Teaching and practicing RE for agility

Bert de Brock [1]

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

Abstract

During the development of an information system, requirements might constantly change. The practice of Information Systems Engineering clearly evolved because of the adoption of agile methods. However, after practice and theory of agile development, *teaching* agile development should follow. We have to let our students experience agile development and to teach them how to deal with it, not only in theory but also in practice. E.g., in practice it is difficult to keep everything consistent in a constantly changing situation. Also, for students it is (very) strange that something which is judged good in an earlier stage might be judged wrong in a later stage. In this paper, we describe how we simulate agility in our course and how we let the students experience constantly changing requirements, and how we teach them how to handle that. And because of a sudden pandemic (COVID), we were forced to teach all this on-line, with all its educational restrictions (e.g., no personal meetings, limited human interaction/dynamics, etc.).

Keywords

Changing requirements, information systems, agile development, increments, iterations, teaching agility, course design, turning on-line

1. Introduction

During the development of an information system, you usually have to deal with constantly changing requirements, because internal and external circumstances might change during development. The practice of Information Systems Engineering (ISE) has evolved substantially because of the adoption of agile methods. However, after practice and theory of agile development comes *teaching* agile development: We have to teach our students how to deal with this, not only theoretically (incremental and iterative development, agility, etc.) but also how to handle it effectively in practice. Moreover, at a certain moment, we had to teach the course completely on-line (because of the COVID-rules).

Therefore, the goal of this paper, intended for requirements engineers in general and RE-educators in particular, is to describe our improved teaching practice to teach our students how to deal with constantly changing requirements. Or, with a variant on the theme of Agil-ISE, the theme of this paper is:

How can teaching practice be improved or innovated to make it more effective in supporting our students with the agility of ISE processes in practice?

The document is based on our course *Problem Analysis & Software Design*. One of the goals of the course is to let our students experience how difficult it is to keep everything consistent in a constantly changing situation and to teach them how to handle that. Moreover, students are not yet used to the phenomenon that something which is judged good in an earlier stage might be wrong in a later stage.

The rest of the paper is organized as follows: In Section 2, we position the course within our curriculum and within several ACM-curricula. Section 3 describes the expected learning outcomes and some main characteristics of the course. In Section 4 we describe how we arrange the course. Section 5 enumerates the topics covered in the course. In Section 6 we describe how we made the course completely on-line. Finally, we summarize and draw conclusions in Section 7.

Proceedings Name, Month XX–XX, YYYY, City, Country
EMAIL: e.o.de.brock@rug.nl
ORCID: : 0000-0003-4400-0187

2. Positioning the course in a curriculum

The course is given in the second quarter of the second year to our students in Computing Science (compulsory). The course is elective for AI-students in their third year. Beforehand, the CS-students had some courses on programming, logic, and algorithms & data structures, among others, and a database course called *Introduction to information systems* (https://www.rug.nl/ocasys/fwn/vak/show?code=WBCS021-05). For CS-students, the course precedes the course *Software Engineering* (https://www.rug.nl/ocasys/fwn/vak/show?code=WBCS017-10). Our students already have some background in system design, programming/implementation, and testing. But in this course, the emphasis is on requirements analysis and software design. For the CS-students, it is the first CS-course without programming.

We can also position the course in terms of computing curricula, say those of the ACM [1,2]. The course would fit in several of those curricula:

For the *Undergraduate* Degree in the *Software Engineering* program, the course covers the knowledge area Requirements analysis and specification; see [3].

For the *Graduate* Degree in the *Software Engineering* program, the course covers much of the knowledge area Requirements Engineering; see [4].

For *Undergraduate* Degree Programs in *Computer Science*, the course covers part of the knowledge area Software Engineering, in particular the knowledge unit Requirements Engineering; see [5].

For *Undergraduate* Degree Programs in *Information Systems*, the third part of the course deepens the course IS Project Management, in particular for the (technical) manager of an agile development project; see [6].

