Watch your (Development) Steps

Bert de Brock [1]

[1] *University of Groningen, PO Box 800, 9700 AV Groningen, The Netherlands*

Abstract

Agility in Information Systems Engineering should not be limited to the software design & implementation phase. Agility should already start early during requirements analysis and customer validation. The objective of this position paper is to sketch how to integrate requirements engineering with agile software development, a.k.a. *Agile RE*. We sketch our approach to support the agility-oriented development of the functional requirements for an information system and to describe our ongoing research on the Agil-ISE topic. Our main message: Align and improve the development path, not only the individual development steps.

Keywords

User Wish, User Story, Main Success Scenario, Alternative Scenarios, Use Case, System Sequence Description, Natural Language Generation, Software Procedures

1. Introduction

Agility often refers to a specific *software* engineering process, but agility in Information Systems Engineering should not be limited to the software design & implementation phase. Agility should already start early during requirements analysis and customer validation (aimed at shared understanding, a cognitive aspect of agile ISE). In this position paper we sketch our (wholistic) approach to the agility-oriented development of the functional requirements for an information system, as well as our recent, ongoing, and future research on this topic. Our approach is based on the combination of our practical experience and our theoretical insights. We concentrate on the *functional* requirements, which we want to get right. Our main message is to align and improve the development path: not only the individual development steps but also their proper combination and alignment. Alignment also supports traceability, as we will illustrate in Section 2.

Requirements Engineering and Software (Systems) Engineering should be attuned to one another, but the RE-world and SE-world seem two worlds apart. In this position paper we sketch how to integrate those two worlds. Popular examples of agile Software Engineering approaches are XP (eXtreme Programming, see http://www.extremeprogramming.org) and Scrum (see https://www.scrum.org or https://www.atlassian.com/agile/scrum).

We notice that certain development steps have textual contents (e.g., texts from future users/ customers) while other intermediate steps have graphical contents (e.g., UML-diagrams) and the final steps have textual contents again (e.g., software programs). See the next suggestive picture (suggesting that the different steps are based on different paradigms):

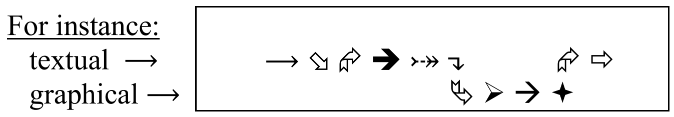


Figure 1: Non-aligned paradigms

Agil-ISE 2022: Intl. Workshop on Agile Methods for Information Systems Engineering, June 06, 2022, Leuven, Belgium
EMAIL: e.o.de.brock@rug.nl (B.de Brock)
ORCID: 0000-0003-4400-0187 (B. de Brock)

2. Our approach

Figure 2 shows a possible conceptual model for an agile development path for a functional requirement where existing notions are in white and our notions are in yellow:

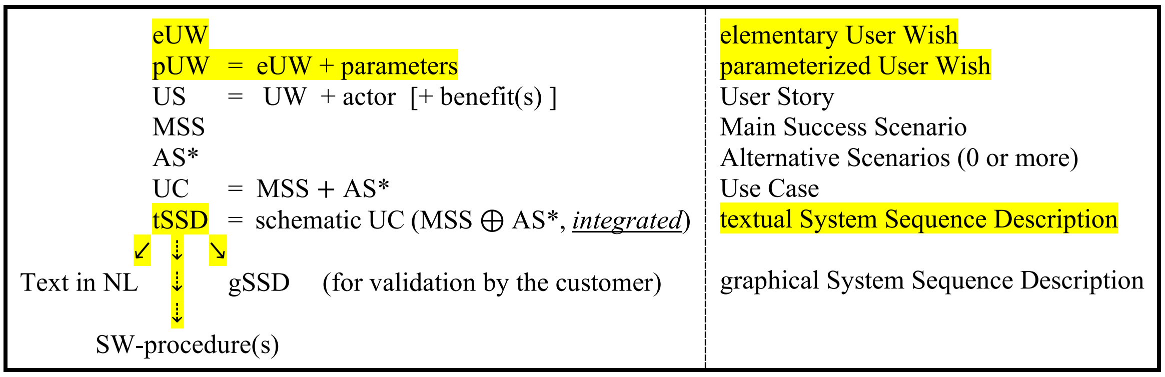

eUW
pUW = eUW + parameters
US = UW + actor [+ benefit(s)]
MSS
AS*
UC = MSS + AS*
tSSD = schematic UC (MSS ⊕ AS*, integrated)
Text in NL
gSSD (for validation by the customer)
SW-procedure(s)
elementary User Wish
parameterized User Wish
User Story
Main Success Scenario
Alternative Scenarios (0 or more)
Use Case
textual System Sequence Description
graphical System Sequence Description

