Software Stories Guide

Ulrich Norbisrath[1], Albert Zündorf[2], Tobias George[2], Ruben Jubeh[2], Bodo Kraft[3], Sabine Sachweh[4]

1 Upper Austria University Appl. , 2 Kassel University, 3 Aachen University Appl. , 4 Dortmund University Appl.

1 Motivation

Classical requirements engineering approaches [Pohl2010] and analysis and design techniques [RJB2004] do not fit agile software development. Agile software development does not aim for a complete requirements elicitation before software development starts. Instead agile software development deploys an on-site customer [Beck2000] or a product owner [TN1986] [Schwaber2004] who provides requirements on-demand. Similarly, agile software development does not aim on a complete system analysis and design before software development starts. Instead agile software development uses refactoring [FB1999] to evolve the system architecture during development. While eXtreme Programming abandons analysis and design totally, there is an urgent need for requirements, analysis, design, and modeling techniques that blend into agile software development. Such an agile design technique needs to address requirements one after the other and shall allow to proceed to analysis, design, modeling, and actual software development, early on. It shall allow to add new requirements and functionality step by step and to extend the existing system accordingly. An agile requirements, analysis, design, and modeling technique should support iterative development and refactoring towards new and changing functionality. In addition, such a technique should enable domain experts and (IT) layperson to participate in the agile development as on-site customers and product owners. At least, layperson domain experts should be able to understand the ideas and concepts underlying the software system in order to be able to judge whether their requirements are met and where the software concepts may have faults or functionality is still missing.

Software Stories are an ideal means for agile requirements, analysis, design, and modeling. Software stories focus on concrete example scenarios thus allowing to address one functionality after the other. A Software Story addresses a single scenario not all possible scenarios, at once. Using examples from the every day life of the domain experts help them to express their knowledge about the process. If there are different cases, Software Stories use multiple different scenarios, one for each case. Thus, one may start with a simple scenario and add more and more complex cases in an agile manner. Within a Software Story, the domain experts may exemplify relevant data in the form it occurs to them. For example, they may add PDF forms, excel sheets, word documents, or GUI mockups of tools they already use. For Software Stories it is perfect to give just concrete examples of e.g. one filled PDF form for a certain process step.[1] Analyzing the concrete scenarios and the provided example data and turning it into classes and software is done by the agile software developers during system development.

This paper introduces Software Stories as a means for agile requirements engineering, system

1 Note, a blank form will not work. A blank form is meaningful for the domain expert, only. The software people will have serious problems understanding the meaning of all the empty form fields. (At least I (Albert) have always difficulties to fill forms.) A form filled with example data that is also known from other scenarios helps the software people to understand the domain, significantly.

analysis and system design and for some user centered design activities. We will first exemplify how we have used Software Stories for the development of a simple administrative tool at our research group. We will then discuss some important aspects and caveats of Software Stories and how Software Stories shall be used in an agile software development process.

2 Example for Current State Analysis

As an example we model the administrative process how bachelor and master theses are managed in our department and research group. This problem is somewhat "academic" as it does not originate from a typical business environment. However, the problem is typical for the kinds of problems addressed by software stories as multiple persons are involved at different points in time and the communication, coordination, and collaboration of these people currently lack transparency and tool support. Thus we wanted to develop a workflow tool supporting us in the coordination of these processes.

First, we needed to do some requirements elicitation defining the theses workflow at our research group. Instead of BPMN notation we did the requirements elicitation with Software Stories. Figure 1 shows the first two steps of our first Software Story for the Theses' Management Tool.

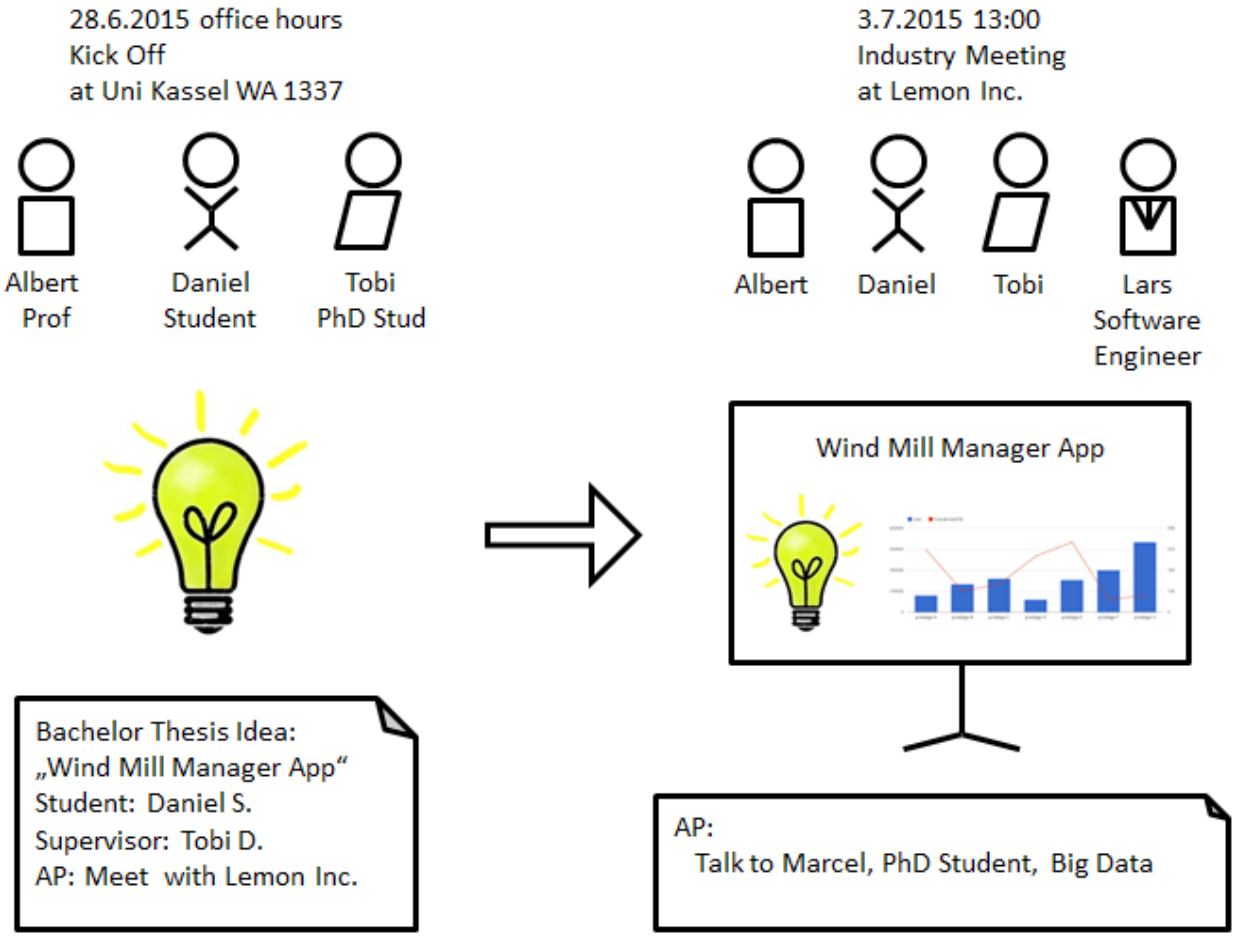


Figure 1: Software Story for Daniel's Bachelor Thesis, The start

A Software Story consists of a number of steps. A step describes a concrete situation and actions executed in that situation. A step also describes when this step has been (or will be) executed and

who participated in this step or who executed it. In Figure 1 the first step is shown on the left. It has been executed at June 28th 2015 at about 3pm. Daniel, a computer science student at Kassel University, met Professor Albert at Albert's office. Daniel wanted to do his Bachelor thesis under Albert's supervision. Daniel was student programmer at Lemon Inc.[2] in Kassel. Daniel was programming some Software managing Wind Mills for electrical power generation. Daniel wanted to do his Bachelor thesis in cooperation with Lemon Inc. in the context of his work at Lemon Inc. In our research group, the actual supervision of bachelor thesis is usually done by PhD students. Thus, Albert asked Tobi, one of his PhD students to join the discussion. Daniel gave some more information on the context of his work at Lemon Inc. The topic seemed to be interesting and related to our research. Therefore, we agreed to supervise Daniel's Bachelor thesis. For theses in cooperation with an enterprise, in our group a meeting with a supervisor provided by the enterprise is mandatory. Thus, Daniel got the task to organize such a meeting.

The second step of our example is shown on the right of Figure 1. On July 7th 2015 at 1pm Albert, Daniel, and Tobi met with Lars at Lemon Inc. Lars was a software engineer responsible for the Wind Mill Management[3] software developed at Lemon Inc. Daniel gave a short presentation on the content of the planned Bachelor thesis and Lars provided more details to the technical challenges. It turned out that Lemon Inc. was looking for a Big Data solution to do the recording and analysis of sensor data from a huge number of wind mills. As Big Data is the special expertise of Marcel, another PhD student of our group, it was discussed to contact Marcel and ask him to supervise Daniel's thesis and to collaborate with Lemon Inc. on the Big Data issues.