3. Expected learning outcomes

As formulated in our Course Catalogue, at the end of the course the student is able to:

(1) use gained experience in using methods and tools for problem analysis and software design efficiently
(2) structure and redefine problems and ask adequate questions (problem analysis)
(3) make a good design for a software application (software design)
(4) get to the heart of a problem using abstraction and modelling and be able to indicate whether existing solutions are applicable (design patterns)
(5) adequately document the results of problem analysis and software design so that both the client and the programmer can get to grips with this
(6) work in a project-based way and in a team to come to an problem analysis and software design.

Or formulated alternatively:
For problems that might be solvable by means of software:

- to be able to clarify, structure, and redefine inadequate (and changing) problem statements and ask adequate questions to the problem owners (Problem Analysis part)
- to be able to make a good software design that can serve as a starting point for the implementation (Software Design part)

So, **no** clearly formulated problems and **no** programming in this course (only design). For (early) second year CS-bachelors, this is difficult to get used to: They are used to (and fond of) solving clearly formulated problems by programming. We let our students experience agile development and teach them how to deal with changing requirements.

Theme of the course is how to come from initial user wishes to a design in a way that has certain 'desirable properties', properties such as *straightforward, modular, flexible, quick, incremental/agile, transparent, traceable.*

We emphasize that RE is not a stand-alone topic but that RE and the subsequent steps (and their feedback for RE) should all be aligned. The emphasis of the course is on practice that is based on theory and principles.

4. How we arranged the course

For several years, my colleague and I are the teachers of the course. First I do the Problem Analysis part (five lectures of 2 hours), then my colleague does the Software Design part (also five lectures of 2 hours), and finally I do the Organization & Management part (two lectures of 2 hours). In between there are some Q&A sessions. All this theory is given during the first 4 weeks, because the students need the theory for their assignments and need time for the iterations (see the iteration deadlines below).

Each year, a new case is used. The cases we used concerned, among others, an auction house, a moving company, a ferry company, a school stay organization, and a teaching effort system for a faculty of a university.

How to let the students experience constantly changing requirements and teach them how to handle it in an agile way? For that purpose, the case is extended and/or changed every single iteration, in such a way that it might also influence the students' previous work! This is meant to let the students deal with the realistic situation that clients will change their requirements while the project is going. In this way, the students experience how difficult it is to keep everything consistent with everything else. In the ferry case, for example, a use case *Delete a reservation* made perfectly sense, initially. However, in a later iteration, the customer wanted to know whether reservations were used well/seriously or whether there were many cancellations or 'no shows' (and why). But then the 'old' use case *Delete a reservation* is considered harmful and should itself be deleted or changed into something like *Archive a reservation* (probably including a reason). Students need to experience that something which is judged good in an earlier iteration might become wrong in a later iteration, and they need to learn how to handle that.

In order to assess the students in our course, we issue a case for which a software system must be designed (not built). We deliberately deliver the case description in 3 rounds / 'iterations', where each description is about 1 to 1½ page. A team needs to write its report in 3 iterations: In each iteration, the report must include their analysis and design regarding that case so far. The intermediate reports will be commented on by the teaching assistants and the teachers in order to help the students to get a better end report. That end report is assessed by the teachers. Some parts are assessed for the individual student and some parts for the team as a whole (e.g., the team's conceptual data model and class diagram).

Each new part of the case description will be revealed immediately after the report deadline for the previous round. The first deadline is after 2 weeks, the second one after 4 weeks, and the last one after 10 weeks. So, the first two rounds are a kind of 'sprint': a short, time-boxed period where a team works to complete a set amount of work. Sprints are at the heart of agile methodologies.

Altogether, a team of 3 students has to do the analysis and design of 6 use cases. In Iteration 1, based on Part 1 of the case description, each team must create 3 relevant use cases, one by each student. In order to get a good spread of clearly different use cases within a team, we stipulate that it should preferably be a Create, an Update, and a Delete. Subsequently, the teacher annotates those use cases.