Figure 2: A conceptual model for a complete development path for a functional requirement

During validation it might turn out that the result wasn't okay yet. In that case, we have to go back, most likely to the MSS or the ASs (as indicated in Figure 6). So, our development approach is *iterative*, i.e., improving 'the same piece' through successive refinements/adaptions. A 'piece' could be only *one functional requirement / use case*, or even only *one scenario* within a use case.

The results of a Requirements Analysis phase for a system essentially consist of a description of the *dynamics*, describing the relevant *processes* (i.e., what the system must be able to *do*), and a description of the *statics*, describing the relevant *data structures* (i.e., what the system must *know*). We specify the *dynamics* in the form of *textual System Sequence Descriptions* (*tSSDs*). The *statics* are often given in the form of a *Conceptual Data Model*. Together, the statics and the dynamics constitute a complete conceptual 'blue print' of the system to be developed, as summarized in Table 1:

Table 1
The aspects and how they are described

Aspect to be described	**Requirements Analysis result**
Dynamics / Processes	Textual SSDs
Statics / Data structure	Conceptual Data Model

For the dynamic aspects we developed:
- a *syntax* for our textual SSDs [1]
- a formal, declarative *semantics* for those textual SSDs [2]
- a mapping of those textual SSDs to Natural Language [3]
- a mapping of those textual SSDs to graphical SSDs [3]

The mappings of a textual SSD to Natural Language and to a graphical SSD can be used to validate the textual SSD (resulting from a Requirements Analysis phase) with the user community.

For the static aspects we developed:
- a mapping of conceptual data models to Natural Language [4]
- a mapping of conceptual data models to SQL [5]

The mapping of a Conceptual Data Model to Natural Language can be used to validate the Conceptual Data Model (resulting from Requirements Analysis) with the user community. The mapping to SQL gives (a first version of) an implementation design.

Figure 3 positions and links our 'development products' schematically (with existing components in white and our 'development products' in yellow):

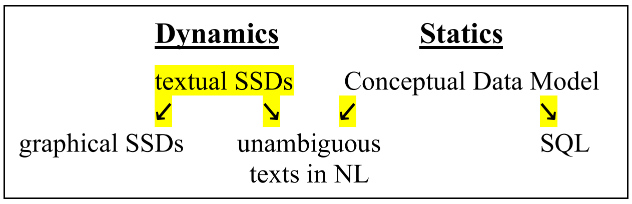


Figure 3: What we did so far

It is worthwhile to sketch a complete example of a development path for a concrete functional requirement, showing the artefacts produced along the development path. Figure 4 contains such an example.

eUW	Register a student
pUW	Register a student *with a given name, address, country, birth date, …*
US	As an administrator, I want to *Register a student* [because …..]
MSS	1. The user asks the system to <pUW> (*Register a student with …*) 2. The system fulfils <pUW> (*Register a student with …*) 3. The system sends result to the user (*e.g., registered data including the generated student nr.*)
AS*	AS1: If (s)he is a foreigner then … AS2: If (s)he was a student before then … AS3: … ⋮
UC	MSS + AS1 + AS2 + AS3 + … (But what if the foreigner was a student before?)
tSSD	User ➞ System: <pUW> ; **if** student is a foreigner **then** ….. *(even if the foreigner was a student before, as the customer told us)* **else** **if** (s)he was not a student before **then** System ➞ System: generate new student number ; System ➞ System: fulfil <pUW, with new student number> **else** ….. **end** **end** ; System ➞ User: result *(e.g., registered data including generated student number)*