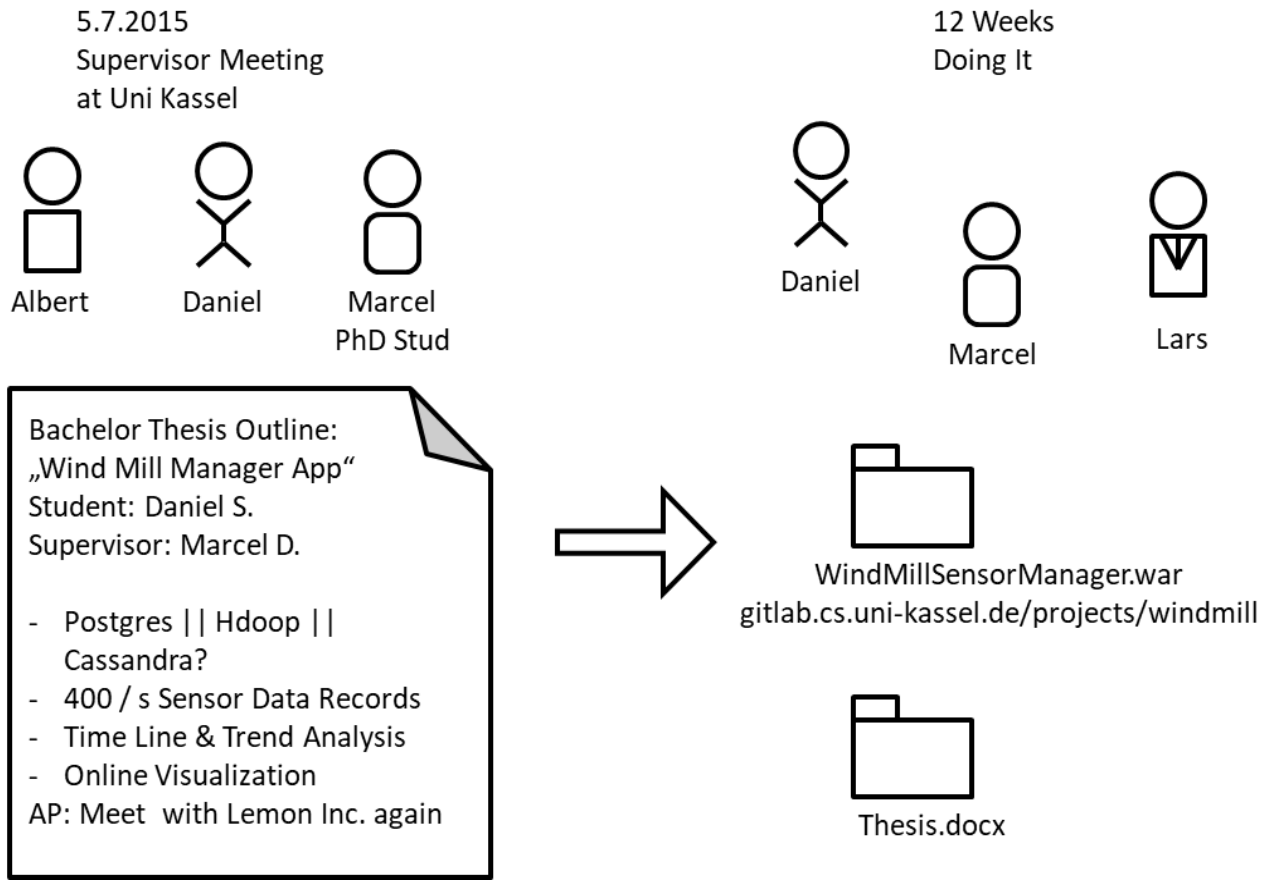


Figure 2: Software Story for Daniel's Bachelor Thesis, Doing It

2 Name changed by editors.

3 Thesis topic simplified and translated from German.

Figure 2 shows on the left the meeting with Albert, Daniel, and Marcel on July 5th. Marcel was happy to help in this thesis and brought at lot of ideas to the table. In the thesis outline the most important topics were added. These included which database system would serve best for the thesis, how much data we expected, and what should be done with the data. Another meeting with Daniel, Marcel, and Lars was scheduled in order to get Marcel and Lars to know each other. Then the actual thesis work started.

The actual thesis work is shown on the right of Figure 2. This has been done mostly by Daniel. Marcel and Lars only assisted by giving directions for areas that needed more investigation and how to organize work and how to structure the thesis itself. This lasted about 3 month. The outcome was the implementation of the Wind Mill Sensor Manager Web App and the thesis document.

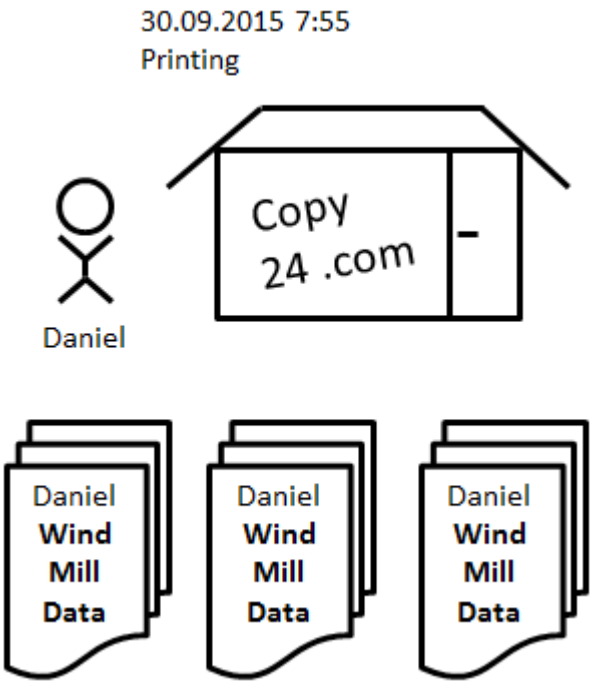


Figure 3: Software Story for Daniel's Bachelor Thesis, Printing

As shown in Figure 3, in the morning of September 30th Daniel went to the copy shop and 3 copies of his thesis were printed.