In the next round (Iteration 2), each use case must be improved by another student in the team, based on our annotations, and then the third student must make a software design for it. The rationale of this is that in practice, you might also have to continue with the work of others. Subsequently, the teacher annotates the report again.

In the third round (Iteration 3), each software design from the previous round must be improved by the original student. Hence, the rotation scheme is:

Student A: Create Use Case	→	Student B: Improve Use Case	→	Student C: Make SW design	→	Student A: Improve SW design

Also in Iteration 3, one of the students in the team must create a relevant Read use case based on Part 2 of the case description, while the other two students must create a relevant use case based on Part 3 of the case description. In that same round, for each new use case, a different student must make a software design. This time, the rotation scheme is 'backwards':

Student A ← Student B ← Student C ← Student A

Two e-mail addresses are available for the students: One towards the customer (for questions about the case) and one towards the helpdesk (for questions about the course itself). My colleague played the customer. He answered the questions 'in a customer's way', i.e., not always concretely, maybe incompletely, or sometimes even inconsistently (with earlier statements), and not knowing our terminology.

(For example, when a student asked which actors were involved, the customer reacted somewhat irritated: '*What do you mean with actors? Are we going to perform a play??*'.)

For the report we provide a template (in LaTeX), with sections such as *Management Summary*, *Introduction*, *General remarks*, *Actors and other stakeholders*, *Elaboration of use cases*, *Data model*, *Class diagram*, and optionally *Notes on possible implementations*, *Potential future developments*, and *Appendices*. It must include an appendix *Correspondence with the customer* and may include an appendix *Glossary: Terms and abbreviations*.

For drawings (of use case diagrams, sequence diagrams, domain models, data models, class diagrams, and packages) we use - and advise our students to use - the open-source drawing generation tool PlantUML (https://plantuml.com). Its specification language matches very well with our grammar for textual System Sequence Descriptions (SSDs). So, it provides an easy way to generate the corresponding graphical SSDs. For that purpose, we also wrote a small, dedicated PlantUML tutorial; see [7].

The teams preferably consist of three students. The teams are formed by the lecturers, deliberately mixed internationally (e.g., one Dutch student, one non-Dutch EU-student, and one student from outside the EU). Our argument against potential objections from the students (because students often want to work together with their own 'mates') is that in practice, you cannot choose your colleagues either.

Not all students that are enrolled will actually follow the course. However, we have to know who will actively participate, so we introduced an (easy) 'Iteration 0' in which each student must write down individually what the relevant user wishes, user stories, and involved actors are, based on the first part of the case description. We call it a 'confirmation of participation'. If a student does not submit an Iteration 0, (s)he is considered 'out'. After that, we compose the teams. The first thing a team must do, is discuss and integrate the 3 individual results of Iteration 0.

Over the recent years, the number of students were growing (and will still be growing). For instance, in the academic year 2020/2021, about 200 students enrolled, while almost 160 students actively followed the course. In that academic year, we had 53 teams (51 of 3 students and 2 of 2 students). It resulted in more than 900 pages in Iteration 1, more than 1,500 pages in Iteration 2, and more than 3,100 pages in Iteration 3. So, on average, there were almost 60 pages per end report. To help us, we had 5 teaching assistants, so that they could work in parallel.

Until a few years ago, we mainly used the book *Applying UML and patterns* of Larman [8]. Gradually, we introduced our own papers and an own syllabus. Meanwhile, I worked out and extended the *Problem Analysis* part and the *Organization & Management* part, which resulted in the textbook [7].

5. What we treated in the course

This section gives a global impression of the topics covered in the course. Most topics might be familiar to you, but some might be less common. The course consists of 3 parts: Problem Analysis, Software Design, and Organization & Management. That last part is not so much targeted at the general software manager but more at the (technical) manager of a development project.

We open the course with the well-known tree swing cartoon ("How the Customer explained it", etc.; see https://www.smart-jokes.org/how-IT-projects-really-work.html), for instance. We use the caption *The problem area* under the cartoon.

