced between a stronger US elements links’ identification plus management using a CASE-Tool against a less formal approach where US are written on post-its and presented on a board or on the ground but are organized faster.

Above this, the question of agility may also be raised—i.e., *Are we not hampering agility?* The willingness is not to revolutionize an entire practice but to add a more formal representation of requirements to be able to better manage requirements across the whole development life cycle so with a positive impact on scalability. We argue that the larger the set of US, the most benefit should be the rationale analysis because it proposes a structuring and consistent

grouping of requirements possibly used as input for iteration content planning. That said, we do not believe that the building of the RV as an impact on the agility of the current process. Indeed, we suggest our approach as a side one (but constantly kept consistent with the US set if our CASE-Tool is used). We thus want to provide structure and complementary information that can be used in the fashion that the software development team wishes (not constraining).

In the illustrative example, the sets of US allowed to build a nearly perfect decomposition which is seldom the case in real-life case studies so that the domain analysis not only requires to build the links between the elements but also to possibly add some elements so to introduce new US leading (as already evoked) to a round-trip between the set of US and the RT rather than an unidirectional transformation process. We consider this as an advantage because it adds consistency to the entire requirements set.

Finally, further studies could be achieved in order to evaluate the value of representing roles' interaction in an advanced manner. We can for now only visualize that different roles are possibly involved in the fulfillment of a coarse-grained element and what their specific fine-grained contribution is. We could study the added value of the use of dependencies in the sense defined by the *i** strategic dependency diagram.

6.9 Conclusion

US are often expressed in a very operational manner leading, in huge projects, to an explosion of US' number, consistency issues and consequently an inherent difficulty to define their hierarchy and consistently plan iterations' content. Building a visual representation where US elements could be sorted, the links between them could be studied and consistent sets of US elements could be build could thus represent a useful guidance for the agile modeler and project manager.

Enhancing on a unified-model for US templates proposed in Chapter 5, we have suggested in this chapter to build a RT largely inspired by the *i** SR diagram to dispose of such a graphical notation. Epic US contain top-level *Task* elements and interrelated elements are grouped around sets of Theme US. As an alternative to USM, this construction allows a more accurate US decomposition analysis leading to an elimination of redundant elements as well as a more accurate study of changing requirements' impact. We also point to the use of consistent sets of US elements for iterative content planning.

To fully support the edition of models, an integrated CASE-Tool has been built. Thanks to its use a change in one view immediately updates the other views in order to keep the requirements analysis consistent.

This chapter showed how a graphical notation associated to the unified meta-model of US templates allows to further study the granularity, interdependencies and scope of the elements contained in US. The next chapter studies how this graphical notation is interpreted by software modelers facing a particular US set.

Part V

An Alternative Graphical Representation for User Story Elements: Suitability of the Industry-Adopted Use-Case Model

Chapter 8

Bridging User Story Sets with the Use-Case Model

User Stories (US) are mostly used as basis for representing requirements in agile development. Written in a direct manner, US fail in producing a visual representation of the main system-to-be functions. A *Use-Case Diagram (UCD)*, on the other hand, intends to provide such a view. Approaches that map US sets to a UCD have been proposed; they however consider every US as a *Use-Case (UC)*. Nevertheless, a valid UC should not be an atomic task or a sub-process but enclose an entire scenario of the system use instead. A unified model of US templates to tag US sets was build in Chapter 5. Within functional elements, it notably distinguishes granularity levels. In this chapter, we propose to transform specific elements of a US set into a UCD using the granularity information obtained through tagging. In practice, such a transformation involves continuous round-tripping between the US and UC views; a CASE-Tool supports this.

The research exposed in this chapter has been realized in collaboration with Y.Wautelet, M. Kolp, D. Hintea and S. Poelmans. Results have been published in the proceeding of the 3rd *International Workshop on Conceptual Modeling in Requirements and Business Analysis (MReBA 2016, [153])*.

This chapter is structured as follows. Section 8.1 provides the research context of the chapter. Section 8.2 exposes related work. Section 8.3 presents a US set of the running example. Section 8.4 discusses the mapping between US elements and UC elements. Section 8.5 exposes tool and mapping rule allowing the systematic automation. Section 8.6 discusses the impact of using UC approach for modeling US in agile methods. Finally, Section 8.7 concludes the chapter.

8.1 Research Context

As evoked earlier in this thesis, the main advantage of tagging US elements is that, if the tagging respects the semantics associated to the concepts, it provides information about both the nature and the granularity of the US element. Such information could possibly be used at a further stage for software analysis: structuring the problem and its solution, identifying missing requirements, etc. [91]. Visual GORE models were envisaged for graphical representation in

Chapter 6; it showed that GORE models are very well adapted for the purpose of US sets representation. Nevertheless, since GORE models are not an industry adopted practice, we explore in this chapter an independent representation with the far more used *Unified Modeling Language* (UML [110]), *Use-Case Model* (UCM). An instance of the latter produces a UCD. A UC is a list of actions or event steps, typically defining the interactions between a role and a system in order to achieve a goal.

Facing the definition of US and UC, we can notice that there is a possible mismatch between the two concepts—i.e., the US can indeed include (very) fine-grained elements and the UC should be a coarse-grained element describing the software problem encompassing a set of fine-grained actions. When sorting is applied within US elements, a transformation process can generate a UCD. This is why, in this chapter, **we envisage the representation of problem domain coarse-grained US elements as UC**. To this end, we map the elements defined in the UCM—i.e., the *Actor*, the *UC* and the relationships between Use-Cases—with the elements defined in the unified US template model in Chapter 5.

In practice it is sometimes discouraged to use US and UCD concurrently because if not kept mutually up to date they can be inconsistent (see e.g., [62]). We therefore propose keeping US sets and the UCD consistent by auto-updating each change performed to one in the other through the use of a CASE-Tool. They are considered here as two complementary views. We identify three primary benefits of our transformation approach: (1) a graphical high-level (coarse-grained) representation of requirements through the UCD; (2) multi-dimensional structuring of requirements and US dependencies at UCD level (not possible with US Maps); (3) identification of missing requirements (not expressed in the current US set) and the elimination of redundant US.

8.2 Related Work

The CASE-Tool *Visual Paradigm* (VP) [111] already proposes to transform US into a UCD. In their approach, the actor expressed in the WHO dimension becomes an actor of the UCD and the element in the WHAT dimension becomes a UC without any selection based on granularity. The rule is simple—i.e., a US with a WHAT element becomes a UC in the UC diagram. Nevertheless, UML points to the use of UC as elements representing an entire process rather than fine-grained (or atomic) elements as US can be. This does not mean that a US cannot include elements adequately describing UC, but that a sorting process is required in order to only include relevant elements as UC. The approach of VP can thus be said to be ‘naive’ because including elements not relevant as UC.

Structuring US is often performed using the *User Story Mapping* (USM) technique (see [118]); the latter uses *Story Maps* which are difficult to maintain and read. Building a UCD from a set of US could be compared to USM. In our approach, we split a US in 2 or 3 dimensions and we use the graphical representation of the UCD for relevant elements. Our aim is to obtain:

- An easy to read graphical representation of requirements. Story Maps remain limited to post-its on a board or even on the ground;
- An advanced structuring of requirements where we can overview the dependencies of coarse-grained elements to one another with the use of `<<include>>` and `<<extend>>` relationships. Fine-grained elements are not part of the UCD but can be represented under the scope of particular UC for example in workflows (UML activity diagrams or BPMN diagrams, etc.). The latter is outside the present scope;
- An identification of gaps in requirements and elimination of redundant US elements from the graphical representation. By limiting the graphical UCD representation to solely coarse-grained elements we can obtain a high-level representation of the system-to-be. This allows to overview if, at an operational level, all aspects required to solve the software problem have been taken into account; if not they can be added to the US set. Similarly, elements appearing to be redundant because appearing in several US of the US set can lead to remove US from the set.

AgileModeling [7] emphasizes the usage of models for better understanding of the system-to-be and argues that it is useful to produce at least some models including a UML UCD and class diagram for the very first iteration called *iteration zero*. No transformation process is nevertheless provided.

8.3 Running Example

Our proposal will be illustrated using a running example about carpooling—i.e., the ClubCar (see Section 5.6).

As shown in Table 8.1, we have taken a sample of the US of the ClubCar application to illustrate the research developed in this chapter. Some of these US contain elements to be transformed in UC and elements that are not relevant as UC (see Section 8.4). The first column depicts the *Dimension* of US *Descriptive_Concept* (D_C), the second column describes the element itself and the last column provides the type of the D_C ¹.

8.4 User Stories Integration through a Use-Case Diagram

This section exposes the mapping between US elements and Use-Case diagram elements. The *Role* is discussed in Section 8.4.1. Section 8.4.2 discusses the *Hard-goal*, *Task*, and *Capability*. Finally, Section 8.4.3 discusses the *Soft-goal*.

8.4.1 The Role

A Role within a US can be forwarded to an Actor in the UCD. The UCD Actor is indeed the only structure defined to represent the WHO dimension (see Figure 8.1).

¹Note that, when there were several possibilities, a choice ensuring the consistency of the entire set has been made.

Table 8.1 US sample of the ClubCar application development.

<i>Dimension</i>	<i>Element</i>	<i>D_C Type</i>
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want to register to the service</i>	Task
WHY	<i>so that I can propose a ride to go from A to B.</i>	Hard-goal
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want to propose a ride from A to B with the price location and time of departure, and number of seats available.</i>	Task
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want to log in to the platform</i>	Capability
WHY	<i>so that I can register to the service.</i>	Task
WHO	<i>As a RIDER</i>	Role
WHAT	<i>I want to be transported from A to B.</i>	Hard-goal
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want to confirm the proposal.</i>	Capability
WHO	<i>As a DRIVER</i>	Role
WHAT	<i>I want the RIDER to be satisfied of my service.</i>	Soft-goal

8.4.2 Hard-goal, Task and Capability

Three functional elements—i.e., the *Hard-goal*, the *Task* and the *Capability*—can be found in the unified model. The *Capability* is located on a fine level of granularity and thus non relevant for inclusion in the UCD. The *Hard-goal* and the *Task* elements are aimed to contain coarse-grained elements thus relevant for inclusion as UC.

Hard-goal and *Task* elements can be distinguished by their nature (i.e., their formulation) rather than by their grain level. Both are indeed intended to represent coarse-grained elements. The *Hard-goal* is the most abstract element and the *Task* is its counterpart expressed in an operational manner. This means that the *Task* represents a way of fulfilling the *Hard-goal* (entirely or parts of it); the *Task* is thus the counterpart of the *Hard-goal* in the solution domain while the *Hard-goal* belongs to the problem domain.

In line with UML, the Use-Case does not necessarily explicit *how* the problem should be solved (this can be documented within on a workflow containing fine-grained elements) but rather *what* should be achieved. **US elements tagged as *Hard-goals* should thus necessarily be represented as Use-Cases** since they depict WHAT problem should be solved. The tricky question is then to determine if *Task* elements must also be represented as UC. Since *Tasks* are part of the solution domain, if they are represented as UC they should be linked to the *Hard-goals* they furnish part of a solution to; this means that we can make their relationship explicit. This is something useful when different US elements represented as *Hard-goals* make use of the same US elements represented as *Tasks*. It indeed allows to show that some behavior can be reused in several situations. This is what <<*include*>> and <<*extend*>> dependency relationships are intended to model in a UCD. We can then link

the UC corresponding to a *Hard-goal* element with the one corresponding to a *Task* element using an `<<include>>` and `<<extend>>` dependency in function of the particular situation. If no behavior is recycled, we do not point to the representation of the US element tagged as *Task* in the UCD because this would lead to several UC that do not need to be externalized (leading to lots of `<<include>>` and `<<extend>>` dependencies) and are only sub-processes of the UC. Moreover, we want to make clear that we do not point to use these dependencies to depict means-end relationships between *Hard-goals* and *Tasks*. We point to keep the UCD as simple as possible and leave the description of solutions to the *Hard-goal* in other views (e.g., workflows). In a phrase, only decompositions of *Hard-goals* in *Tasks* where the *Tasks* are used in multiple *Hard-goals* are represented as UC in the UCD.