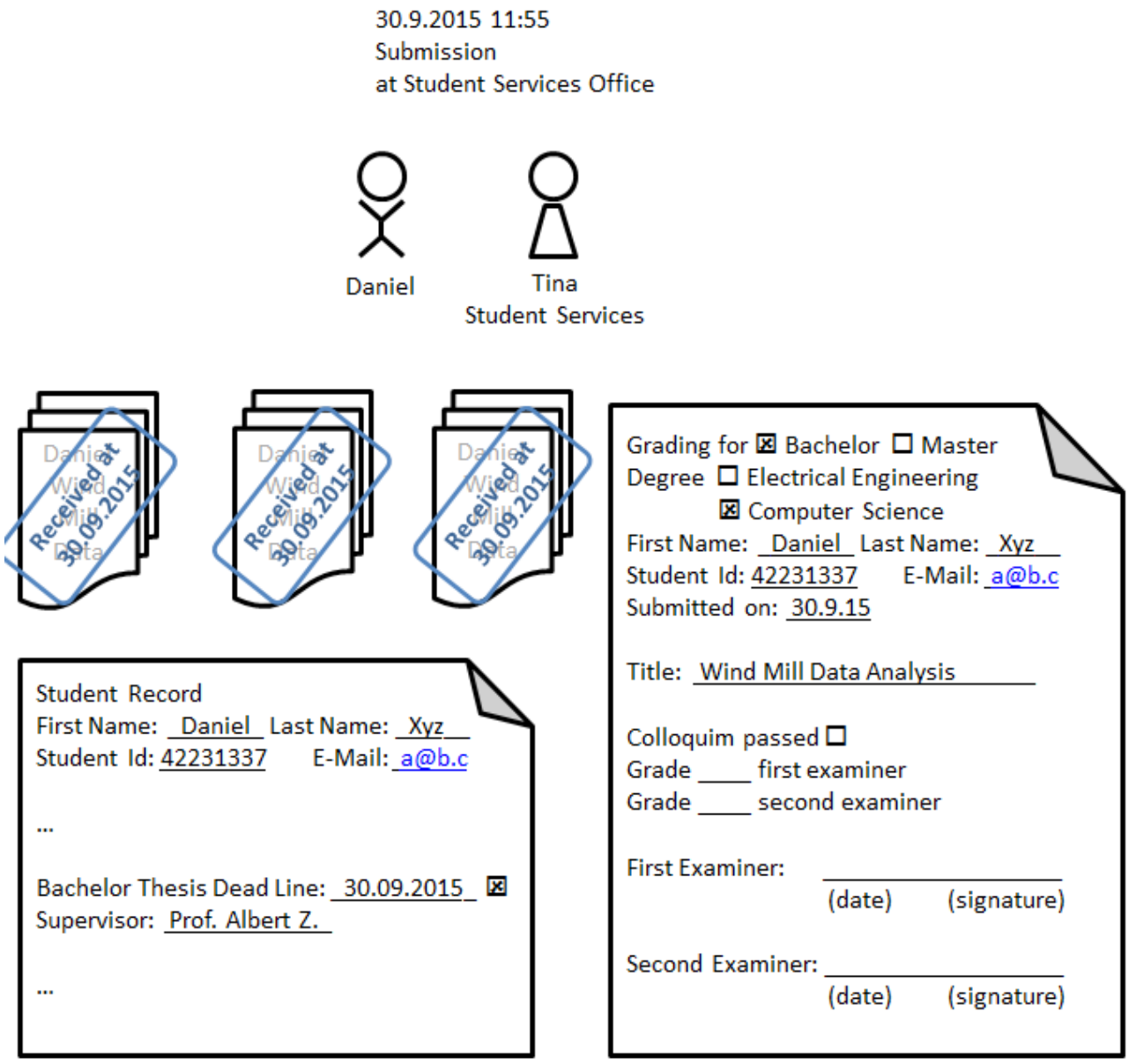


Figure 4: Software Story for Daniel's Bachelor Thesis, Submission

Then the difficult administrative process started. Just in time Daniel entered the Student Service Office of our department and handed in the three copies, cf. Figure 4. Tina is working at the Student Service. She stamped the three thesis copies with the date of their reception. Then, Tina fetched Daniel's Student Record and marked the dead line for the bachelor thesis as met. Finally, Daniel and Tina filled in two copies of the Thesis Grading Form[4] [3] of our department. Tina handed the stamped copies of thesis and the two grading forms to Daniel. Daniel had the responsibility to deliver the copies and the forms to his examiners.

4 Forms simplified and translated from German, therefore also using European date notations.

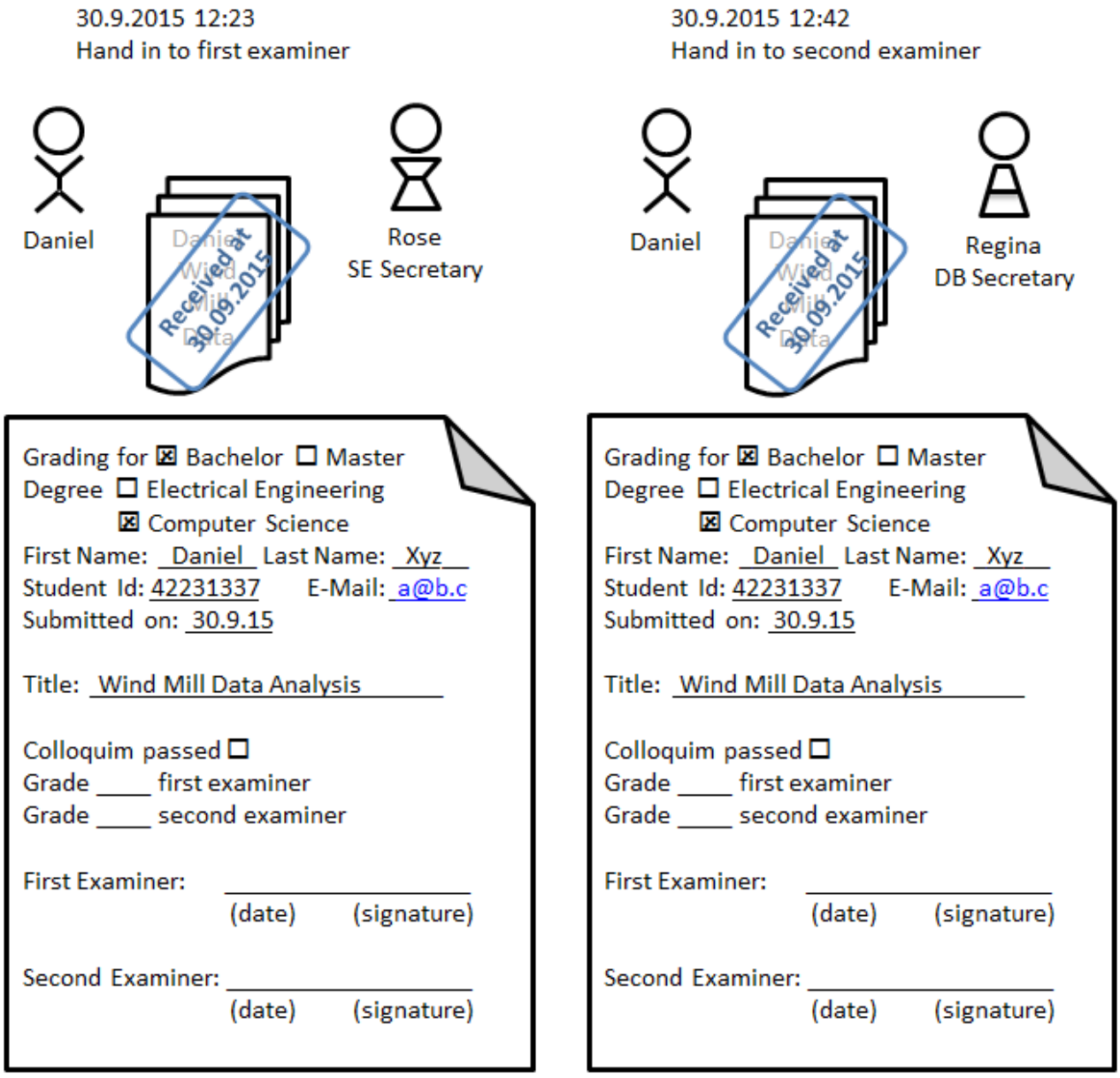


Figure 5: Software Story for Daniel's Bachelor Thesis, Examiners

Figure 5 shows how Daniel delivered one copy of his thesis and one copy of the grading form to Rose, the secretary of the Software Engineering group, and to Regina, the secretary of the database group, respectively. The professors of these groups, Albert and Lutz, are the two examiners for Daniel's thesis.

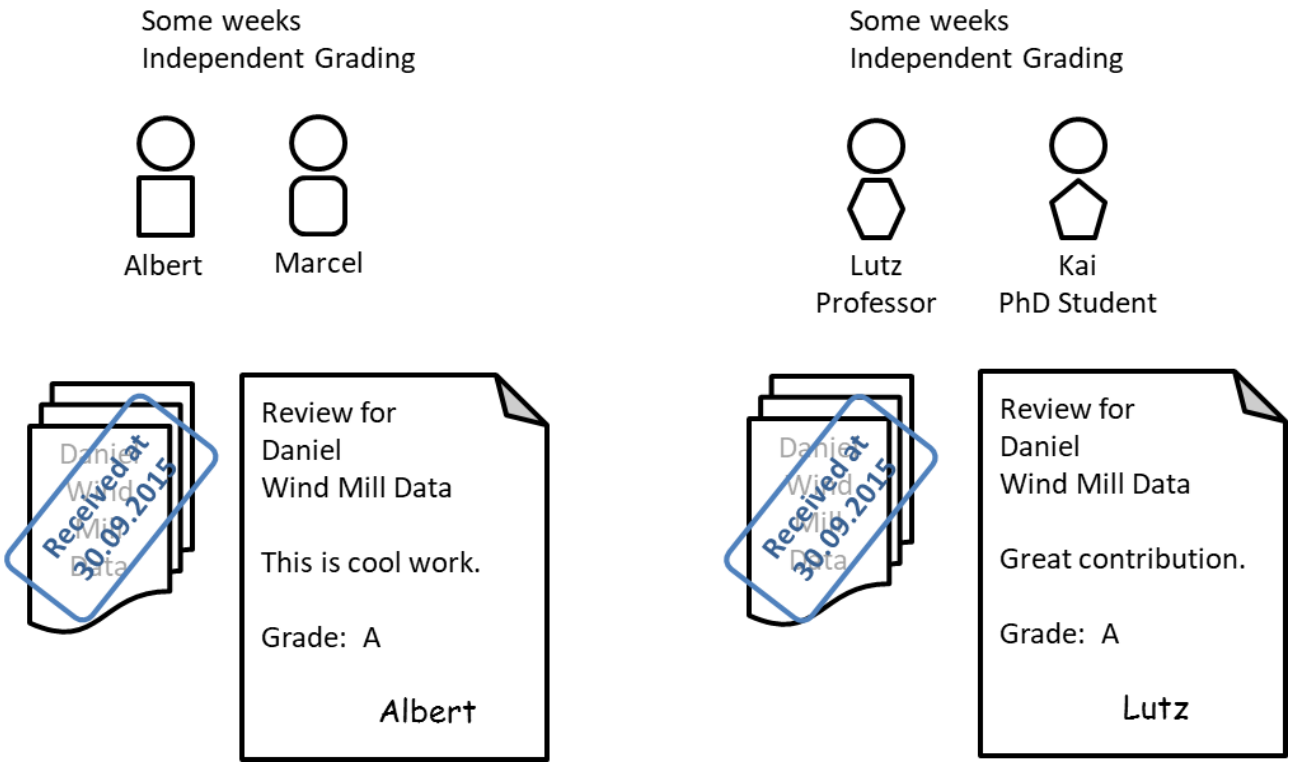


Figure 6: Software Story for Daniel's Bachelor Thesis, Examination

In Figure 6 Albert and Lutz read Daniel's thesis and wrote a review giving a grade for it. This has been done completely independent from each other. There are rumors that some professors do not read the theses themselves, but just ask their PhD students to do this. There are other rumors that sometimes the second examiner waits with his review until the first examiner sends his review as a guideline. Actually, in our department bachelor and master thesis reviews are not mandatory and sometimes the second reviewer does not produce one.