The topics covered per part are as follows:

(A) Global contents of the Problem Analysis part:

1. **Introduction:** Sketch of the problem area, Overview of our approach, Functional and other requirements
2. **Developing a functional requirement:** User Wish (= User Story without role and reason), CRUD-verbs, User Story, Use Case (= Main Success Scenario + Alternative Scenarios), Dos and Don'ts for writing Use Cases, Scenario Integration, (*textual*) System Sequence Description, Grammar for textual SSDs, Generating *graphical* SSDs from *textual* SSDs. So, many 'agile ingredients'
3. **Development patterns for functional requirements:** General pattern, CRUD-patterns, Registration 'forms' and Search 'forms', Indicating the parameters one by one, Browsing, Personalization, Communicating with others
4. **Domain modelling:** What is it? Purpose, Finding its ingredients, Frequent concepts (customer, product, sale, etc.). Special cases: Many-to-many associations, Generalization/Specialization, Individual items vs. 'Catalogue' items, Concepts related to themselves (e.g., (sub)products), Representing trees, hierarchies, graphs, etc.

5 **Conceptual data models:** Purpose, Ingredients, From Domain Model to Data Model. Special cases: The same as under point 4 (Domain modelling)

(B) Global contents of the Software Design part:

1 **From Analysis to Design:** Black Box SD (Sequence Diagram), Grey Box SD, White Box SD
2 **System Design vs. Detailed Design:** From DM to Class Diagram, Different layers, Packages, Life Lines
3 **Sequence Diagrams vs. Communication Diagrams:** Advantages and disadvantages
4 **GRAS-patterns:** Creator, Controller, Façade, Adaptor, Observer, (Abstract) Factory, Visibility, etc.
5 **Design Guidelines:** Low Coupling, High Cohesion, Design by Contract, MVC, etc.

(C) Finally, and overarching the other two parts: **Organizing and managing the development process**

1 **Communication Problems:** Users' language/thinking ≠ developers' language/thinking, User wishes are vague and also changing, Communication shortcomings within the development organization itself, Scope of the system is unclear, System must support an entirely new business and/or business model, further explanation of that tree swing cartoon, and hence the need for agility
2 **How to overcome them?** By stepwise *clarification*, stepwise *refinement*, and stepwise *specification*. This is schematically expressed in the table below (followed by the abbreviations used):

Elaborating what …	**Problem Analysis** part		**Software Design** part
the system must **do**:	UW → US → UC (= MSS + AS*) → SSD	=	BB-SD → GB-SD → WB-SD
the system must **know**:	Domain Models* → Conceptual Data Model	→	Class Diagrams^{+} (refinements)

*: zero or more $^{+}$: one or more

Abbreviations used in the table

UW: User Wish	MSS: Main Success Scenario	BB-SD: Black Box Sequence Diagram
US: User Story	AS: Alternative Scenario	GB-SD: Grey Box Sequence Diagram
UC: Use Case	SSD: System Sequence Diagram	WB-SD: White Box Sequence Diagram

3 **Organization & Management Problems:** The 'times to market' must become shorter and shorter, hundreds (or thousands) of requirements, ever quicker changing circumstances (externally and internally)

All in all: The requirements are usually only clear at the end of the project… How to manage that?

Here again, we emphasize the need for agility in order to manage that

4 **Possible (and usual) consequences for the development project:** Insufficient functionality (too little), not within time (too late), and not within budget (too costly). *So, actually failing all three basic project requirements*
5 **Some methods history (Why are we where we are now?):** Waterfall methods, Parallel development, CASE tools etc., Forms of Prototyping, Iterative development vs. Incremental development, Agile development, and Continuous development
6 **How to overcome those Organization & Management Problems?** Arranging real customer participation and availability (on-site *customer* vs. on-site *developer*), Iterative & Incremental development, Short feedback loops, Agile/rolling planning (not predictive or prescriptive but *adaptive*), Reasons for changes, Dealing with (disruptive) changes, Ranking requirements (but only the next ones), Potential ranking criteria, Teams and team composition