There is thus no universal answer to the representation of US elements tagged as *Tasks* in the UCD; as evoked it may be interesting to highlight the reuse of more special behavior but some *Tasks* are just subprocesses of the *Hard-goal* elements and should then not be represented as UC. These need to be documented for example in workflows depicting the realization scenario(s) of the *Hard-goal* (thus UC).

Let's finally note that UC transformed from *Hard-goal* elements can also be linked with other UC transformed from *Hard-goal* elements through an `<<include>>` relationship. In our context this shows that some *Hard-goals* are possibly needed for the realization of other *Hard-goals*. We do not consider `<<extend>>` relationships among *Hard-goals* because such elements do have a possible stand-alone realization.

Concretely, in Figure 8.1, the *Task* is linked with the UC representing the *Hard-goal* with an `<<include>>` dependency relationship from the *Hard-goal* to the *Task*. This is illustrated in the left side of Figure 8.1 in a canonical form and instantiated on the Carpooling example in the right side of the same figure.

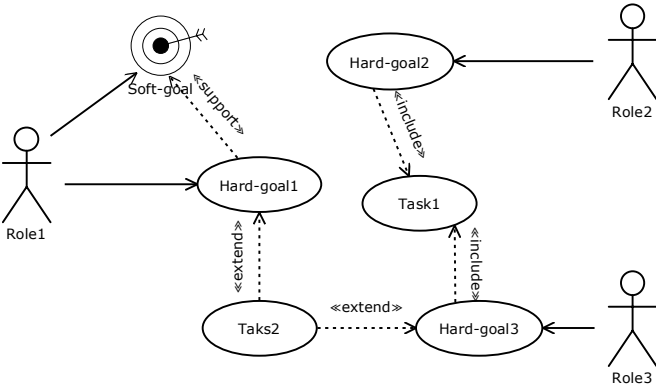
Representing both elements in the UCD is thus in some cases a way of explicitly linking the problem and solution domains where system behavior can be recycled in multiple Use-Cases (thus *Hard-goals*).

8.4.3 The Soft-goal

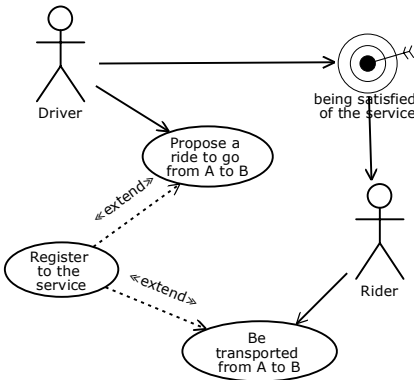
A Soft-goal is “a condition or state of affairs in the world that the actor would like to achieve” [165]. For a *Soft-goal* there are no clear-cut criteria for whether the condition is achieved; it cannot be represented as such into an element of a standard UCD. In a standard UML UCD there is no element for the representation of *Soft-goals* but a refinement of the UCD is included in the *Rational Unified Process* (RUP, see [82]) and known as the RUP/UML Business UCM (see [135]). A representation in the UCD would allow us to trace which functional requirement (in the form of a *Hard-goal* or a *Task*) supports the realization of a *Soft-goal*. Wautelet and Kolp [156] suggests to map the *Soft-goal* with the RUP/UML Goal because a semantic analysis of both definitions concludes that those represent the same type (or at least closely related) elements. This solution is relevant for us since it allows a graphical representation of *Soft-goals* in the UCD as well as a potential support analysis

(by highlighting which UC contributes to the satisfaction of the represented *Soft-goal*). Consequently and even though it is not standard UML, we map the *Soft-goal* elements from the US set to the graphical representation of the business Goal element. As shown in Figure 8.1, we can have:

- The *Soft-goal* in the WHAT dimension. Then, in the UCD, we immediately relate the Actor (*Role* in the US WHO dimension) to a Business Goal (*Soft-goal* in the US WHAT dimension) and a simple link is used;
- The *Soft-goal* in the WHY dimension. Then, in the UCD, if the element in the WHAT dimension leads to a UC (*Hard-goal* or a *Task* in the US), it can be linked to the Business Goal (*Soft-goal* in the US WHAT dimension) using a `<<support>>` dependency relationship. This link visually expresses that the functional element contributes to the realization of the *Soft-goal* within the software implementation.



(a) Use-Case model forwarded on the basis of stereotyped user stories.



(b) Partial Use-Case Diagram for the Carpooling Example.

Fig. 8.1 Use-Case diagram: Canonical form and carpooling example.

8.5 Automating the Approach and Round-Tripping Between Views

In order to support the approach, we have built an add-on to the cloud version of the Descartes Architect CASE-Tool [46] that, for the present purpose, allows multiple views:

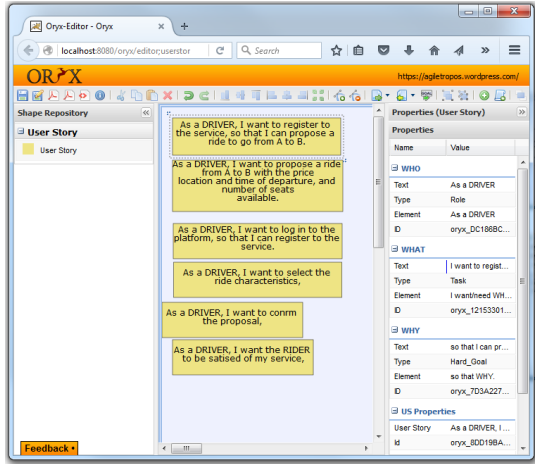
- The *User Story View (USV)* to edit US through virtual US cards. Each US element in a dimension must be tagged with a concept of the unified model;
- The *Use-Case View (UCV)* to edit a UCD. The UCD is automatically transformed from the US set defined in the USV. When changes are made to UC or Actors in the UCV, the corresponding elements are automatically updated in the USV and vice-versa. These indeed are the same logical element represented in multiple views;
- The *Class, Sequence and Activity Diagram Views* (outside the scope of this chapter).

The CASE-Tool immediately build elements in the UCV, when elements are built in the USV following the rules given in this chapter and summarized in Table 8.2.

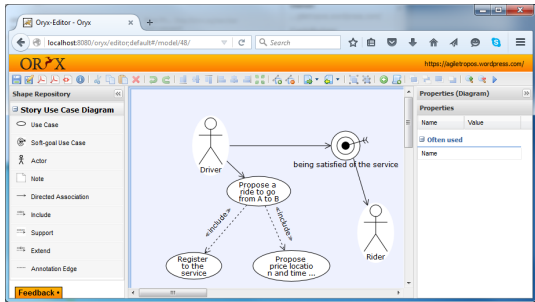
Table 8.2 Mapping a user story set with the Use-Case diagram.

<i>US Set Element</i>	<i>UCD Element</i>
Role	Actor
Hard-goal	Use-Case; several Use-Cases transformed from <i>Hard-goals</i> can be linked through <i><<include>></i> dependencies
Task	(Possible) Use-Case; the Use-Case transformed form a Task should be linked through <i><<include>></i> or <i><<extend>></i> dependencies with Use-Cases transformed from <i>Hard-goals</i>
Capability	No possible transformation
Soft-goal	RUP/UML Business Goal

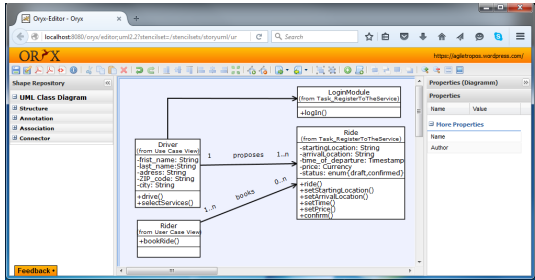
The editing process is continuous over the requirements analysis stage and over the entire project life cycle. In practice, US elements are re-tagged several times when they analyzed and structured. Consistency among views is ensured by separating the conceptual element in the CASE-Tool memory from its representation in a view.



(a) User Story View



(b) Use-Case View



(c) Class Diagram View

Fig. 8.2 The supporting CASE-Tool.

8.6 Impact on Produced Software: Future Work

Two types of impacts will be evaluated in future work:

- *What is the impact on the software design of using our approach versus using another one?* Since we aim to transform coarse-grained US elements in UC, the produced UC are likely to become the scope elements for which

realization scenarios will be build. Also, sets of US are expressed in a fined-grained way only or parts of the requirements are not expressed. In order to fill the gap, one or more UC can then be added; US are then automatically generated accordingly. Fine-grained elements can also be omitted from the US set and be identified through the approach. Finally, identifying and representing *Soft-goals* in the UCD may lead to better take them into account in the design. These aspects need further investigation;

- *What will be the variability in the UCD produced from the same US set by different modelers?* Various modelers applying the transformation will not produce exactly the same model. They are indeed likely to interpret elements differently and consequently tag them differently. Analysis activities occurring after the initial transformation often lead to reconsider some of the associated tags (granularity of US elements is thus not set once and for all but refined through the requirements elicitation). This variability needs to be studied and evaluated.

8.7 Conclusion

Agile methods use very simple requirements descriptions in the form of US. These are easy to read but difficult to structure leading to the need of visual requirements representations to sort them, understand the system-to-be, dialogue with stakeholders, etc. We have consequently suggested to structure coarse-grained elements found in US sets in a UML UCD. The UCD view is aimed to remain consistent with the set of US, encompassing changing requirements to furnish the possibility of UC driven development in methods where US sets are the firstly expressed requirement artifact. This work is complementary to previous work focusing on the representation of US sets with GORE models (see Chapter 6).

Part VI

Conclusion

Chapter 9

Conclusion

9.1 General conclusions

User Stories (US), a well-known engineering practice in agile software development, are mostly developed in a semi-structured way through a WHO, WHAT and WHY dimension. Nothing nevertheless allows to distinguish the nature and granularity of the elements depicted in the WHAT (and WHY) dimension(s). Such information is the one required to further structure requirements models, link relevant US functional elements with one another so to build a graphical representation of the to-be-software system.

This dissertation has shown that, with a bit more domain analysis and modeling effort, one could start from a US set and build such a graphical representation of requirements. The graphical representation, that we have called the *Rationale Tree (RT)*, can then be (if consistently built) used as a basis for model-driven software development in agile methods. Elements of Epic US constitute the core functionalities of the system, their decomposition possibilities are explicitly shown and documented in our RT. In the perspective of model-driven *Project Management (PM)*, elements of the RT can be—at various levels of decomposition—used as scope elements to drive the iterations’ planning depending on the size of the iterations that the team wants to deal with.

Next to this, we also showed that, using a consistent set of US—i.e., the nature and granularity of its constituting elements—could, in most cases, be adequately interpreted by its practitioners. This increases the potential for adoption of our contributions. Also, a representation through Use-Case diagrams can be made possible to dispose of a graphical representation in a form already adopted by industry. Third party contributions to model-driven development with the Use-Case model can then also be adopted starting from a US set and our transformation approach.

9.2 Summary of the main Contributions

The first contribution of this thesis is the unified meta-model for US templates. This (meta-)model has been built on the basis of 85 US templates found in different (formal and inform sources) sources. US are generally written as follows:

As a [WHO], I want [WHAT], so that [WHY]. From the 85 US template dataset, we collected the keywords related to each dimension and a selection process has been made based on the syntax and a semantic that we associated to each of these keywords. These semantics were selected out of the i*framework, KAOS, BPMN and the IREB glossary. At the end of the research process, the constituting elements of the meta-model are the *Role* for the WHO dimension, the *Hard-goal*, *Soft-goal*, *Task*, and *Capability* for the WHAT dimension; and finally, the *Hard-goal*, *Soft-goal*, and *Task* for the WHY dimension. An example of a US Template that can be built out of our meta-model is: *As <Role>, I want <Task>, so that <Hard-goal>.*

The second contribution of this thesis is the improvement of requirements engineering practices of agile methods through a graphical representation of a US set. We have proposed two types of graphical representations.