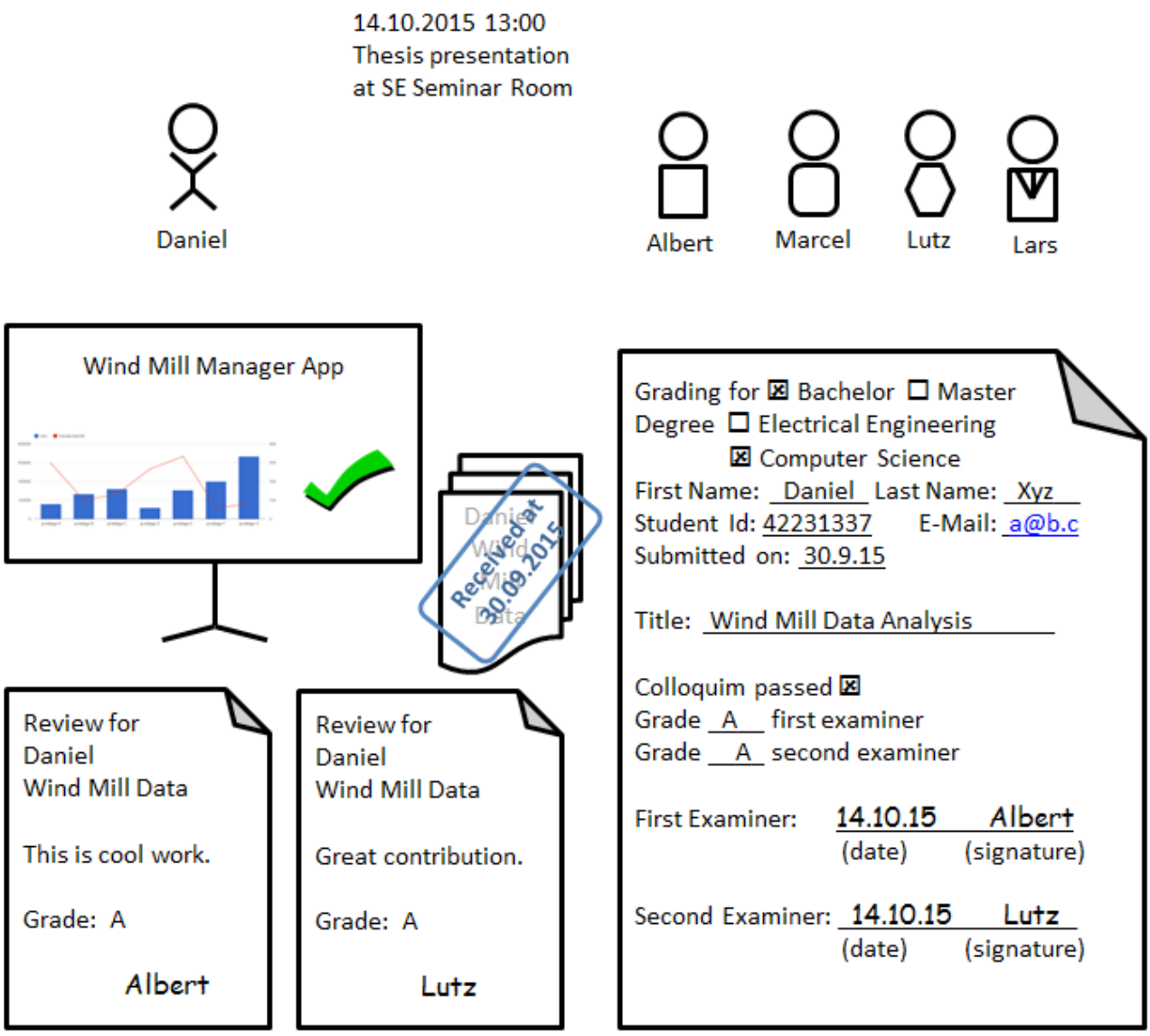


Figure 7: Software Story for Daniel's Bachelor Thesis, Colloquium

On October 14th Daniel had his Bachelor Thesis Colloquium, cf. Figure 7. Daniel gave a presentation of his results. There were several interested people from our group and especially Albert as first examiner, Marcel as university supervisor, Lutz as second examiner, and Lars as the industrial supervisor. After the presentation there were a lot of questions on details and a lively discussion. At the end, Albert, Marcel, Lutz, and Lars asked for some privacy and then discussed their impression from the work done by Daniel, from his thesis, and from his presentation. Finally, Albert and Lutz filled in and signed the Grading Form for Daniel's thesis. Now its almost done.

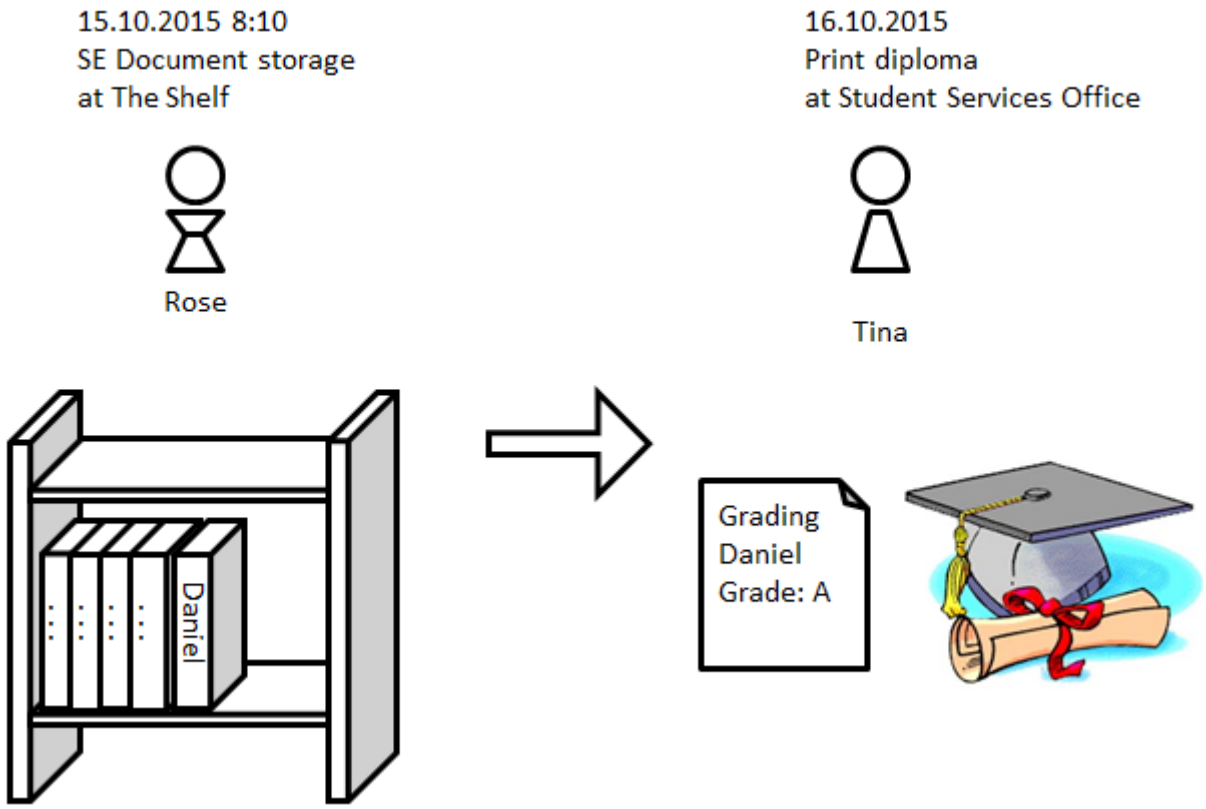


Figure 8: Software Story for Daniel's Bachelor Thesis, Diploma

As the very last administrative steps, Rose the secretary of our group adds the hard copy of Daniel's thesis to our library book shelf and she keeps a copy of the Grading Form for Daniel's thesis in our records, cf. Figure 8 left side. Then Rose sends the original Grading Form to Tina, the Student Service secretary. Tina, completes the Student Record of Daniel and creates Daniel's Diploma.