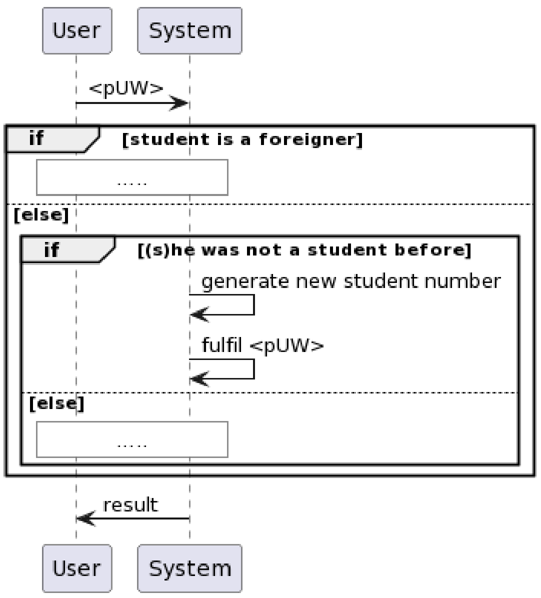
SW-procedure(s)	E.g., a stored procedure in SQL
	@output is declared as a return parameter here
	(Let's suppose that student number is declared as an `AUTO-INCREMENT` primary key field)
	`CREATE PROCEDURE RegisterStudent @n VARCHAR(50), @a VARCHAR, @c VARCHAR(20),`
	`@bd DATE, ...., @output VARCHAR OUTPUT`
	`AS`
	`BEGIN`
	`  IF @c <> 'NL'`
	`  THEN .....`
	`  ELSE IF .....`
	`       THEN BEGIN INSERT INTO Student(name, address, country, ...)`
	`                   VALUES(@n, @a, @c, @bd, ...)`
	`                SELECT @output = 'Done. New student nr. is '+ <...>`
	`            END`
	`       ELSE .....`
	`END`

Figure 4: A complete example of a development path for a concrete functional requirement

Note that the resulting procedure follows the structure of the textual SSD.

Figure 4 also illustrates the *traceability virtues* of our aligned approach: The original user wish (*Register a student*) forms a 'trace' all the way from that user wish down to the software procedure (`RegisterStudent`). Since textual SSDs can also be named in our syntax, the textual SSD produced during the development could be named *RegisterStudent* as well. All in all, the traces arise *during* and *as part of* the development, not produced as a (ceremonial) duty afterwards, as is typically the case [6].

In Figure 5 we show the *graphical SSD* generated from our textual SSD and the generated *NL-text* next to it, but both with all scenarios integrated now. For the generation of the graphical SSD we used the drawing generation tool Plantuml (see https://plantuml.com/sequence-diagram, and our appendix for the code). For a complex example we refer to [7].



The User **asks the System to** <pUW>.
If student is a foreigner
  then .....
  else if (s)he was not a student before
    then the System **does** generate new student number**.**
    The System **does** fulfil <pUW>
    else .....
   end
end.
The System sends result **to** User

Figure 5: The *graphical SSD* and the *natural language text* generated from our *textual SSD*

All this is worked out in detail in the forthcoming book [8].

3. An Agile Method for Information Systems Engineering

Now we are ready to sketch an agile method for information systems engineering (Agil-ISE).

When a new user wish pops up, then you can 'walk the line' we sketched, i.e., follow the aligned development path, completely from *initial user wish* all down to *implementation* (say, in Java or SQL). The development path for a single functional requirement also includes *validation*.

We can write it out as a business process, an agile *ISE-business* process. In other words, this is about business process alignment and *ISE-business agility*. All in all, it brings us to the following agile method for information systems engineering (Agil-ISE):

An agile method for information systems engineering (Agil-ISE) for the development of a functional requirement	
eUW	An elementary User Wish comes in
pUW	Which parameters should it have?
US	Which benefit(s) does it have? (Mention at least one)
US	Which actor(s)/role(s)?
MSS	Write out the Main Success Scenario
tSSD	Convert it to a textual SSD
NL-text	Generate the corresponding NL-text
gSSD	Generate the corresponding graphical SSD (if desirable)
Validate	Let the customer validate the generated NL-text and/or gSSD
SW-procedures	Convert the (possibly adapted) tSSD to software procedure(s)
For each individual Alternative Scenario there is an additional integration step:	
AS	Write out the Alternative Scenario
tSSD	Convert it to a (local) textual SSD

MSS ⊕ AS*	Integrate it into the already existing textual SSD
NL-text	Generate the corresponding NL-text
gSSD	Generate the corresponding graphical SSD (if desirable)
Validate	Let the customer validate the generated NL-text and/or gSSD
SW-procedures	Convert the (possibly adapted) tSSD to (adapted) software procedure(s)

Note that each development cycle results in one or more actual software procedures.

Until now we discussed the development of an individual functional requirement (FR). Developing several FRs can be done *incrementally*, i.e., developing (and delivering) the system 'piece by piece'. And if there are several development teams to develop the FRs, an individual FR should preferably be handled *within* one team. Figure 6 illustrates these aspects. This way of working makes the approach well scalable.