The first graphical representation builds a custom decomposition tree called the RT, it is largely inspired by the Strategic Rationale diagram of i*. Indeed, the semantic domain of the meta-model for US templates is very close to the one of i*. The RT allows to highlight dependencies between US elements and constitutes an advanced tool for domain analysis. It thus partly addresses the lacks of natural language US. Furthermore, it also allows to clearly identify the grain of the modelled elements; coarse-grained elements provide a big picture of the system-to-be.

The second graphical representation is the use of Use-Case model for representing the coarse-grained US elements. This gives a formal bridge to the further use of UML models in agile methods.

The third and last contribution of this thesis concerns the PM aspects of agile methods. The graphical representation of US elements in the form of either RT or a Use-Case model *can be used for project planning (in the planning game) of iterative life cycle methods.* These models provide a basis for the selection of US to be implemented in the next iteration(s). Since decompositions and functional alternatives are highlighted in the RT, the latter model eases the selection of a coherent US set to be implemented. Usually such a selection is made in terms of coarse-grained elements. These can be identified in both the RT and Use-Case diagram.

9.3 Future Work

This section introduces the future work that can improve the actual results of the thesis (Section 9.3.1) and also provides new research directions on the basis of this work (Section 9.3.2).

9.3.1 Improvement on User Story Model

Concerning the improvements of the actual results of the thesis, we identify:

- *Each of the individual contributions of the thesis need to gain experience through their use.* Large scale implementations of the transformation from US sets to RT is, for example, being performed these days on a real life case. The results will be analysed and depicted in future work;

- *The CASE-Tool.* As seen, a CASE-Tool has been implemented as a proof of concept and supports limited size US sets. Nevertheless, we did not test it on very large US set so that the scalability of the tool yet remains to be evaluated. Moreover, the CASE-Tool in its actual form only provides support for engineering activities (US set and RT editing, Use-Case model, ...) but no project management abilities are provided. Gantt charts could for example be easily implemented and be immediately integrated with the PM scope elements defined in the graphical models;
- *Improvement in the systematic tagging of US elements and the building of the RT.* Machine learning techniques can be used for the systematic tagging US set. Base on the work of Robber et al. [124], we can systematically extract/decompose the three dimensions of a US. The idea is then to automatically assign a type (thus tag) from the unified model for US templates to a US element. To achieve this we, at the first stage, need to build a training dataset based on human assigned tags. Once the dataset validated, it can be used to build an algorithm for automatically tagging new US sets of the same project through a linguistic approach. Then, these tagged US elements can, of course, be further used in the transformation processes to RT evoked in the thesis. Next to the last research track, it would also be interesting to calculate the semantic similarity or relationship between elements of the first generated RT—i.e., the US elements are randomly organized in the RT and no links are presented. The supporting tool can, at first, group the similar elements in to a cluster and later proposes links between two elements if there is some semantic similarity between them. This allows to have a pre-built RT that modelers can use to model the US set. This, in our opinion, can save time for software modelers;
- *Systematic planning approach.* We can easily propagate the priority and user story points attributed to US in the analysis stage to the corresponding elements in the RT. From this we can use a clustering method to cluster depending US elements so that we can suggest a default iterative planning with highest priority to be implemented US for the next iteration(s). For each cluster, the total of the user story points gives an estimation of the total iteration effort allowing to balance iterations;
- *A comparison of a linguistic approach and graphical model approach for US sorting.* While the previous points suggested to mix our approach with linguistic based ones, we could also compare the RT-based approach with the one proposed by Lucassen et al. [94] in order to evaluate which of the two methods performs better for sorting US sets. We would proceed through an experiment with student groups;
- *Use of the RT to justify US implementation choices with respect to the company strategy/tactics.* For now we consider implementation choices on an ad-hoc and operational bases only. Since we dispose of a RT for reasoning about implementation possibilities we could also model the

strategic and tactical layers of the company to highlight the positive and negative impacts of implementation choices on it;

- *Inclusion of acceptance tests in the US modeling.* For now US validation has not been taken into account. In agile projects, acceptance tests are systematically associated to US; these could also be included in the building of the RT for validation purpose.

9.3.2 Using Rationale Tree in Software Development Process

As future research direction we point to the full and smooth integration of the RT within agile methods. Currently, we have proposed a *process fragment* for the integration of agent-based software development in agile development; this fragment requires the building of RT from the US set before deriving the *Multi-Agent Systems (MAS)* architecture. The RT allows indeed to show the reasoning options and the MAS architecture follows the *Belief Desire Intention (BDI)* paradigm [157]. However, this topic is still under investigation and we provide a brief description of the process hereafter.

The process fragment is composed of three main stages: *Requirements Analysis*, *Architectural Design* and *Implementation*. Several types of stakeholders are involved in these different stages.

In order to show the process fragment's stakeholders as well as the interactions among them, we have built an i* Strategic Rationale diagram representing our software process fragment (see Figure 9.1).

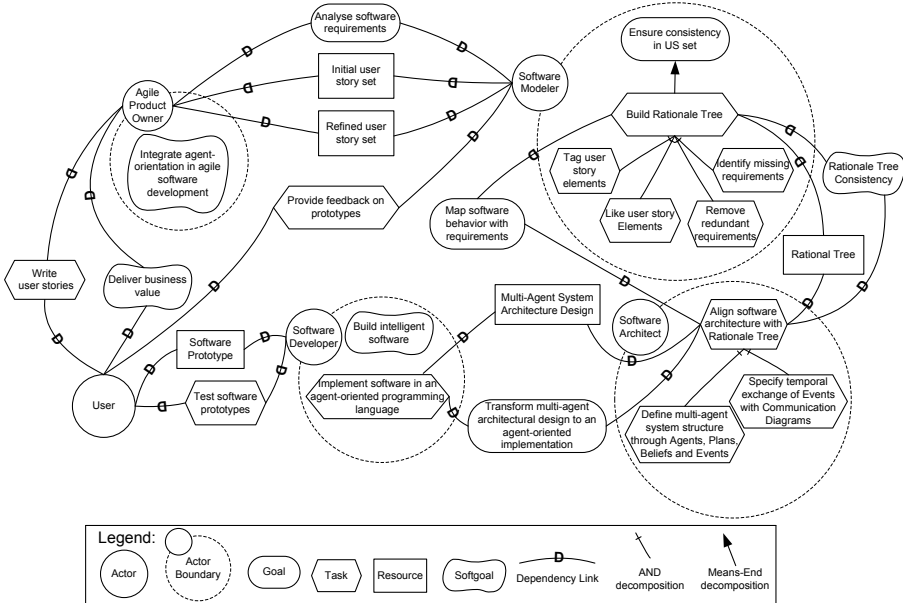


Fig. 9.1 Process fragment for integrating Agent-Oriented development in agile methods.

We distinguish 5 types of stakeholders:

- The *Agile Product Owner (APO)* is a key stakeholder of the project. Its activities are outside the scope of our process fragment and are guided by the adopted agile methods; we represent the APO here because he constitutes an entry point to our process fragment. Usually, the APO is a senior manager that is mainly in charge of developing a vision of what software system needs to be built and propagate that vision over the development team. That way *Users* expect him to lead the development of software that will *Deliver business value* (represented on the diagram as a Soft-goal dependency). The APO uses the product backlog (see [30, 112] to store the *Initial US Set* collected over multiple future users;
- The *Software Modeler* is in charge of understanding the software problem and suggest a consistent software solution for it. Basically our process fragment starts with the activities of this role; this happens during the *Requirements Analysis* stage of our process fragment. The *Software Modeler* uses the initial US set to *Analyze the software requirements*; the latter is thus represented as a goal dependency. When the analysis is performed, the main goal of the Software Modeler is to *Ensure consistency in the US set* Goal; this is achieved through *Building the Rationale Tree* Task. The latter requires the *US set's elements to be tagged using the unified template model*, the *US elements to be linked*, the *redundant requirements to be removed* and the *missing requirements to be identified* Tasks. As an output, the Software Modeler sets at disposal of the APO a *Refined US Set* and at disposal of the Software Architect a *Rationale Tree* that this latter role expects to be *Consistent*;
- The *Software Architect* is in charge of building a *MAS Architectural Design aligned with the RT*. This happens at the *Architectural Design* stage of our process fragment. The *MAS Architectural Design* is more a set of models that one stand-alone model; we notably have:
 - The *Structural Diagram* that documents all the agents involved in the software implementation as well as their *Plans*, *Beliefs* and *Events* (see [157]);
 - The *Dynamic Diagram* that captures the synchronization mechanisms between Events and Plans.
 - The *Communication Diagram* that specifies the temporal exchange of events between agents.
- The *Software Developer* is in charge of implementing the *MAS Architectural Design* furnished by the *Software Architect* within an agent-oriented development language. The latter can, for example, be the Jadex framework¹. This happens at the *Implementation* stage of our process fragment.

¹www.activecomponents.org

- The (final) *User* that will use the software application and is thus in charge of testing the software prototypes to provide feedback to the *Software Modeler*. These activities are outside the scope of our process fragment and are guided by the adopted agile methods. We represent the User here because he constitutes an exit point to our process fragment.

References

- [1] Abad, K., Pérez, W., Carvallo, J. P., and Franch, X. (2017). i* in practice: Identifying Frequent Problems in its Application. In *will be appeared in Proceedings of the 32nd Annual ACM Symposium on Applied Computing*. ACM.
- [2] Abbas, N., Gravell, A. M., and Wills, G. B. (2008). Historical roots of Agile methods: where did “Agile thinking” come from? In *International Conference on Agile Processes and Extreme Programming in Software Engineering*, pages 94–103. Springer.
- [3] Abrahamsson, P., Salo, O., Ronkainen, J., and Warsta, J. (2002). Agile Software Development Methods: Review and Analysis. [online] www.pss-europe.com/P478.pdf.
- [4] Abrahamsson, P., Warsta, J., Siponen, M. T., and Ronkainen, J. (2003). New directions on agile methods: a comparative analysis. In *Proceedings 25th International Conference on Software Engineering*, pages 244–254.
- [5] AgilesKillsProject (2012). Agile Skills Project Wiki, Agile skills inventory, business value. [online] <http://www.agilekillsproject.org/agile-skills-inventory/business-value>.
- [6] Alexander, I. F. and Maiden, N. (2004). *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*. Wiley Publishing, 1st edition.
- [7] Ambler, S. (2002). *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA.
- [8] Ambler, S. (2014). User Stories: An Agile Introduction. [online] <http://www.agilemodeling.com/artifacts/userStory.htm>.
- [9] Annett, J. (2004). Hierarchical task analysis. *The handbook of task analysis for human-computer interaction*, 6:17–35.
- [10] Anton, A. I. (1997). *Goal Identification and Refinement in the Specification of Software-based Information Systems*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA. UMI Order No. GAX97-35409.
- [11] Apel, S. and Kästner, C. (2009). An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84.
- [12] Auer, K. and Miller, R. (2002). *Extreme Programming Applied: Playing to Win*. Addison-Wesley Boston.