3 Example for Application Design

Albert wrote down the Software Story of Daniel's Bachelor Thesis discussed in Section 2 as a first attempt at describing the problem. It shows only the current situation. Later, it turned out that this story is still incomplete and that there are some other scenarios that are not covered by Daniel's story. For example, internal theses without an industrial partner are somewhat simpler as the coordination with the industrial partner can be omitted. In addition, cases where Albert is only second examiner require some internal steps beyond those performed by Lutz in Daniel's story.

As a first step towards the design of an application helping us to manage theses in our group we did a requirements elicitation workshop with Rose, our secretary, Tobi, the PhD student going to implement the thing, and Albert, eager to evaluate Software Stories in practice. At the beginning of this requirements workshop, Albert explained Daniel's Software Story to Rose and Tobi using a large white board. During the discussion some missing elements popped up:

- For the very first step of our Software Story, we decided that we want a desktop application allowing us to record the protocol of the first meeting with a student bringing up the idea for a thesis. Thus we added a GUI mockup to the first step of Daniel's Software Story, cf. Figure 9.

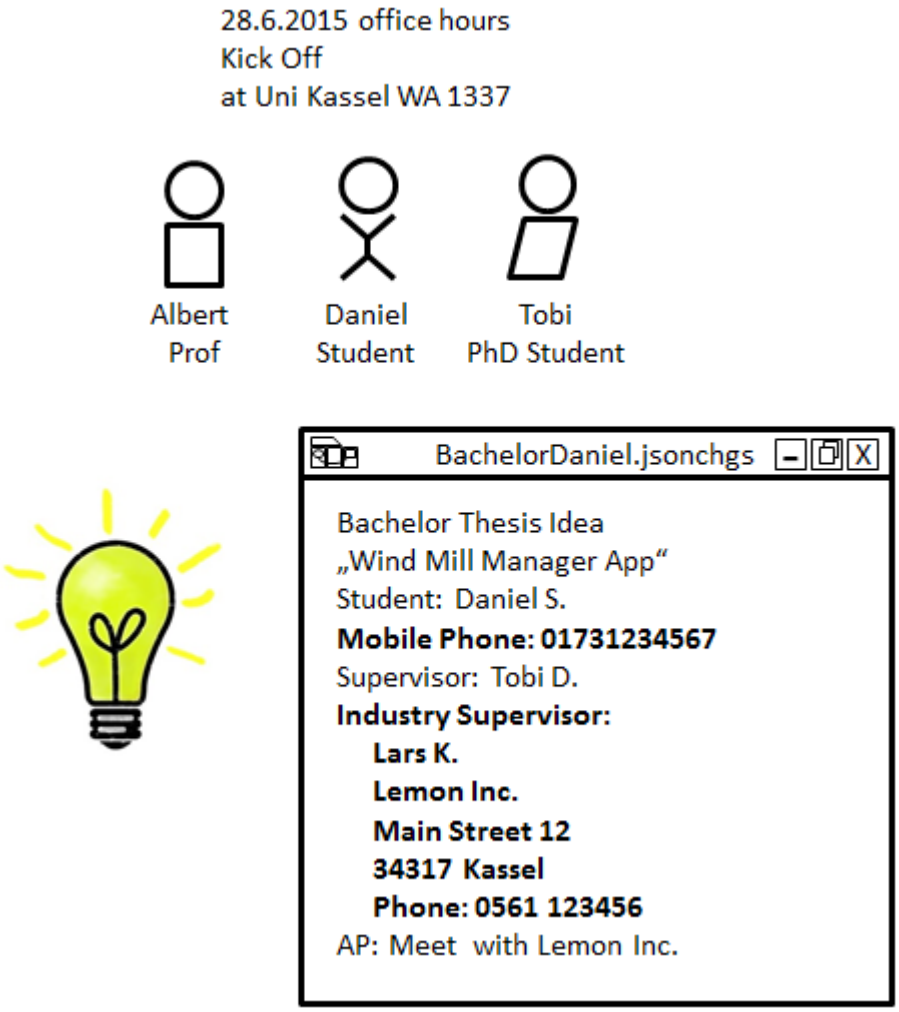


Figure 9: Gui Mockup Create Thesis

- Next, Tobi recalled that we were late for the first meeting at Lemon Inc. We had no phone number nor contact name to call to announce that we are late. Similarly, there have been cases where we reached an enterprise in time but when the door man asked for our contact we could only name the student who had no entry in the enterprise's phone book. To avoid such problems, we decided that in case of an external thesis our new Theses Management Tool shall force us to record the student's mobile phone number and the contact information for the industrial supervisor. See the bold parts of the GUI mockup in Figure 9.
- Then, we had a discussion whether the new Theses Management Tool should also support the process of agreeing on dates and times for meetings. This usually involves checking Albert's Google Calendar and the calendars of the other participants and may require to set up a Doodle call. We anticipated problems in accessing the calenders of external participants and did not come to a conclusion for this feature. However, the Thesis Management Tool should allow to add action points or tasks on the fly and support us in keeping track of their execution.
- Beyond action points the new Theses Management Tool might also allow to take notes on meetings. Thus, we would be able to protocol e.g. the outline of a thesis as done in Figure 2. However, the Thesis Management Tool was meant to facilitate the administration of theses and we were not sure whether it should deal with the actual content of theses. We decided to keep things simple at first and keep such a feature in mind for future extensions.
- As we reached the submission of Daniel's thesis in Figure 4, we noticed that Albert omitted an important step in the whole process: a thesis needs to be registered at the Student Service

Office before you can submit it. Actually, the department rules require that you register your thesis before you start working on it. The rational behind this is that the amount of time spent on for example a Bachelor Thesis should not exceed 9 weeks. Thus, on registration the Student Service Office sets a deadline for the submission of the thesis and this deadline is controlled on submission. For some reasons students tend to be sloppy with the registration of their theses'. To enforce early registration, the Student Service Office set up the rule that one may not submit a thesis earlier than 4.5 weeks after registering it. To address the registration of Daniel's thesis, we added another step to our Software Story as shown in Figure 10.