6. How we made the course completely on-line

Given all the limitations during the COVID-pandemic, everything had to go on-line and personal meetings were not allowed during that period. The lectures and tutorials were held on-line using Blackboard Collaborate (https://help.blackboard.com/Learn/Instructor/Ultra/Interact/Blackboard_Collaborate). While one of the lecturers was lecturing, the other lecturer plus at least one teaching assistant attended, meanwhile answering questions in the chat. The students could react in the chat: asking questions, answering our questions, making remarks, etc. They were able to do it anonymously, by logging in with a nick name. We also recorded all our lectures (because students might be at their home in completely different time zones).

During the lectures, we were mainly presenting and discussing the slides. We rewrote all our lecture slides and tutorial slides (more than 600), to adapt to the new circumstances. During the tutorials, we were also 'writing' on the slides, denoting the suggestions and solutions. Although the students could

toggle between the slides and the video showing the lecturer, there was not much added value in looking at our 'talking heads'. We didn't see our (dozens of) students. This was a real drawback because we could not see whether they did understand the material.

To discuss the course with our teaching assistants, we used Slack and Google Meet (with video). Some teaching assistants were not even in the country. We also arranged a 'meeting with the customer', on-line via Blackboard Collaborate. Since the customer was played by my colleague, we pretended that the employee was working from home (as usual in those days), 'unfortunately' without an audio- and video-connection…

In previous years, before the pandemic, the students had to hand in printed versions of the intermediate reports and end reports at a concrete location at the faculty. We used those printed versions to make annotations, to cross out texts, to tear off and easily compare different pages, to draw arrows between conflicting statements, etc. Very handy (for us). With all the restrictions because of the pandemic, the students could (remotely) upload a pdf-version of their report in Blackboard, our digital learning environment, and we made our annotations on the digital versions. We could sometimes extend our annotations later on when they turned out to be unclear for the students. But, all in all, it was not that handy anymore.

Before the pandemic, the team members could (and should) also meet each other physically, but now they couldn't. Therefore, per team we made a group in our digital learning environment so that they could communicate with each other.

The move to on-line teaching had implications for the simulation of agile development practices because team meetings or personal meetings were not allowed then, while that is an important aspect in agile development (and in many other development paradigms as well).

7. Summary and conclusion

In this paper, we described how we simulated agility in a course (*Problem Analysis & Software Design*). We taught our students how to deal effectively with constantly changing requirements, theoretical as well as practical. For instance, students had to learn how to deal with the practical fact that something which is good in an earlier iteration might become wrong in a later iteration… We also described how we arranged this, also completely on-line (during the COVID period), with all its educational limitations. For instance, the student teams could not have personal (stand-up) meetings and missed many of the human aspects.

With this improved course design, we are more effective in supporting our students to deal with the agility of ISE processes in practice. The course is challenging and demanding, but the students learn a lot, not only the theory but also how to apply the theory in (agile) practice.

8. References

[1] Curricula Recommendations, ACM. URL: https://www.acm.org/education/curricula-recommendations.

[2] Computing Curricula 2005: The Overview Report. URL: https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2005-march06final.pdf.

[3] Software Engineering 2014. URL: https://www.acm.org/binaries/content/assets/education/se2014.pdf.

[4] Graduate Software Engineering 2009. URL: https://www.acm.org/binaries/content/assets/education/gsew2009.pdf.

[5] Computer Science Curricula 2013. URL: https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf.

[6] IS 2010: Curriculum Guidelines for Undergraduate Degree Programs in Information Systems. URL: https://www.acm.org/binaries/content/assets/education/curricula-recommendations/is-2010-acm-final.pdf.

[7] E.O. de Brock, Developing Information Systems Accurately – A Wholistic Approach, Springer Cham, 2023.

[8] C. Larman, Applying UML and patterns, Addison Wesley Professional, 2005.