- [13] Batool, A., Motla, Y. H., Hamid, B., Asghar, S., Riaz, M., Mukhtar, M., and Ahmed, M. (2013). Comparative study of traditional requirement engineering and agile requirement engineering. In *15th International Conference on Advanced Communication Technology (ICACT)*, pages 1006–1014.
- [14] Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional.
- [15] Beck, K. and Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2nd edition.
- [16] Beck, K. and Fowler, M. (2001). *Planning Extreme Programming*. Addison-Wesley Professional.
- [17] Bijan, Y., Yu, J., Stracener, J., and Woods, T. (2013). Systems requirements engineering—State of the methodology. *Systems Engineering*, 16(3):267–276.
- [18] Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72.
- [19] Boehm, B. W. (2000). Requirements that Handle IKIWISI, COTS, and Rapid Change. *IEEE Computer*, 33(7):99–102.
- [20] Booch, G. (2004). *Object-Oriented Analysis and Design with Applications*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 3rd edition.
- [21] Bosch, J. (2000). *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.
- [22] Bourque, P. and Fairley, R. E. (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition.
- [23] Bruegge, B. and Dutoit, A. A. (1999). *Object-Oriented Software Engineering; Conquering Complex and Changing Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- [24] C3 Team (1998). Case study: Chrysler goes to “extremes”. *Distributed Computing*.
- [25] CalCentral (2013a). *CalCentral Team, CalCentral User Stories*. [online] <https://confluence.media.berkeley.edu/confluence/display/MYB/CalCentral+User+Stories>.
- [26] CalCentral (2013b). *CalCentral Team, CalCentral’s Mission Statement*. [online] <https://confluence.media.berkeley.edu/confluence/display/MYB/Mission+Statement>.
- [27] Cardinal, M. (2013). *Executable Specifications with Scrum: A Practical Guide to Agile Requirements Discovery*. Addison-Wesley Professional, 1st edition.
- [28] Carroll, J. M. (1994). Making Use: A Design Representation. *Commun. ACM*, 37(12):28–35.

-
- [29] Castro, J., Kolp, M., and Mylopoulos, J. (2002). Towards requirements-driven information systems engineering: the *Tropos* project. *Inf. Syst.*, 27(6):365–389.
- [30] Cervone, H. F. (2011). Understanding agile project management methods using Scrum. *OCLC Systems & Services*, 27(1):18–22.
- [31] Chemuturi, M. (2012). *Requirements Engineering and Management for Software Development Projects*. Springer Publishing Company, Incorporated.
- [32] Cheng, B. H. C. and Atlee, J. M. (2007). Research Directions in Requirements Engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 285–303, Washington, DC, USA. IEEE Computer Society.
- [33] Cleland-Huang, J., Czauderna, A., and Mirakhorli, M. (2014). Chapter 4 - driving architectural design and preservation from a persona perspective in agile projects. In Babar, M. A., , Brown, A. W., , and Mistrik, I., editors, *Agile Software Architecture*, pages 83 – 111. Morgan Kaufmann, Boston.
- [34] Cockburn, A. (2004). *Crystal Clear a Human-powered Methodology for Small Teams*. Addison-Wesley Professional, 1st edition.
- [35] Cohen, D., Lindvall, M., and Costa, P. (2004). An introduction to agile methods. *Advances in Computers*, 62:1–66.
- [36] Cohn, M. (2004). *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.
- [37] Cohn, M. (2005). *Agile Estimating and Planning*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- [38] Cohn, M. (2008). Advantages of the “As a user, I want” user story template. [online] <http://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template>.
- [39] Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, 1st edition.
- [40] Constantine, L. L. and Lockwood, L. A. D. (1999). *Software for Use: A Practical Guide to the Models and Methods of Usage-centered Design*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.
- [41] Cooper, A. (2004). *The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity (2Nd Edition)*. Pearson Higher Education.
- [42] Dalpiaz, F. (2015). Teaching goal modeling in undergraduate education. In Horkoff, J., Lockerbie, J., Franch, X., Yu, E. S. K., and Mylopoulos, J., editors, *Proceedings of the 1st International iStar Teaching Workshop co-located with the 27th International Conference on Advanced Information Systems Engineering (CAiSE 2015), Stockholm, Sweden, June 9, 2015.*, volume 1370 of *CEUR Workshop Proceedings*, pages 1–6. CEUR-WS.org.
- [43] Denger, C., Berry, D. M., and Kamsties, E. (2003). Higher Quality Requirements Specifications through Natural Language patterns. In *2003 IEEE International Conference on Software - Science, Technology and Engineering (SwSTE 2003), 4-5 November 2003, Herzelia, Israel*, page 80. IEEE Computer Society.

- [44] Dennis, A., Wixom, B. H., and Tegarden, D. (2015). *Systems Analysis and Design: An Object-Oriented Approach with UML*. Wiley Publishing, 5th edition.
- [45] Dennis, L. A., Farwer, B., Bordini, R. H., Fisher, M., and Wooldridge, M. (2007). A Common Semantic Basis for BDI Languages. In Dastani, M., Fallah-Seghrouchni, A. E., Ricci, A., and Winikoff, M., editors, *Programming Multi-Agent Systems, 5th International Workshop, ProMAS 2007, Honolulu, HI, USA, May 15, 2007, Revised and Invited Papers*, volume 4908 of *Lecture Notes in Computer Science*, pages 124–139. Springer.
- [46] Descartes (2016). The Descartes Architect CASE-Tool. [online] <http://www.isys.ucl.ac.be/descartes/>.
- [47] Diaper, D. and Stanton, N. (2004). *The handbook of task analysis for human-computer interaction*. CRC Press.
- [48] Dimitrijević, S., Jovanović, J., and Devedžić, V. (2015). A comparative study of software tools for user story management. *Information & Software Technology*, 57:352–368.
- [49] Dingsøyr, T., Dybå, T., and Moe, N. B. (2010). Agile Software Development: An Introduction and Overview. In *Agile Software Development*, pages 1–13. Springer.
- [50] Duursma, C., Olsson, O., and Ulf, S. (1993). Task Model definition and Task Analysis process. *ESPRIT Project P5248 KADS-II CK-VUB-04, Vrije Universiteit Brussel*.
- [51] Eberlein, A. and Leite, J. (2002). Agile requirements definition: A view from requirements engineering. In *Proceedings of the International Workshop on Time-Constrained Requirements Engineering (TCRE’02)*, pages 4–8.
- [52] Elshandidy, H. and Mazen, H. (2013). Agile and Traditional Requirements Engineering: A Survey. *International Journal of Scientific and Engineering Research*, 4(9).
- [53] Faulkner, S. and Kolp, M. (2003). Towards an agent architectural description language for information systems. In *ICEIS 2003, Proceedings of the 5th International Conference on Enterprise Information Systems, Angers, France, April 22-26, 2003*, pages 59–66.
- [54] Firesmith, D. (2005). Are Your Requirements Complete? *Journal of Object Technology*, 4(1):27–44.
- [55] Forsberg, K. and Mooz, H. (1991). The Relationship of System Engineering to the Project Cycle. In *INCOSE International Symposium*, pages 57–65. Wiley Online Library.
- [56] Gibbs, R. D. (2006). *Project Management with the IBM®Rational Unified Process®: Lessons From The Trenches*. IBM Press.
- [57] Giorgini, P., Kolp, M., and Mylopoulos, J. (2002). Multi-agent and software architectures: A comparative case study. In Giunchiglia, F., Odell, J., and Weiß, G., editors, *Agent-Oriented Software Engineering III, Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions*, volume 2585 of *Lecture Notes in Computer Science*, pages 101–112. Springer.

-
- [58] Glinz, M. (2012). *A Glossary of Requirements Engineering Terminology, Version 1.4*. [online] <https://www.ireb.org/en/downloads>.
- [59] Golick, J. (2010). The Problem with User Stories. [online] <http://jamesgolick.com/2010/1/4/the-problem-with-user-stories.html>.
- [60] Gomaa, H. (2004). Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures.
- [61] Gunal, V. (2012). Agile Software Development Approaches and Their History. *Enterprise Software Engineering*. [online] http://sewiki.iai.uni-bonn.de/_media/teaching/labs/xp/2012b/seminar/1-agile.pdf.
- [62] Hastie, S. and Wick, A. (2014). User Stories and Use Case - Don't Use Both! [online] <http://www.batimes.com/articles/user-stories-and-use-cases-dont-use-both.html>.
- [63] Hastie, S. and Wojewoda, S. (2015). Standish Group 2015 Chaos Report-Q&A with Jennifer Lynch. [online] <https://www.infoq.com/articles/standish-chaos-2015>.
- [64] Hazzan, O. and Dubinsky, Y. (2009). *Agile Software Engineering*. Springer Science & Business Media.
- [65] Hsia, P., Davis, A. M., and Kung, D. C. (1993). Status Report: Requirements Engineering. *IEEE Software*, 10(6):75–79.
- [66] Hudson, W. (2013). User stories don't help users: introducing persona stories. *interactions*, 20(6):50–53.
- [67] Hull, M. E. C., Jackson, K., and Dick, J., editors (2011). *Requirements Engineering*. Springer, 3rd edition.
- [68] Hunt, J. (2006). *Agile Software Construction*. Springer.
- [69] IBM (2007). *The Rational Unified Process, Version 7.0.1*.
- [70] IEEE Computer Society (1998). *IEEE Recommended Practice for Software Requirements Specifications*. Institute of Electrical and Electronics Engineers.
- [71] IEEE Std (1990). IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84.
- [72] ISO9000:2005 (2005). *Quality Management Systems—Fundamentals and Vocabulary*. International Organization for Standardization.
- [73] ISO/IEC/IEEE 24765:2010(E) (2010). Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418.
- [74] ISO/IEC/IEEE 29148:2011(E) (2011). ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes – Requirements engineering. *ISO/IEC/IEEE 29148:2011(E)*, pages 1–94.
- [75] Jacobson, I. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press; Addison-Wesley Pub, revised edition.

- [76] Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [77] Kalermo, J. and Rissanen, J. (2002). Agile software development in theory and practice. Master's thesis, University of Jyväskylä. [online] http://www.cs.jyu.fi/sb/Publications/KalermoRissanen_MastersThesis_060802.pdf.
- [78] Keith, C. (2010). *Agile Game Development with Scrum*. Addison-Wesley Professional, 1st edition.
- [79] Klugh, D. (2011). User Story Authorship: Defining the User Role. [online] <http://agilerepublic.com/?p=29>.
- [80] Kolp, M., Giorgini, P., and Mylopoulos, J. (2002). Organizational multi-agent architectures: a mobile robot example. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*, pages 94–95. ACM.
- [81] Kolp, M. and Mylopoulos, J. (2001). Software architectures as organizational structures. In *Proc. ASERC Workshop on "The Role of Software Architectures in the Construction, Evolution, and Reuse of Software Systems"*, Edmonton, Canada.
- [82] Kruchten, P. (2003). *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.
- [83] Laplante, P. A. (2013). *Requirements Engineering for Software and Systems*. CRC Press, 2nd edition.
- [84] Lapouchnian, A. (2005). Goal-oriented Requirements Engineering: An Overview of the Current Research. *University of Toronto*, page 32. [online] <http://www.cs.utoronto.ca/~alexei/pub/Lapouchnian-Depth.pdf>.
- [85] Larman, C. and Basili, V. R. (2003). Iterative and Incremental Development: A Brief History. *IEEE Computer*, 36(6):47–56.
- [86] Larman, C. and Vodde, B. (2010). *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*. Addison-Wesley Professional, 1st edition.
- [87] Leffingwell, D. (2011). *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley Professional, 1st edition.
- [88] Lieberman, H., Paternò, F., Klann, M., and Wulf, V. (2006). End-user development: An emerging paradigm. In Lieberman, H., Paternò, F., and Wulf, V., editors, *End User Development*, Human-Computer Interaction Series, pages 1–8. Springer.
- [89] Limbourg, Q. and Vanderdonckt, J. (2004). Comparing task models for user interface design. *The handbook of task analysis for human-computer interaction*, 6:135–154.
- [90] Lindstrom, L. and Jeffries, R. (2004). Extreme Programming and Agile Software Development Methodologies. *IS Management*, 21(3):41–52.