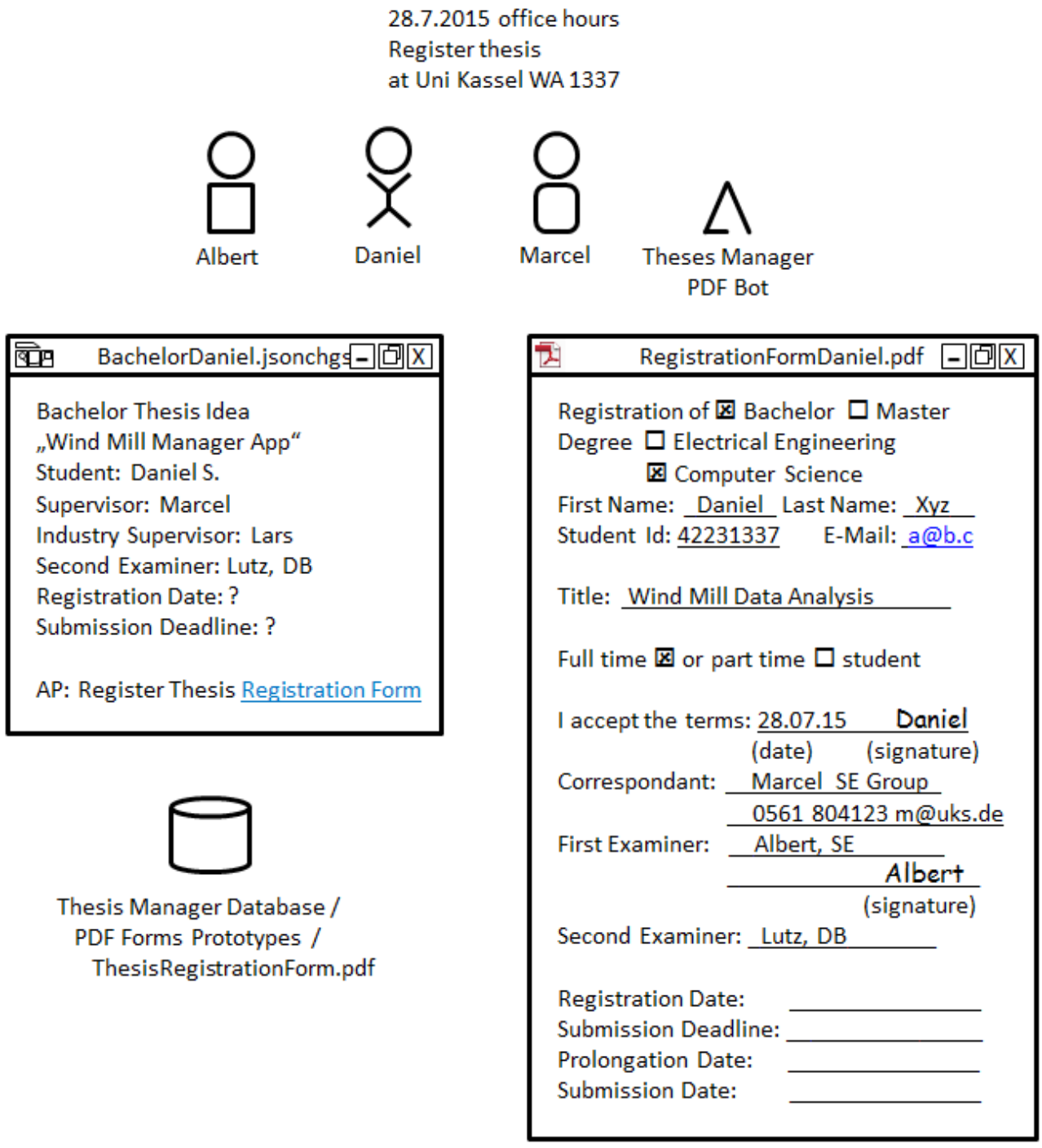


Figure 10: Thesis Registration

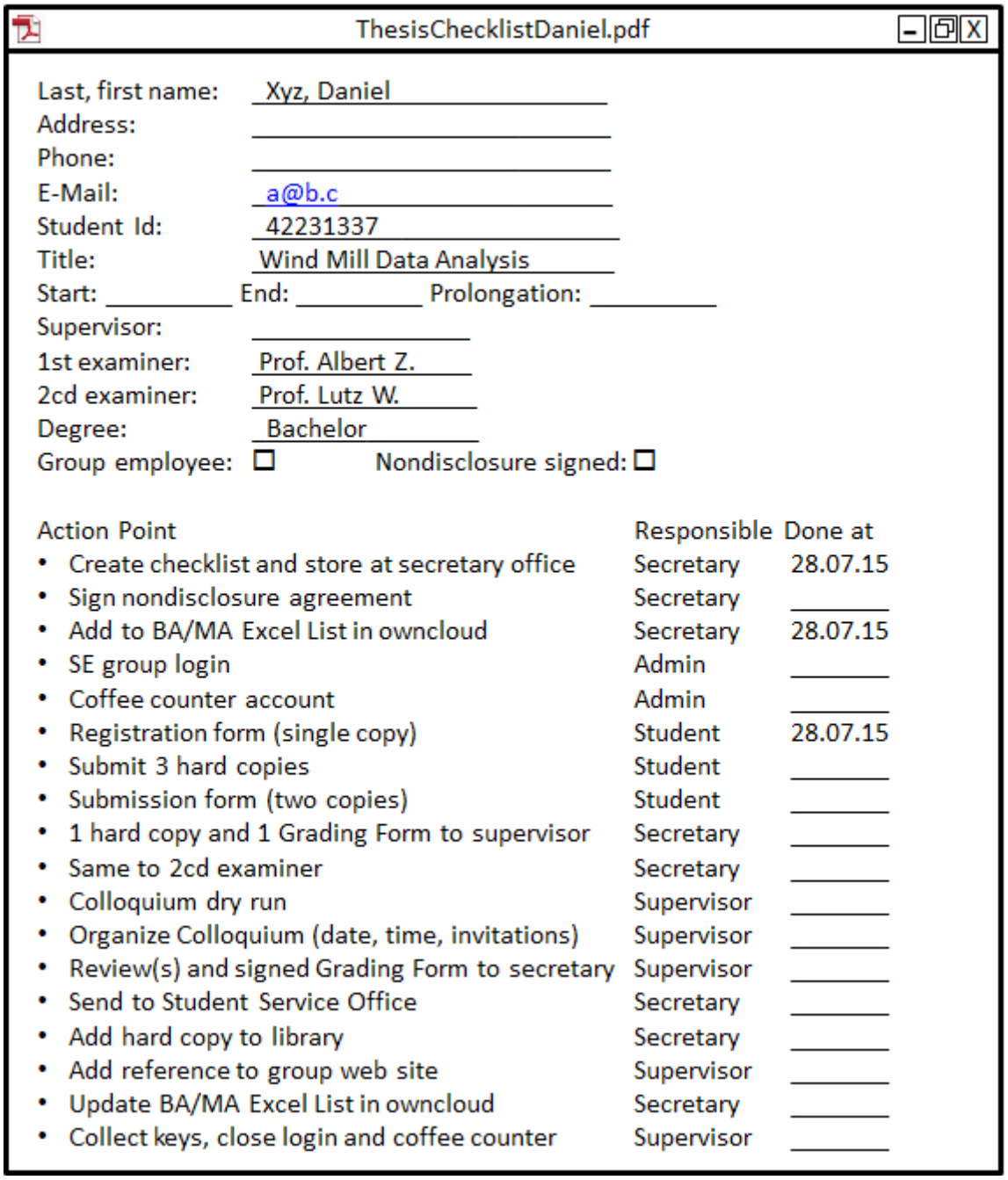
- Analyzing the Register Thesis step brought up an essential idea for our Thesis Management

Tool. We noticed that during the process one enters the credentials of various persons and for example the title of the thesis multiple times in multiple forms. Thus, the new Thesis Management Tool should have an address book keeping track of the contact data of all involved persons. Some contact data may also be imported from other sources, our group has for example a web based assignment management system for the courses given by us. Most likely, Daniel is already registered in this system with his credentials. Next, a lot of information managed by the Theses Management Tool needs to be transferred into various PDF forms provided by the Student Service Office. Thus, it would be great if the new Thesis Management Tool provides links to the appropriate PDF forms and if the new Thesis Management Tool would be able to auto-fill such forms with the data it already has. In the example of Figure 10 a click on the link "Registration Form" shown in the bottom right corner of the Theses Management Tool shown as mockup on the left of Figure 10 should open some PDF tool with the Registration Form for theses and the Theses Manger's PDF Bot shall auto-fill as many fields of the PDF form as possible. In our example, the PDF Bot might auto-fill all shown information except the signatures. This auto-fill feature will greatly enhance the motivation of the members of our research group to actually use the Thesis Management Tool and to keep its content up to date. This is considered a killer feature.

The Story Step shown in Figure 10 includes a non-human actor like the Thesis Manager PDF Bot, and another software component the Thesis Manager Database shown in the lower left corner. In Software Stories, such components resemble software functionality that are parts of the system under design. Later, these components will be implemented by the software development team.

- Next, our secretary Rose recognized that some students need to prolongate the submission deadline. Theoretically, this should not happen, but sometimes reality can be cruel. Our Student Service Office provides another PDF form for this case. Thus, our Thesis Management should allow to add such an action point on demand and it should auto-fill that form too and keep track of the changed submission deadline.

- Finally, our secretary Rose came up with another PDF form which she uses to keep track of theses, cf. Figure 11. Our secretary's checklist revealed a number of new insights. First of all, we should have asked her on the first run. She had much more insight in the administration of theses than the other group members. Second, there is a lot of redundancy in the current system. There are the forms used by the Student Service Office, the form used by our secretary, and an excel list in our owncloud, cf. Figure 12. And currently there is little motivation for most group members to enter the same information into multiple forms again and again. This results in the excel list being outdated all the time and in a lot of frustration for our secretary as she cannot answer questions from students nor questions from the Student Service Office like who is in charge of supervising a certain thesis and where are the hard copies of that thesis located and when the Colloquium is scheduled.



ThesisChecklistDaniel.pdf

Last, first name: Xyz, Daniel
Address:
Phone:
E-Mail: a@b.c
Student Id: 42231337
Title: Wind Mill Data Analysis
Start: _____ End: _____ Prolongation: _____
Supervisor:
1st examiner: Prof. Albert Z.
2cd examiner: Prof. Lutz W.
Degree: Bachelor
Group employee: ☐ Nondisclosure signed: ☐

Action Point	Responsible	Done at
• Create checklist and store at secretary office	Secretary	28.07.15
• Sign nondisclosure agreement	Secretary	
• Add to BA/MA Excel List in owncloud	Secretary	28.07.15
• SE group login	Admin	
• Coffee counter account	Admin	
• Registration form (single copy)	Student	28.07.15
• Submit 3 hard copies	Student	
• Submission form (two copies)	Student	
• 1 hard copy and 1 Grading Form to supervisor	Secretary	
• Same to 2cd examiner	Secretary	
• Colloquium dry run	Supervisor	
• Organize Colloquium (date, time, invitations)	Supervisor	
• Review(s) and signed Grading Form to secretary	Supervisor	
• Send to Student Service Office	Secretary	
• Add hard copy to library	Secretary	
• Add reference to group web site	Supervisor	
• Update BA/MA Excel List in owncloud	Secretary	
• Collect keys, close login and coffee counter	Supervisor	