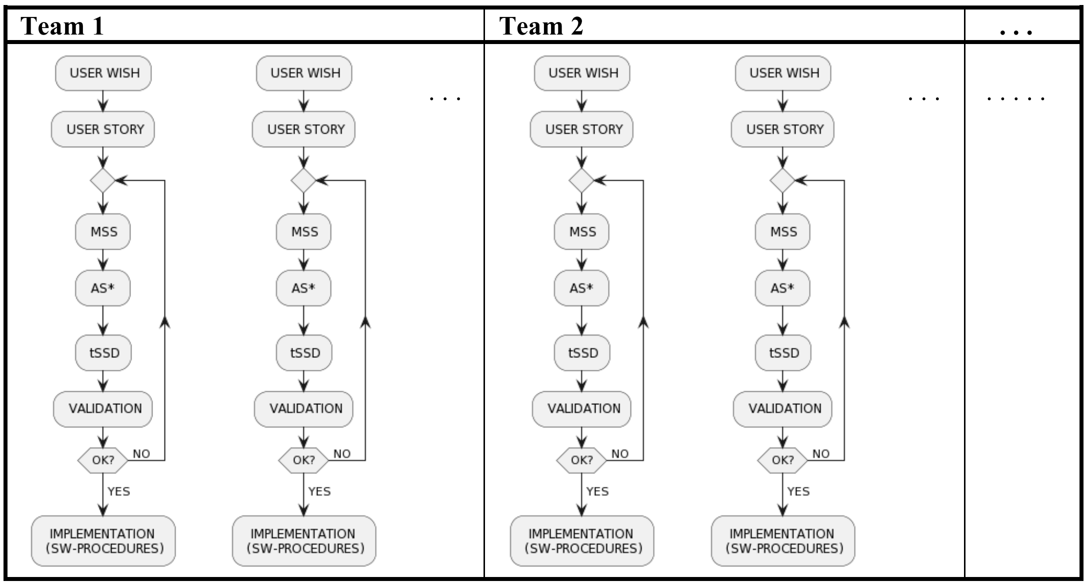


Figure 6 Iterative and incremental development with several teams

Furthermore, the usual agile practices can be applied here. We use the principles so nicely formulated in the Agile Manifesto (http://agilemanifesto.org/):

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Our approach is worked out in more detail in [8]

4. Future work / Research agenda

Currently we are working on a mapping of those textual SSDs to SQL (see the dotted downward arrow in Figure 2). Our planned future research is to elevate our theory to meta-level in order to be able to generate all this from a central repository (so, automation in agile ISE).

5. References

[1] E.O. de Brock: *From Business Modeling to Software Design*. In B. Shishkov (ed.): Proceedings BMSD (Business Modeling and Software Design), LNBIP 391, pp. 103–122 (2020)

[2] E.O. de Brock: *Declarative Semantics of Actions and Instructions*. In B. Shishkov (ed.): Proceedings BMSD (Business Modeling and Software Design), LNBIP 391, pp. 297–308 (2020)

[3] E.O. de Brock: *On System Sequence Descriptions*. In M. Sabetzadeh et al (eds.): Joint Proc. of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track (2020)

[4] E.O. de Brock and C. Suurmond: *NLG4RE: How NL Generation Can Support Validation in RE*. In J. Fischbach et al (eds.): Joint Proceedings of REFSQ-2022 Workshops, Doctoral Symposium, and Poster & Tools Track (2022)

[5] E.O. de Brock: *Foundations of Semantic Databases*. Prentice Hall (1995)

[6] R. Kasauli, E. Knauss, J. Horkoff et al: *Requirements Engineering Challenges and Practices in Large-Scale Agile System Development*. Journal of Systems and Software, vol. 172 (2021) http://dx.doi.org/10.1016/j.jss.2020.110851

[7] E.O. de Brock: Converting a non-trivial Use Case into an SSD: An exercise. SOM Research Report 2018011, University of Groningen (2018)

[8] E.O. de Brock: *Developing Information Systems Accurately - A Wholistic Approach*. Springer books (2022)

6. Appendix: Plantuml-code used to generate the graphical SSD

The Plantuml-code used for generating the graphical SSD has a close resemblance to the original tSSD:

```
User -> System: <pUW>
group if [student is a foreigner]
       rnote over User, System #white: …..
else else
group if [(s)he was not a student before]
        System -> System: generate new student number
        System -> System: fulfil <pUW>
      else else
        rnote over User, System #white: …..
      end
end
System -> User: result
```