-
- [91] Liskin, O., Pham, R., Kiesling, S., and Schneider, K. (2014). Why We Need a Granularity Concept for User Stories. In *Agile Processes in Software Engineering and Extreme Programming - 15th International Conference, XP 2014, Rome, Italy, May 26-30, 2014. Proceedings*, pages 110–125.
- [92] Logue, K. and McDaid, K. (2008). Handling Uncertainty in Agile Requirement Prioritization and Scheduling Using Statistical Simulation. In Melnik, G., Kruchten, P., and Poppendieck, M., editors, *Agile Development Conference, AGILE 2008, Toronto, Canada, 4-8 August 2008*, pages 73–82. IEEE Computer Society.
- [93] Lucassen, G., Dalpiaz, F., van der Werf, J. M. E. M., and Brinkkemper, S. (2015). Forging high-quality User Stories: Towards a discipline for Agile Requirements. In Zowghi, D., Gervasi, V., and Amyot, D., editors, *23rd IEEE International Requirements Engineering Conference, RE 2015, Ottawa, ON, Canada, August 24-28, 2015*, pages 126–135. IEEE Computer Society.
- [94] Lucassen, G., Dalpiaz, F., van der Werf, J. M. E. M., and Brinkkemper, S. (2016). Visualizing user story requirements at multiple granularity levels via semantic relatedness. In Comyn-Wattiau, I., Tanaka, K., Song, I., Yamamoto, S., and Saeki, M., editors, *Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings*, volume 9974 of *Lecture Notes in Computer Science*, pages 463–478.
- [95] Lucia, A. D. and Qusef, A. (2010). Requirements Engineering in Agile Software Development. *Journal of Emerging Technologies in Web Intelligence*, 2(3):212–220.
- [96] Machado, R. J., Ramos, I., and Fernandes, J. M. (2005). Specification of Requirements Models. In *Engineering and managing software requirements*, pages 47–68. Springer.
- [97] Madhavji, N. H. (1991). The process cycle [software engineering]. *Software Engineering Journal*, 6(5):234–242.
- [98] ManifestoAgile (2001). Manifesto for Agile Software Development. [online] <http://www.agilemanifesto.org>.
- [99] Martin, J. (1991). *Rapid Application Development*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA.
- [100] Mellor, S. J. and Balcer, M. (2002). *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [101] Meyer, B. (1985). On formalism in specifications. *IEEE Softw.*, 2(1):6–26.
- [102] Mori, G., Paternò, F., and Santoro, C. (2002). CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. *IEEE Trans. Software Eng.*, 28(8):797–813.
- [103] Mouratidis, H., Kolp, M., Faulkner, S., and Giorgini, P. (2005). A secure architectural description language for agent systems. In Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M. P., and Wooldridge, M., editors, *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands*, pages 578–585. ACM.

- [104] Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.
- [105] Mylopoulos, J., Kolp, M., and Giorgini, P. (2002). Agent-oriented software development. In Vlahavas, I. P. and Spyropoulos, C. D., editors, *Methods and Applications of Artificial Intelligence, Second Hellenic Conference on AI, SETN 2002. Thessaloniki, Greece, April 11-12, 2002, Proceedings*, volume 2308 of *Lecture Notes in Computer Science*, pages 3–17. Springer.
- [106] Nawrocki, J. R., Ochodek, M., Jurkiewicz, J., Kopczynska, S., and Alchimowicz, B. (2014). Agile Requirements Engineering: A Research Perspective. In Geffert, V., Preneel, B., Rován, B., Stuller, J., and Tjoa, A. M., editors, *SOFSEM 2014: Theory and Practice of Computer Science - 40th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 26-29, 2014, Proceedings*, volume 8327 of *Lecture Notes in Computer Science*, pages 40–51. Springer.
- [107] North, D. (2012). WHAT’S IN A STORY? [online] <http://dannorth.net/whats-in-a-story/>.
- [108] Nuseibeh, B. and Easterbrook, S. (2000). Requirements Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE ’00, pages 35–46, New York, NY, USA. ACM.
- [109] OMG (2013). Business Process Model and Notation (BPMN). Version 2.0.1. Technical report, Object Management Group.
- [110] OMG (2015). OMG Unified Modeling LanguageTM (OMG UML). Version 2.5. Technical report, Object Management Group.
- [111] Oscar, S. (2013). *Visual Paradigm for UML*. Int’l Book Market Service Limited.
- [112] Paetsch, F., Eberlein, A., and Maurer, F. (2003). Requirements Engineering and Agile Software Development. In *Proceedings of the Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, WETICE ’03, pages 308–, Washington, DC, USA. IEEE Computer Society.
- [113] Palmer, S. R. and Felsing, M. (2001). *A Practical Guide to Feature-Driven Development*. Pearson Education, 1st edition.
- [114] Patel, C. and Ramachandran, M. (2009). Story Card Based Agile Software Development. *International Journal of Hybrid Information Technology*, 2(2):125–140.
- [115] Patel, C. and Ramachandran, M. (2010). Best Practices Guidelines for Agile Requirements Engineering Practices. In *Handbook of Research on Software Engineering and Productivity Technologies: Implications of Globalization.*, pages 1–14. IGI Global.
- [116] Patton, J. (2005a). Finding the forest in the trees. In Johnson, R. E. and Gabriel, R. P., editors, *OOPSLA Companion*, pages 266–274. ACM.
- [117] Patton, J. (2005b). It’s All in How You Slice. [online] http://jpattonassociates.com/wp-content/uploads/2015/01/how_you_slice_it.pdf.

-
- [118] Patton, J. and Economy, P. (2014). *User Story Mapping: Discover the Whole Story, Build the Right Product*. O'Reilly Media, Inc., 1st edition.
- [119] Pohl, K. (2010). *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Publishing Company, Incorporated, 1st edition.
- [120] Pohl, K. and Rupp, C. (2011). *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation Level - IREB Compliant*. Rocky Nook, 1st edition.
- [121] Pressman, R. S. (2010). *Software Engineering: A Practitioner's Approach*. Palgrave Macmillan, 7th edition.
- [122] Pressman, R. S. and Bruce, R. M. (2013). *Software Engineering: A Practitioner's Approach*. Palgrave Macmillan, 8th edition.
- [123] Ramesh, B., Cao, L., and Baskerville, R. (2010). Agile requirements engineering practices and challenges: an empirical study. *Inf. Syst. J.*, 20(5):449–480.
- [124] Robber, M., Lucassen, G., va der Werf, J. M. E., Dalpiaz, F., and Brinkkemper, S. (2016). *Automated Extraction of Conceptual Model from User Stories via NLP*. RE2016.
- [125] Rodríguez, P., Yagüe, A., Alarcón, P. P., and Garbajosa, J. (2009). Some findings concerning requirements in Agile methodologies. In *International Conference on Product-Focused Software Process Improvement*, pages 171–184. Springer.
- [126] Roman, G.-C. (1985). A taxonomy of current issues in requirements engineering. *Computer*, 18(4):14–23.
- [127] Royce, W. W. (1970). Managing the development of large software systems. In *proceedings of IEEE WESCON*, number 8, pages 328–388. Los Angeles.
- [128] Rubin, K. S. (2012). *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Professional, 1st edition.
- [129] Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language Reference Manual*. Pearson Higher Education, 2nd edition.
- [130] Saunders, M., Lewis, P., and Thornhill, A. (2009). *Research Methods for Business Students*. Pearson Education.
- [131] Schreiber, G. T. and Akkermans, H. (2000). *Knowledge Engineering and Management: The CommonKADS Methodology*. MIT Press, Cambridge, MA, USA.
- [132] Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*. Upper Saddle River, NJ, USA, 1st edition.
- [133] Shapiro, R., White, S., and Bock, C. (2011). *BPMN 2.0 Handbook Second Edition: Methods, Concepts, Case Studies and Standards in Business Process Management Notation*. Future Strategies Incorporated.
- [134] Shergill, M. P. K. and Scharff, C. (2012). Developing Multi-Channel Mobile Solutions for a Global Audience: The Case of a Smarter Energy Solution. *SARNOFF'12, New Jersey*.

- [135] Shuja, A. and Krebs, J. (2007). *IBM Rational Unified Process Reference and Certification Guide: Solution Designer*. IBM Press, 1st edition.
- [136] Silva, T. R., Hak, J., and Winckler, M. (2016). Testing prototypes and final user interfaces through an ontological perspective for behavior-driven development. In Bogdan, C., Gulliksen, J., Sauer, S., Forbrig, P., Winckler, M., Johnson, C. W., Palanque, P. A., Bernhaupt, R., and Kis, F., editors, *Human-Centered and Error-Resilient Systems Development - IFIP WG 13.2/13.5 Joint Working Conference 6th International Conference on Human-Centered Software Engineering, HCSE 2016, and 8th International Conference on Human Error, Safety, and System Development, HESSD 2016 Stockholm, Sweden, August 29-31, 2016, Proceedings*, volume 9856 of *Lecture Notes in Computer Science*, pages 86–107. Springer.
- [137] Sommerville, I. (2010). *Software Engineering*. Addison-Wesley, 9 edition.
- [138] Spivey, J. M. (1988). *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, New York, NY, USA.
- [139] Stapleton, J. (1997). *DSDM: Dynamic Systems Development Method: The Method in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [140] Sutherland, J. and Schwaber, K. (2011). The Scrum Papers: Nut, Bolts, and Origins of an Agile Framework. [online] <http://34slpa7u66f159hfp1fhl9aur1-wpengine.netdna-ssl.com/scrumpapers.pdf>.
- [141] Takeuchi, H. and Nonaka, I. (1986). The new new product development game. *Harvard business review*, 64(1):137–146.
- [142] Tsui, F., Karam, O., and Bernal, B. (2013). *Essentials Of Software Engineering*. Jones and Bartlett Publishers, Inc., USA, 3rd edition.
- [143] Vähäniitty, J. and Rautiainen, K. T. (2008). Towards a Conceptual Framework and Tool Support for Linking Long-term Product and Business Planning with Agile Software Development. In *Proceedings of the 1st International Workshop on Software Development Governance*, SDG '08, pages 25–28, New York, NY, USA. ACM.
- [144] Van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 249–, Washington, DC, USA. IEEE Computer Society.
- [145] Van Lamsweerde, A. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley Publishing, 1st edition.
- [146] Van Lamsweerde, A. and Letier, E. (2004). From object orientation to goal orientation: A paradigm shift for requirements engineering. In *Radical Innovations of Software and Systems Engineering in the Future*, pages 325–340. Springer.
- [147] Velghe, M. (2015). Requirements Engineering in Agile Methods: Contributions on User Story Models. Master’s thesis, KU Leuven, Belgium. [online] <http://www.isys.ucl.ac.be/descartes/ThesisMattijs.pdf>.