Figure 11: Our secretary's check list

Bachelor /Master							
	Deadline	Colloquium	BA/MA	1.Examiner	2.Examiner	Supervisor	Remarks:
Ole	26.02.2015	End of April	BA	Prof. Zündorf	Prof. Sick	Tobi	does an internship in Austria
Alex	19.08.2015	Midth of Sept	MA	Prof. Zündorf	Prof. Stumme	Lennert	e-mail to Lennert 19.08.2015
Constantin	21.08.2015	End of Sept	BA	Prof. Zündorf	Prof. Wenzel	Tobias	
Seppel	02.09.2015	Midth of Okt.	BA	Prof. Zündorf	Prof. Geihs	?	
Chris	10.09.2015		MA	Prof. Zündorf	Prof. Gheis	Tobias	
Bob			BA	Prof. Zündorf	?	Lennert	still an idea
Mirco			BA	Prof. Zündorf	?	Marcel	
Ed	september 14?			Prof. Zündorf			
2. Examiner:							

Figure 12: Theses Overview in our Owncloud

- To overcome our problems with the administration of theses, we decided to build a workflow software that helps all of us to keep the informations about theses up-to-date and consistent. The users of this workflow software would be the scientific members of our group that supervise the theses and our secretary that deals with student requests and communicates with the people outside of our research group like the staff of the Student Service Office and supervisors of other research groups. In addition, our secretary circulates a lot of copies of various documents. As GUI for the system we wanted some overview over all theses and their current states. This overview might be organized like the Excel table shown in Figure 12. In addition, for each single thesis we want a view like the checklist shown in Figure 11. This view should provide some common data about the student and the thesis at the top and it should show a checklist of TODOs and people in charge. TODO items that are done might go to the bottom of the list and form some kind of history. Current TODO items should appear on top. Future TODO items might show in the middle. TODO items that involve filling a PDF form shall provide links to that form and should auto-fill it as much as possible.

As the new workflow software mimics a checklist, we decided to name it E-Checkman for Electronic Checklist Manager.

As a next step we developed prototypes for the GUI of E-Checkman. These will be included in the next version of this paper.

4 Example for Concurrency Design Details

Our thesis administration system shall allow concurrent changes by multiple users and thus we have to deal with concurrency issues. The following Software Story outlines our concurrency concepts.

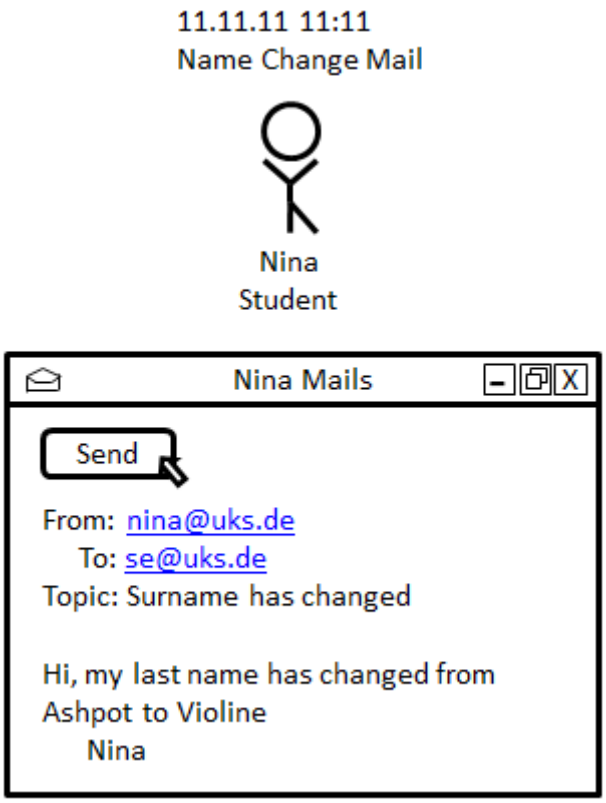


Figure 13: Nina mails her name change

At the beginning of our scenario, Nina sends an email announcing that she has married and that her new last name is now Mrs. Violine. At 12:12 o'clock Albert and Tobi both react to that email and open the student record tool on their respective PCs, cf. Figure 14. Albert just changes Ashpot to Violine, not recognizing that Nina's last name is stored in the first name field, erroneously. Albert's student record tool employs a so-called property change listener shown as "albert edit listener" in Figure 14. This change listener notices the change of the first name field and sends a change notification to the SE Student Records Database shown at the bottom of Figure 14. The change description consists of a change number that is merely a time stamp, the id of the changed data record, the name of the changed record field and the new value. Actually, the SE Student Record Database does not store data records but just such change records. To retrieve such a full data record for Nina, you retrieve all changes referring to Nina's record id 123456. New changes to a certain field of a certain data record replace old changes of the same field. Thus, the change with the timestamp 12:12:12 will replace some old change with recordId `123456`, fieldName `firstName`, and newValue `AshPot.`

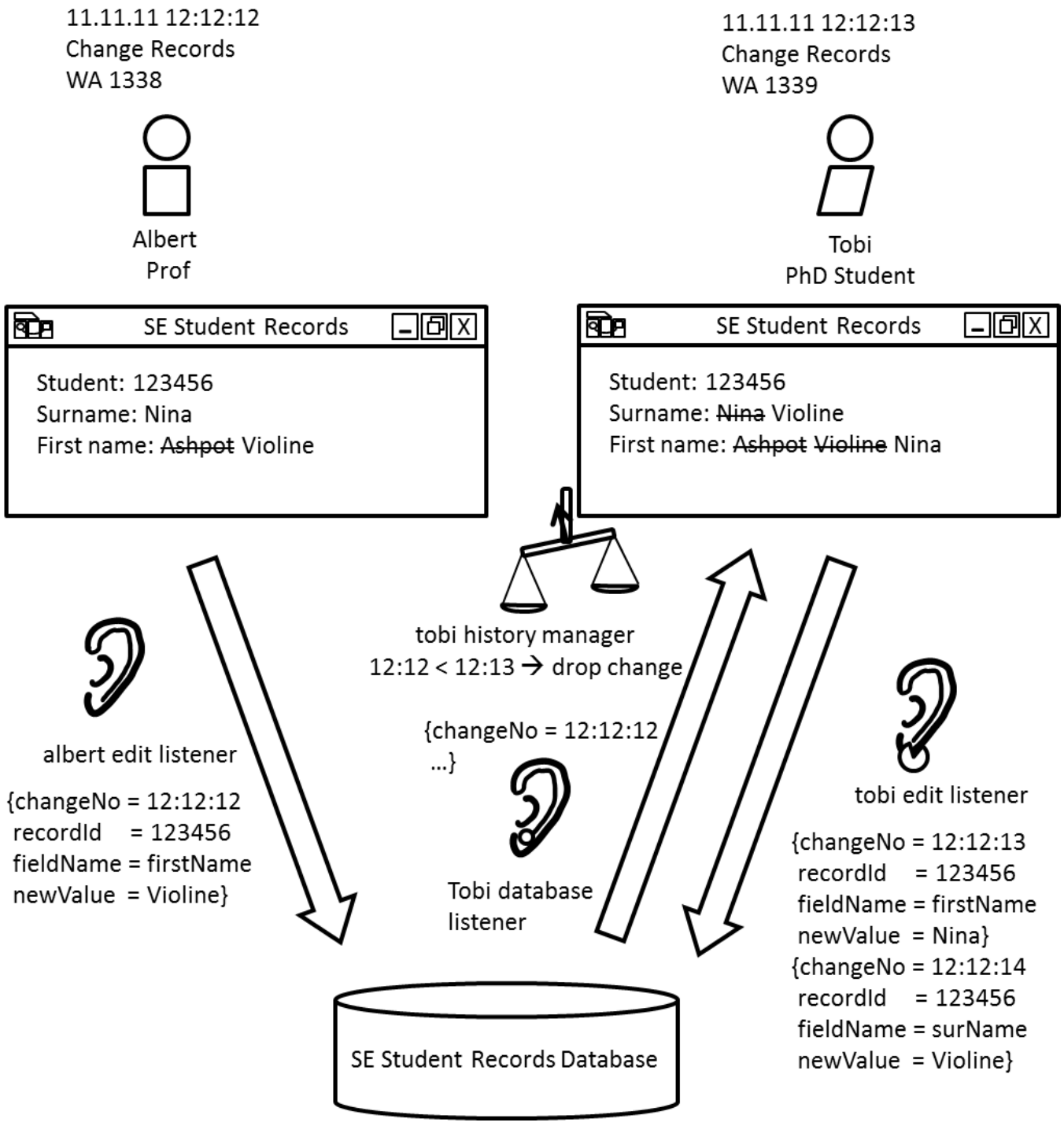


Figure 14: Handling concurrent record changes

Tobi's student record tool has a database change listener. This database listener notices change 12:12:12 when it is added to the SE Student Record Database. Thus, the change is send to Tobi's student record tool in order to update Tobi's GUI. Meanwhile, Tobi changes the surname and the first name field of Nina's data record. Tobi's edit listener recognizes that Tobi has entered `Nina` as new first name and `Violine` as last name. Thus, Tobi's edit listener creates change number 12:12:13 and 12:12:14 shown in the lower right corner of Figure 14. These changes are not only send to the SE Student Records Database but Tobi's history manager keeps track of these changes, too. Thus, when Albert's change with number 12:12:12 reaches Tobi's student record tool, Tobi's history manager looks into its change records and finds record 12:12:13 that changes the same field of the same data record. As the change number is interpreted as time stamp, Tobi's history manager

notices that change 12:12:13 is newer than Albert's change and thus it drops Albert's change.