-
- [148] Verner, J., Cox, K., Bleistein, S., and Cerpa, N. (2005). Requirements Engineering and Software Project Success: an industrial survey in Australia and the U.S. *Australasian Journal of Information Systems*, 13(1).
 - [149] VersionOne (2016). The 10th Annual State of Agile Report. [online] <http://stateofagile.versionone.com/>.
 - [150] Vlaanderen, K., Jansen, S., Brinkkemper, S., and Jaspers, E. (2011). The agile requirements refinery: Applying SCRUM principles to software product management. *Information & Software Technology*, 53(1):58–70.
 - [151] Wake, B. (2003). Invest in Good Stories, and SMART Tasks. [online] <http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/>.
 - [152] Wang, X., Zhao, L., Wang, Y., and Sun, J. (2014). The Role of Requirements Engineering Practices in Agile Development: An Empirical Study. In Zowghi, D. and Jin, Z., editors, *Requirements Engineering: First Asia Pacific Requirements Engineering Symposium, APRES 2014, Auckland, New Zealand, April 28-29, 2014. Proceedings*, pages 195–209. Springer Berlin Heidelberg, Berlin, Heidelberg.
 - [153] Wautelet, Y., Heng, S., Hintea, D., Kolp, M., and Poelmans, S. (2016a). Bridging User Story Sets with the Use Case Model. In *3rd International Workshop on Conceptual Modeling in Requirements and Business Analysis, MReBA2016, Gifu, Japan, November 14-17, 2016*.
 - [154] Wautelet, Y., Heng, S., Kolp, M., and Mirbel, I. (2014). Unifying and Extending User Story Models. In *CAiSE 2014, Thessaloniki, Greece. Proc.*, volume 8484 of *LNCS*, pages 211–225. Springer.
 - [155] Wautelet, Y., Heng, S., Kolp, M., Mirbel, I., and Poelmans, S. (2016b). Building a Rationale Diagram for Evaluating User Story Sets. In *Tenth IEEE International Conference on Research Challenges in Information Science, RCIS 2016, Grenoble, France, June 1-3, 2016*, pages 1–12. IEEE.
 - [156] Wautelet, Y. and Kolp, M. (2013). Mapping i* within UML for Business Modeling. In Doerr, J. and Opdahl, A. L., editors, *Requirements Engineering: Foundation for Software Quality - 19th International Working Conference, REFSQ 2013, Essen, Germany, April 8-11, 2013. Proceedings*, volume 7830 of *Lecture Notes in Computer Science*, pages 237–252. Springer.
 - [157] Wautelet, Y. and Kolp, M. (2016). Business and model-driven development of BDI multi-agent systems. *Neurocomputing*, 182:304–321.
 - [158] Wautelet, Y., Kolp, M., and Poelmans, S. (2011). Requirements-Driven Iterative Project Planning. In Escalona, M. J., Cordeiro, J., and Shishkov, B., editors, *Software and Data Technologies - 6th International Conference, ICSOFT 2011, Seville, Spain, July 18-21, 2011. Revised Selected Papers*, volume 303 of *Communications in Computer and Information Science*, pages 121–135. Springer.
 - [159] Wautelet, Y., Schinckus, C., and Kolp, M. (2008). A Modern Epistemological Reading of Agent Orientation. *IJIIT*, 4(3):46–57.
 - [160] Wiegers, K. (2005). *More about software requirements: thorny issues and practical advice*. Microsoft Press.

- [161] Wiegers, K. and Beatty, J. (2013). *Software Requirements*. Microsoft, 3rd edition.
- [162] Williams, L. (2007). A Survey of Agile Development Methodologies. [online] <http://www.smaele.nl/documents/Williams-AgileMethods-2007.pdf>.
- [163] Yourdon, E. et al. (1989). *Modern Structured Analysis*, volume 191. Yourdon Press Englewood Cliffs, NJ.
- [164] Yu, E. S. K., Amyot, D., Mussbacher, G., Franch, X., and Castro, J. (2013). Practical applications of i* in industry: The state of the art. In *21st IEEE International Requirements Engineering Conference, RE 2013, Rio de Janeiro-RJ, Brazil, July 15-19, 2013*, pages 366–367. IEEE Computer Society.
- [165] Yu, E. S. K., Giorgini, P., Maiden, N., and Mylopoulos, J. (2011). *Social Modeling for Requirements Engineering*. The MIT Press.
- [166] Yu, E. S. K. and Mylopoulos, J. (1994). Understanding “Why”; in Software Process Modelling, Analysis, and Design. In *Proceedings of the 16th International Conference on Software Engineering, ICSE '94*, pages 159–168, Los Alamitos, CA, USA. IEEE Computer Society Press.
- [167] Zave, P. (1997). Classification of Research Efforts in Requirements engineering. *ACM Comput. Surv.*, 29(4):315–321.

Appendix A

List of Publications

International Journals

1. “Designing a MOOC as an Agent-Platform Aggregating Heterogeneous Virtual Learning Environments.” *Behaviour & Information Technology*, pp. 1-18. 2016. With Yves Wautelet, Manuel Kolp, Loris Penserini, and Stephan Poelmans. (**IF: 1.211**)

Referred Conference Publications (Full Papers)

1. “Building a Rationale Diagram for Evaluating User Story Sets.” In *IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, pp 1-12. 2016. With Yves Wautelet, Manuel Kolp, Isabelle Mirbel, and Poelmans Stephan. (**AR: 28.86%**)
2. “Unifying and extending user story models.” In *Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAiSE)*, pp. 211-225. Springer International Publishing, 2014. With Yves Wautelet, Manuel Kolp, and Isabelle Mirbel. (**AR: 18.10%**)
3. “A Usage-Based Unified Resource Model.” In *24th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 299-304. 2012. With Yves Wautelet, and Manuel Kolp. (**AR: 27.00%**)
4. “Stress Level on Global Software Projects using Waterfall and Scrum: A Preliminary Comparison.” In *1st Asian Conference on Information Systems (ACIS)*. 2012. Distinguished Paper Award. With Christelle Scharff, and Kulkarni Vidya. (**AR: Not provided**). **Distinguished paper award.**

Referred Conference Publications (Short Papers)

1. “Towards an Agent-driven Software Architecture Aligned with User Stories.” In *Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART)*, vol. 2, pp. 337-345. 2016. With Yves Wautelet, Manuel Kolp, and Christelle Scharff. (**AR: 45.00%**)

Referred Workshop Papers

1. “Bridging User Story Sets with the Use Case Model.” In 3rd International Workshop on Conceptual Modeling in Requirements and Business Analysis (MReBA). 2016. With Yves Wautelet, Diana Hintea, Manuel Kolp, and Poelmans Stephan. (**AR: 33.33%**)
2. “On the difficulties for students to adhere to scrum on global software development projects: preliminary results.” In Collaborative Teaching of Globally Distributed Software Development Workshop (CTGDSD), pp. 25-29. IEEE, 2012. With Christelle Scharff, and Kulkarni Vidya. (**AR: Not provided**)

Posters

1. “Perspectives on User Story Based Visual Transformations.” Joint Proceedings of REFSQ-2017 Workshops, Doctoral Symposium, Research Method Track, and Poster Track co-located with the 23rd International working conference on Requirements Engineering: Foundation for Software Quality (REFSQ). 2017. With Yves Wautelet, and Manuel Kolp. (**AR: Not provided**)
2. “An ontological basis for resource representation.” In 27th Annual ACM Symposium on Applied Computing (SAC), pp. 765-766. ACM, 2012. With Yves Wautelet, and Manuel Kolp. (**AR: Not provided**)

Appendix B

User Story Templates Dataset

B.1 Introduction

This document presents the results of searching for user story templates in formal and informal sources. The formal source refers to books, journal papers, conference papers and websites of key person involved in agile methods. We used scientific search engines such as ScienceDirect, Springer, IEEE Xplore and ACM Digital Library for doing search. We used the simple search option for every searching engine. In addition, we also used the Google Scholar to search for the formal sources. The informal source, on the other hand, refers to websites and blogs of agile practitioners; however, we do not consider forums of discussion.

For each source, we took 100 links or papers. We scrutinized each paper and link for user story template. We only considered the US templates that contain WHO, WHAT and WHY dimensions; other form of US templates were not taken. Table B.1 presents the keywords used for searching user story template for both sources. Our search were conducted by mid of 2013. Later search can produce different results.

Table B.1 Keywords for searching user story templates.

KW1: ‘User Story Template’
KW2: ‘User Story’ and ‘XP’
KW3: ‘User Story’ and ‘Extreme programming’
KW4: ‘User Story’ and ‘Agile’
KW5: ‘Agile Requirement’ and ‘User Story’

B.2 User Story Templates Set

The Tables B.2 and B.3 provides the final result of the US template found respectively in formal and informal source.

Table B.2 User story templates found in formal source.

N	User Story Templates
UST1	<i>I as a <role> I want <function> so that <business value></i> e.g. As a creator, I want to upload a video so that any users can view it. From Cohn [36].
UST2	<i>As a <type of user> I want <capability> so that <business value></i> e.g. As a book buyer, I want to search for a book by ISBN so that I can find the right book quickly. From Cohn [37]
UST3	<i>As a <type of user>, I want <some goal> so that <some reason></i> e.g. As a moderator, I want to create a new game by entering a name and an optional description so that I can start inviting estimators. From Cohn [39]
UST4	<i>As a <user role>, I want <goal> [so that <reason>].</i> e.g. As a player, I want a player mute button so that I stop being distracted by some of the other players online. From [78].
UST5	<i>As a <role>, I can <activity> so that <business value>.</i> e.g. As a Consumer, I want to be able to see my daily energy usage so that I can lower my energy costs and usage. From Leffingwell [87].
UST6	<i>As a <type of user>, I want <some particular feature> so that <some benefit is received>.</i> e.g. As a bank customer, I want to view my current account balance so that I know my recent deposit went through. Patton, J. (2005) It's all in how you slice, better software, www.stickyminds.com.
UST7	<i>As a <type of user> I want to <perform some task> so that I can <achieve some goal>.</i> e.g. As a harried shopper I want to locate a CD in the store so that I can purchase it quickly, leave, and continue with my day. Patton, J. (2007). From User Story to User Interface, Tutorial Handouts for Agile 2007. www.agileproductdesign.com.
UST8	<i>As a <role> I need <feature> So that <value></i> e.g. As an employee I need to read the company news so that I am up to date on what is happening in my company. Tom J. Bang. An agile approach to requirement specification. In Giulio Concas, Ernesto Damiani, Marco Scotto, and Giancarlo Succi, editors, Agile Processes in Software Engineering and Extreme Programming, volume 4536 of Lecture Notes in Computer Science, pages 193–197. Springer Berlin Heidelberg, 2007.
UST9	<i>As <role> I would like to <action> so that <benefit>.</i>

	e.g. As a Job seeker I would like to register so that I can make my data available to headhunters and use the system capabilities reserved for registered job seekers. Miranda, E., Bourque, P., and Abran A. (2009). <i>Sizing user stories using paired comparisons</i>. Inf. Softw. Technol., 51(9):1327–1337.
UST10	<i>As a <role>, I want <behaviour> so that <benefit>.</i> e.g. No example. Sharp, H., Robinson, H., and Petre, M. (2009). <i>The role of physical artefacts in agile software development: Two complementary perspectives</i>. <i>Interacting with Computers</i>, 21(12):108 – 116.
UST11	<i>As a <role>, I can <action>, so that <goal>.</i> e.g. No example Cohn, M., Sim, S., and Lee P. C. (2009). <i>What counts as software process? negotiating the boundary of software work through artifacts and conversation</i>. <i>Computer Supported Cooperative Work (CSCW)</i>, 18:401–443.
UST12	<i>As a <role> I want to <action> so that <result>.</i> e.g. As an online customer I want to enter a product name so that I can view details of that product. OhEocha, C. and Conboy, K. (2010). <i>The role of the user story agile practice in innovation</i>. In Pekka Abrahamsson and Nilay Oza, editors, <i>Lean Enterprise Software and Systems</i>, volume 65 of <i>Lecture Notes in Business Information Processing</i>, pages 20–30. Springer Berlin Heidelberg.
UST13	<i>As a <user type>, I want to <feature or functionality>, so that <value or expected benefit>.</i> e.g. As a library user, I want to search for books by author, with speed and ease of use, so that I can find all books of the same author. COSMIC (2011). <i>Guideline for the use of COSMIC FSM to manage Agile projects</i>. VERSION 1.0. www.cosmicon.com.
UST14	<i>As a <type of user>, I want <some action> so that <business benefit>.</i> e.g. As a Customer I want to be able to add products to my basket So that I can continue shopping. Blankenship, J., Bussa, M., and Millett, S. (2011). <i>extreme programming</i>. In <i>Pro Agile .NET Development with Scrum</i>, pages 29–51.
UST15	<i>As a <type of user> I want <capability or feature> so that <business value or benefit>.</i> e.g. As a faculty, I want to select the names of the students from the database so that I can make them as a batch of some particular class which I am going to handle. Kavitha C.R and Sunitha Mary Thomas (2011). <i>Requirement gathering for small projects using agile methods</i>. <i>IJCA Special Issue on Computational Science - New Dimensions and Perspectives</i>, (3):122–128. Published by Foundation of Computer Science.

UST16	<i>As a <role> I want <something> so that <benefit>.</i> e.g. As a student I want to purchase a parking pass so that can drive to school. Vinicius Pereira and AntonioFrancisco Prado (2011). <i>Introducing a new agile development for web applications using a groupware as example</i>. In Jr. Hruschka, EstevamRafael, Junzo Watada, and Maria Carmo Nicoletti, editors, Integrated Computing Technology, volume 165 of Communications in Computer and Information Science, pages 144–160. Springer Berlin Heidelberg.
UST17	<i>As a <role> I want <something> so that <benefit>.</i> e.g. As a player, I would like to change my class at any time during an online game. Victor Schetinger, Cesar Souza, Lisandra Manzoni Fontoura, and Cesar Tadeu Pozzer (2011). <i>User stories as actives for game development</i>, proceedings of sbgames.
UST18	<i>As a <role>, I want <some goal> so that <some reason>.</i> e.g. As a string manipulation library user, I want to have a fancy case method in order to gain fancy cased strings. Mathias Landhausser and Adrian Genaid, (2012). <i>Connecting user stories and code for test development</i>. In Recommendation Systems for Software Engineering (RSSE), Third International Workshop on, pages 33 –37.
UST19	<i>As a <role>, I want to <capability> so that <goal>.</i> e.g. As a bank customer, I want to be able to view all relevant information about my account balance so that I can make impulse purchases using my bank card. Neil Maiden, (2012). <i>Exactly how are requirements written?</i> Software, IEEE, 29(1):26 –27.
UST20	<i>As a <user type>, I want to <goal> So That <reason>.</i> e.g. As a Purchase head I want packaging authority to process packaging items extremely efficiently, accurately, cost effectively and satisfactorily within a week as per the strict adherence to prescribed suppliers list, purchase rules and quality assessment rules and provide packaging report. Vibha Gaur and Anuja Soni, (2012). <i>A novel approach to explore inter agent dependencies from user requirements</i>. Procedia Technology, 1(0):412 – 419. First World Conference on Innovation and Computer Sciences.

Table B.3 User story templates found in informal source.

N	User Story Templates
UST21	<i>As a <person in a role> I want to <perform some activity> so that <some goal is achieved> .</i> e.g. No example.

	Yunyun Zhu. Requirements Engineering in an Agile Environment. Master's Thesis, Uppsala University, June 2009.
UST22	<i>As a <role> the user wants to <function> so that <rationale>.</i> e.g. No example. Yunyun Zhu. Requirements Engineering in an Agile Environment. Master's Thesis, Uppsala University, June 2009.
UST23	<i>As a <type of user> I want <capability> so that <business value>.</i> e.g. No example. Anna Georgsson. Introducing Story Points and User Stories to Perform Estimations in a Software Development Organisation. A case study at Swedbank IT. Master's Thesis, UME UNIVERSITY, January 2011.
UST24	<i>As a <type of user> I want to <do something> so that <I get some benefit>.</i> e.g. No example. Autho's blog: http://jpattonassociates.com/downloads/quickrefs/patton_story_essentials_quickref.pdf
UST25	<i>As a <type of user> I want to <perform some action> so that I can <reach some goal>.</i> e.g. No example. Autho's blog: http://jpattonassociates.com/downloads/quickrefs/patton_agile_11x17.pdf
UST26	<i>As a <role> I want <something> so that <benefit>.</i> e.g. As a student I want to purchase a parking pass so that I can drive to school. Autho's blog: http://www.agilemodeling.com/artifacts/userStory.htm
UST27	<i>As a X I want Y so that Z.</i> e.g. As a Payroll Administrator I want the system to compute and track Social Security tax deductions So that we can reduce the paperwork and tracking, and avoid accidentally deducting too much. http://c2.com/cgi/wiki?UserStoryTemplate
UST28	<i>As a ... I want ... so that ...</i> No example. http://ronjeffries.com/xprog/blog/how-should-user-stories-be-written/
UST29	<i>As a <system role>, I can <do something> so that <reason/value>.</i> No example. https://intranet.5amsolutions.com/display/process/User+Story
UST30	<i>As a(n) <actor> I would like to <description> so that <outcome>.</i> No example. http://stlouis.iiba.org/index.php/resources/template
UST31	<i>As a <role/persona> I want <outcome> so that <value>.</i> e.g. As a Regular User I want a Print Function So that I can keep a hard copy of my data.

	http://www.sprint0.com/ft3-peak-performance-articles/30-1-introduction-to-user-stories
UST32	<i>As a <User> I want to <do something> to <achieve business goal>.</i> e.g. As an Account Holder I want to withdraw cash from an ATM So that I can get money when the bank is closed. http://agile.dzone.com/news/bdd-holy-grail-user-story
UST33	<i>As a <user type> want <an interaction+outcome> so that <I get some form of value>.</i> No example. http://www.betterprojects.net/2012/02/what-is-user-story.html
UST34	<i>As a <user> I can <do something> to <achieve some benefit>.</i> No example. http://www.bridging-the-gap.com/moving-from-an-epic-to-a-user-story-in-an-agile-product-backlog/
UST35	<i>As a <role>, I want to <do something> so that <reason/benefit>.</i> No example. http://www.industriallogic.com/blog/as-a-developer-is-not-a-user-story/
UST36	<i>As a <role> I want <something> so that <benefit>.</i> e.g. As a student I want to purchase a parking pass so that I can drive to school. http://www.agilemodeling.com/artifacts/userStory.htm
UST37	<i>As a <role>, I want to <do something> <with some frequency> so that I can/will <achieve some goal>.</i> e.g. As a user I want to download music to my media player daily so that I can listen to my favorite song when I chose. https://www.ibm.com/developerworks/mydeveloperworks/blogs/c914709e-8097-4537-92ef-8982fc416138/entry/agile_in_practices_user_stories_explained2?lang=en
UST38	<i>As a <Type of User> I want <Something> So that <I get some value>.</i> No example. http://submit2011.agilealliance.org/node/8967
UST39	<i>As a <user> I can <do something> so that <user value received>.</i> No example. ://www.slideshare.net/JEMILOD/scaling-agile-requirements-from-user-stories-to-agile-portfolio-management-10111787
UST40	<i>As a <user type>, I want to <function> so that <benefit></i> e.g. As a delivery team member, I want to know which tasks I own so that I can decide what to work on now. http://www.rallydev.com/help/writing-great-user-story
UST41	<i>As a <user>, I want <function>, so that <value>.</i> e.g. As a Creator, I want to upload a video so that any users can view it.

	http://www.subcide.com/articles/how-to-write-meaningful-user-stories/
UST42	<i>As a <user> I want <function> so that<value>.</i> No example. http://www.agileforall.com/2009/05/14/new-to-agile-invest-in-good-user-stories/
UST43	<i>As a <customer role> I want <functionality> because <customer value explanation>.</i> <i>In order to <value to achieve> as a <customer role> I want <some functionality>.</i> No example. http://softwaredevelopmenttoday.blogspot.be/2008/06/better-format-for-user-storiesuser.html
UST44	<i>As a <user role>, I want to <function> so that <benefit>, unless <exception>.</i> e.g. As a student, I want to register for a class...”; as opposed to “As an AHCI driver, I want to read a sequential stream of data... http://agilerepublic.com/?p=29
UST45	<i>As a <User or Role> I want <Business Functionality> so that <Business Justification></i> As a e.g. Account Holder I want Mobile Payments so that I can pay by phone. http://www.slideshare.net/zenpdm/introduction-to-user-stories-for-agile-product-development
UST46	<i>As a <user type>, I want to <function> so that I can <Business value>.</i> No example. http://www.slideshare.net/petersaddington/agile-and-user-story-workshop-peter-saddington
UST47	<i>As a <Type of User>, <Function to Perform> so that <Business Value>.</i> e.g. As a sales person, I want to add a new contact so that I can follow up later with prospects. http://www.extremeplanner.com/resources/Agile-Requirements.html
UST48	<i>As a <role> I want <feature> so that <benefit>.</i> e.g. As an Account Holder I want to withdraw cash from an ATM So that I can get money when the bank is closed. http://dannorth.net/whats-in-a-story/
UST49	<i>s a <role>, I can <feature> so that <reason>.</i> e.g. As a account owner, I can check my balance online so that I can keep a daily balance 24 hours a day. http://codesqueeze.com/the-easy-way-to-writing-good-user-stories/
UST50	<i>As a <role> I want <feature> so that <benefit>.</i> No example.

	http://mike2.openmethodology.org/wiki/User_Stories_Deliverable_Template
UST51	<i>As a <role> I want <feature> so that <business objective>.</i> e.g. As a bank customer I want to withdraw money from an ATM So that I'm not constrained by opening hours or lines at the teller's. http://guide.agilealliance.org/guide/stories.html
UST52	<i>As a <role>, I want <feature> so that <reason>.</i> e.g. As a user, I want to upload photos so that I can share photos with others. http://searchsoftwarequality.techtarget.com/definition/user-story
UST53	<i>As a <Role Name> I want <a feature> so that <some value delivered></i> No example. http://www.theagileleader.com/2012/01/the-3-cs-of-user-stories/
UST54	<i>As a <type of user>, I want <some feature>, so that <some goal>.</i> No example. http://blogs.adobe.com/agile/2012/06/20/does-every-item-in-the-product-backlog-require-a-user-story/
UST55	<i>As a <role>, I want <goal/desire> so that <benefit></i> <i>In order to <receive benefit> as a <role>, I want <goal/desire></i> e.g. As a user, I want to search for my customers by their first and last names. http://en.wikipedia.org/wiki/User_story
UST56	<i>As a <type of user>, I want <some goal> so that <some reason>.</i> No example. http://onproductmanagement.net/2012/08/17/user-stories-that-developers-can-actually-work-with-2/
UST57	<i>As a <end user role>, I want <the desire> so that <the rationale></i> e.g. As a PC user, I want a calculator with basic functionality on my PC so that I can conveniently perform basic mathematics operations. http://scrummethodology.com/scrum-user-stories/
UST58	<i>As a <insert role here> I want <goal or desire> so that <benefit></i> No example. http://pages.managementconcepts.com/UserStoryTemplate/
UST59	<i>As a <user role> I want to <goal> so I can <reason></i> e.g. As a registered user I want to log in so I can access subscriber-only content. https://sites.google.com/site/alensit/project-management/methodologies/scrum/user-stories/user-story-template
UST60	<i>As a <type of user>, I want <some goal> so that <some reason></i> e.g. As a user, I can pick which order show up on my home page so I can track their progress.

	http://blogs.msdn.com/b/aaronbjork/archive/2010/04/19/msf-agile-5-0-tip-2-simple-user-story-titles.aspx?Redirected=true
UST61	<i>As a <user role> I want <goal> so that <business value></i> No example. http://www.methodsandtools.com/archive/archive.php?id=113
UST62	<i>As a <User Role> I can <goal> so That <Business Value>.</i> No example. http://www.agile-ux.com/tag/user-stories/
UST63	<i>As a <type of user>, I want <some goal> so that <some reason>.</i> No example. http://www.agilehelpline.com/2011/03/user-stories-backlog-management.html
UST64	<i>As a <user>, I want <goal> so that <reason></i> No example. http://blogs.globallogic.com/simplifying-agile-the-art-of-writing-user-stories
UST65	<i>As a <user role> I want to <goal> so I can <reason></i> e.g. As a provider search user, I need the ability to search for providers by speciality so that I can more efficiently refer patients to specialists. http://www.scrumalliance.org/articles/169-new-to-user-stories
UST66	<i>As a <user role> I want to <goal> so I can <reason></i> As an ATM user I want to withdraw funds from my bank account so I can increase my cash on hand. http://newjersey.iiba.org/index.php/linksdownloads
UST67	<i>As a <type of user> I want to <goal> so that <reason>.</i> e.g. As a cyclist I want to switch gears so that I can go faster. http://www.slideshare.net/bartvermijlen/user-stories-develop-better-products-faster-and-cheaper
UST68	<i>As a <role>, I want <goal/desire> so that <benefit></i> No example. http://sprintometer.com/node/112
UST69	<i>As a <user role>, I want to <goal>, so I can <reason></i> e.g. As a job seeker, I want to search for a job, so I can advance my career. http://www.allaboutagile.com/user-stories/
UST70	<i>As a <user role> I want to <goal> so that <benefit></i> e.g. As an administrator I want to have centralised configuration so I can remotely change settings across all units. http://techportal.inviqa.com/2011/07/19/how-to-create-user-stories/
UST71	<i>As a <role>, I want <goal/desire></i> e.g. As a user, I want to search for my customers by their first and last names. http://www.csce.uark.edu/mquang/courses/3513/f2012/lectures/SE_Lecture_4.pdf
UST72	<i>As a <Role> I want to <goal> so I can <reason></i>

	e.g. As a registered user I want to log in so I can access subscriber-only content. http://www.slideshare.net/rsrivastava91/introducing-agile-user-stories
UST73	<i>As a <Role> I want to <goal> so I can <reason></i> e.g. As a registered user I want to log in so I can access subscriber-only content. http://www.slideshare.net/rsrivastava91/introducing-agile-user-stories
UST74	<i>As a <role> I want to <action> so that <benefits></i> No example. http://scrumpad.wpengine.com/features/user-stories
UST75	<i>An an <actor> I want <action> so that <achievement>.</i> e.g. As a Flickr member I want to be able to assign different privacy levels to my photos so I can control who I share which photos with. http://www.boost.co.nz/blog/agile/user-stories/
UST76	<i>As an <role> I want to <action> so that <achievement>.</i> e.g. As a customer I want to see the most popular blu-ray discs sold so that I can order one or more of them. http://breathingtech.com/2009/writing-user-stories-for-agile-scrum-projects/
UST77	<i>As a <user/actor/role> I want to <task/activity> so that <desired outcome>.</i> No example. http://sw-analyst.com/general/user-story-use-case-template/
UST78	<i>As a <user type> I want to <do some action> so that <desired result></i> e.g. As a wiki user I want a tools menu on the edit screen so that I can easily apply font formatting. https://en.wikipedia.org/wiki/Scrum_(software_development)
UST79	<i>As an <actor>, I want <action> so that <achievement/benefit></i> No example. http://www.csce.uark.edu/mqhuang/courses/3513/f2012/lectures/SE_Lecture_4.pdf
UST80	<i>As a <type of user> I want to <perform some task> so that I can <achieve some goal></i> e.g. As a harried shopper I want to locate a CD in the store so that I can purchase it quickly, leave, and continue with my day. http://www.agileproductdesign.com/downloads/patton_user_story_to_ui_handouts.pdf
UST81	<i>As an <actor> I want <action> so that <achievement></i> No example. http://www.boostagile.com/blog/user-stories-part-2-acceptance-criteria
UST82	<i>As an <actor> I want <action> so that <achievement></i>

	e.g. As a Flickr member, I want to set different privacy levels on my photos, so I can control who sees which of my photos with. http://www.boost.co.nz/blog/agile/use-cases-or-user-stories/
UST83	<i>As a <User or role> I want <Business Functionality> so that <Business Justification></i> As a Account Holder, I want to be able to withdraw funds from my checking account, So that I can buy some bling. http://minerva-group.com/downloads/Scrum-UserStories.pdf
UST84	<i>As a <user> I want <business functionality> so that <business value></i> No example. http://scrumcoaching.wordpress.com/2010/10/09/breaking-down-user-stories/
UST85	<i>As a <product user> I want <what> so that <users benefit></i> e.g. As a salesman I want car to be equipped with GPS so that I can easily set my direction. http://looksok.wordpress.com/2012/03/31/scrum-user-stories/

B.3 Summary of User Story Templates' Elements

In general, we found that user story templates are structured as following form:
As a [WHO], I want [WHAT] so that [WHY].

Table B.4 summaries number of occurrence of terminologies found in each dimension of user story template in the form of 'Formal + Informal'.

Table B.4 Syntax used in user story template

WHO	WHAT	WHY
Role (13+31)	Goal (4+18)	Business Value (7+18)
Type of User (8+15)	Something (3+10)	Benefit (7+18)
User (0+10)	Action (4+7)	Reason (4+14)
Actor (0+6)	Feature (4+7)	Goal (3+6)
System Role (0+1)	Function (1+7)	Achievement (0+4)
Persona (0+1)	Desire (0+6)	Rationale (0+2)
"x" (0+1)	Functionality(1+4)	Desire (0+2)
	Capability (3+1)	Outcome (0+1)
	Task (1+2)	Result (0+1)
	Activity (1+2)	"z" (0+1)
	Outcome (0+2)	
	Behaviour (0+1)	
	Description (0+1)	
	What (0+1)	
	"y" (0+1)	

Note: Business Value (5 + 9) = Value (2 + 9) 'Something' is not counted – it is too general.

Appendix C

Descriptive_Concept’s Definitions

This document provides different definitions of selected Descriptive_Concept we found in i*, KAOS, BPMN, IREB Glossary, IEEE and UML. Table C.1 exposes the selected Descriptive_Concept after the syntax selection process. The definition in **bold** is the selected definition ones for each Descriptive_Concept.

Table C.1 Selected syntaxes of Descriptive_Concept

WHO Dimension	WHAT Dimension	WHY Dimension
Role (13 + 31)	Goal (4 + 18)	Goal (3 + 6)
User (0 + 10)	Feature (4 + 7)	
Actor (0 + 6)	Functionality (1 + 4)	
	Capability (3 + 1)	
	Task (1 + 2)	
	Activity (1 + 2)	

Table C.2 Definitions of Role.

Source	Definition
i* [165]	“A role is an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor.”
KAOS [145]	“All stakeholder roles, including the customer and user roles, can be reduced to one single role.”
IREB Glossary [58]	N/A
UML [20]	“The purpose or capacity wherein one class or object participates in a relationship with another; the role of an object denotes the selection of a set of behaviours that are well-defined at a single point in time; a role is the face an object presents to the world at a given moment.”
BPMN	N/A
CommonKADS	N/A

Table C.3 Definitions of User.

Source	Definition
i* [165]	N/A
KAOS [145]	N/A
IREB Glossary [58]	“A person who uses the functionality provided by a system. Also called end user.”
UML [76]	“The term user refers not only to human users but to other systems. In this sense, the term user represents someone or something (such as another system outside the proposed system) that interacts with the system being developed.”
BPMN	N/A
CommonKADS	N/A

Table C.4 Definitions of Actor.

Source	Definition
i* [165]	“An actor is an active entity that carries out actions to achieve goals by exercising its know-how.”
KAOS [145]	N/A
IREB Glossary [58]	N/A
UML [76]	N/A
BPMN	N/A
CommonKADS	N/A

Table C.5 Definitions of Goal.

Source	Definition
i* [165]	“A hard-goal is a condition or state of affairs in the world that the stakeholders would like to achieve. In general, how the goal is to be achieved is not specified, allowing alternatives to be considered.” “A soft-goal is a condition or state of affairs in the world that the actor would like to achieve. But unlike a hard-goal, there are no clear-cut criteria for whether the condition is achieved, and it is up to the developer to judge whether a particular state of affairs in fact achieves sufficiently the stated soft-goal.”

KAOS [145]	“A goal is a prescriptive statement of intent that the system should satisfy through the cooperation of its agents.” “Behavioural goals describe intended system behaviours declaratively. A behavioural goal can be established in a clear-cut sense.” “Soft-goals prescribe preferences among alternative system behaviours. A Soft-goal cannot be established in a clear-cut sense.”
IREB Glossary [58]	“A desired state of affairs (that a stakeholder wants to achieve).” “Goals describe intentions of stakeholders. They may conflict with one another.”
UML [76]	N/A
BPMN	N/A
CommonKADS	“Goal and Value describe the goal of the task and the value that is execution adds to the process this task is a part of.”

Table C.6 Definitions of Feature

Source	Definition
i* [165]	N/A
KAOS [145]	“Units of functionality are sometime called feature in some problem worlds [...]” “The term feature is sometimes used to refer to a change unit.”
IREB Glossary [58]	“A delimitable characteristic of a system that provides value for stakeholders.” “Normally comprises several requirements and is used for communicating with stakeholders on a higher level of abstraction and for expressing variable or optional characteristics.”
UML [129]	“A property, such as operation or attribute, which is encapsulated as part of a list within a classifier, such as an interface, a class, or a datatype.”
BPMN	
CommonKADS	
IEEE Std 830-1998, [70]	“A feature is an externally desired service by the system that may require a sequence of inputs to effect the desired result. For example, in a telephone system, features include local call, call forwarding, and conference call. Each feature is generally described in a sequence of stimulus-response pairs.”

Table C.7 Definitions of Functionality.

Source	Definition
i* [165]	N/A
KAOS [145]	N/A
IREB Glossary [58]	“The capabilities of a system as stated by its functional requirements.”
UML	N/A
BPMN	
CommonKADS	
ISO 9126, [72] (take it from [121] p.513)	“Functionality. The degree to which the software satisfies stated needs as indicated by the following subattributes: suitability, accuracy, interoperability, compliance, and security.”

Table C.8 Definitions of Capability.

Source	Definition
i* [165]	“A capability represents the ability of an actor to define, choose, and execute a plan for the fulfilment of a goal, given certain world conditions and in the presence of a specific event.”
KAOS [145]	“The capabilities of an agent are defined statically in terms of monitoring links and control links to objects in the object model. Such links capture the agent’s ability to monitor or control object attributes or associations.”
IREB Glossary [58]	N/A
UML	N/A
BPMN	
N/A	
CommonKADS	N/A

Table C.9 Definitions of Task.

Source	Definition
i* [165]	“Specifies a particular way of attaining a goal.”
KAOS [145]	“Environment operations are performed by human agents, devices or existing software agents in the environment of the software-to-be. They are sometimes called tasks when they are performed by human agents.”
IREB Glossary [58]	N/A

UML [76]	N/A
BPMN [109]	“An atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail. Generally, an end-user, an application, or both will perform the Task. A Task object shares the same shape as the Sub-Process, which is a rectangle that has rounded corners.”
CommonKADS [131]	“A task is a piece of work that need to be done by an agent.”
CommonKADS [50]	“A task statement refers to a set of coherent activities that are performed to achieve a goal in a given domain. Here ‘activity’ refers to actual actions that are performed in the world, whereas a task statement adds a teleological dimension to these actions.”
CommonKADS [102]	“A task is an activity that should be performed in order to reach a goal.”

Table C.10 Definitions of Activity.

Source	Definition
i* [165]	N/A
KAOS [145]	N/A
IREB Glossary [58]	N/A
UML [82]	“An activity is a unit of work that an individual in that role may be asked to perform, and that produces a meaningful result in the context of the project.”
UML [129]	“Ongoing nonatomic execution within a state machine. Contrast: action.”
UML [20]	“An operation that takes some time to complete.”
UML [60]	“An activity is a computation that executes for the duration of state.”
UML [100]	“A use case specifies an interaction between the system and one or more actors together with the activities performed by the system.... We name an activity so it does not include the name of the actor. Hence, we prefer Order Books over Online Customer Orders Books.”
BPMN	N/A
CommonKADS	N/A