Similarly, the SE Student Records Database has its own history manager. In our scenario, the database history manager receives Albert's change first and Tobi's changes later. When Tobi's change 12:12:13 arrives, the database history manager notices that this change effects the same attribute as Albert's change but it has a newer time stamp. Thus, in the database the new change 12:12:13 replaces Albert's old change. If Albert's change 12:12:12 needs some more time to reach the SE Student Record Database and Tobi's changes have already arrived, the database history manager would identify the conflict of changes 12:12:12 and 12:12:13 and it would drop Albert's change instead of storing it.

Overall, the history managers are used to detect conflicting edits on the same model element, Nina's names in our case. Without the history managers Albert's change would have been send to Tobi and applied to Tobi's data and at the same time Tobi's change would have been send to Albert and applied there. At the end, Albert has “Nina Violine” (as received from Tobi) and Tobi has “Violine Violine” (the first name received from Albert and the last name changed by Tobi himself. Thus without the history managers the two copies of the data would become inconsistent in case of conflicting concurrent edits.

The outlined scheme of handling concurrent edits depends on the time stamps of changes. Usually, the later change wins. This may result in so-called lost updates. Tobi may change the first name field without noticing Albert's change. Well, this happens only when Tobi changes the first name field in that split second that is needed to transport Albert's change to the database and from there to Tobi's tool. Formally, it is not a lost-update, if Albert's change is shown to Tobi for some millisecond. Actually, Tobi might see only some flickering in the GUI while starting his own change. Thus, from the user’s perspective, there is little difference whether Albert's change is shown for a millisecond or it is ignored at all. On the other hand, if Tobi does his change a split second earlier than Albert, he would change the first name to Nina and when Albert's change arrives some milliseconds later, Albert's change would be recognized as the later change and the GUI would be updated accordingly. As Tobi has just edited the field, he would probably see that his change has been overridden and he would just repeat it in order to set the first name to Nina, correctly. When two people edit the same data at the same time, you may expect such situations. And letting the last change win is a reasonable behavior. However, if this is for example a flight reservation system, you would probably want that the earlier reservation wins. In such cases you need another concurrency handling scheme to be discussed in another paper.[5]

5 Notation Details

A Software Story consists of a sequence of Software Story Steps. Each step should have

- a time and date (or a duration),
- an activity name,
- a location,
- some actors that execute the step or participate in the step,
- some picture or bullet list outlining the content of the step,
- some example data showing what information is processed and produced in this step.

5 One simple solution for a fair seat reservation system is to let the customer not change the seat data record directly but the customer creates a new data record for a reservation request. Then some seat manager component may listen for new reservation records. The seat manager may wait some time for other reservation records that are still on their way through the network. The seat manager component then assigns the seat to the earliest reservation he knows. (Reservations arriving lately due to network latency may have bad luck.)

The activity name of a Story Step frequently refers to a later implementation of the illustrated activity. For example, in our E-Checkman system, the name "Register thesis" of the Story Step of Figure 10 refers to an Action Point or TODO item in the E-Checkman GUI that will be used to open the Thesis Registration Form of our department and to (auto) fill this form. Thus, the name of a Story Step is merely used for tracing the described functionality in the later implementation.

Each Story Step should provide a time and date when this example step has been executed or will be executed. The first purpose of the step execution time is to help the people developing the Software Story to focus on the example level. It is not the “Register Thesis dialog is opened by somebody” but it is “July 28th 10:42” and it is “Albert together with Daniel and Marcel”. We experienced that forcing domain experts and other Software Story developers to start with a concrete point in time helps them significantly to stick to a concrete example and to prevent them going to a more abstract (rule level) description.

Next, the concrete time and date helps to deal with parallel and concurrent activities. At the rule level, concurrency issues require a lot of careful design regarding when a particular activity might be executed, which other activities need to be completed first, and which set of activities might be mutual exclusive. In a concrete example, it is much easier to specify just when the activity is executed in this particular example run. One may choose similar dates for two activities done by different actors in a single Software Story in order to give a hint that these two activities are independent from each other. One may also choose a sequence of points in time for a number of Story Steps in order to emphasize that the involved Story Steps trigger each other. One may also indicate that some Story Step happens several days later as another department or another organization is responsible for it and information has to be send around and the issue may need to wait before it is scheduled.

In general, timing and concurrency issues are complicated topics that need sound analysis. However, in early requirements engineering where domain experts are involved, it is nearly impossible to address such issues in all details. Thus, we made the experience, that domain experts as well as software engineers can do a reasonable job in discussing concurrency issues just by providing time stamps for Story Steps.

The location where a Story Step is executed again helps the Software Story developers to focus on a concrete example. It also gives an idea of the kind of infrastructure you might expect for the execution of such a step. Story Step "Register Thesis" of Figure 10 is executed in the office of Albert at Kassel University. However, we can expect that a desktop computer is available and that we have access to the Internet and we have access to the E-Checkman system of our group. In other situations like on the right of Figure 1 we might only have a mobile system available or as shown in Figure 4, we might be in a different department and thus we might not be able to access E-Checkman.

Next, each Story Step should provide the actors that execute it or that are involved. In our example, these actors are usually persons like Albert, Daniel, or Tina. These persons represent users of the described software system. However, to focus on the concrete example we use individual names to refer to the involved actors and we also use individual icons for each of them. Still, the different Actors represent different roles or different kinds of users. These roles may be given as a second name on a second name line like Albert Prof, Daniel Student or Tina Student Service. We frequently provide the role of an actor on its first occurrence in our Software Story and omit it later on.

One additional remark on concrete named actors versus just role names: Different people doing the same job might frequently execute it quite differently. For example, some years ago the

administration agent in charge for checking and refunding travel costs at Kassel University was Mister Four-Corner. When Mr. Four-Corner was in charge, it was enough to fill some simple form and to put all your receipts in a bag and send it to him. If there was something unclear, he would phone you and clarify things – issue solved. Then Mr. Four-Corner retired. The new guy in charge was Mr. Spare-Time. Mr. Spare-Time did not call you back but he just sent back your bag of receipts with a remark like "details are missing". This frequently required quite a number of iterations and you were waiting for you money for ages. Finally, we had to change the process. Now you give your bag of receipts to Rose, our secretary and she will ask you for the details and then send a detailed and complicated report to Mr. Spare-Time. The lesson learned is, the same user role in the same process may function quite differently depending on the concrete person doing it. One might argue that such cases indicate missing process standardization and strictness. However, even with good standardization, in practice, different people do the same job differently. Thus, processes and the accompanying tools need to deal with different personalities and need to be flexibly adaptable to such issues. During requirements engineering this means, when developing a Software Story, knowing the concrete person who is doing a certain step will greatly facilitate to describe that step. It is much easier to describe how Tina handles the submission of Daniel's Thesis than to describe how some Student Service servant handles the submission of some student. To generalize from the concrete example to the rule level is the task of software developers, later on.

Our example Software Story models a workflow with several human actors. In other examples, some steps may be executed by a software system that does for example some consistency checking or that calculates a certain price. Actually, the Register Thesis step shown in Figure 10 shows an incomplete triangle icon representing the PDF Bot component of our Theses Manager. This piece of software is responsible to open PDF forms and to auto-fill them using the data collected in our Theses Manager. If (that piece of) your software already has a logo, you may use that logo to represent it in a Software Story (as the open triangle resembles the Adobe icon). You may also use icons resembling a computer or a robot or for example a xerox machine to show software components that do a certain job. Adding software components as actors to your software stories greatly helps to illustrate internal activities executed by your program(s). They may also be used for internal discussions in the software development teams to clarify the responsibilities of the various software components to be developed. They also help domain experts to understand how the software works internally. This will help your users in working with the system. Note, a software component participating in a certain step of a Software Story always represents a running instance of that software component that is executed on a certain computer at a certain point in time. This shall not be mixed with the part of the software that implements this behavior. The difference is that the same program binary may be executed on different computers at a certain point in time and on each computer it may modify its own copy of the data, cf. Figure 14. Using runtime instances of a software in our Software Stories allows to model such situations. Still, the identification of software components and their roles in the example scenarios is a pretty good input for the software analysis and design. Later, the software developers will revisit these components and (together with some customer or product owner) clarify the details by refining the Software Stories with more data and more functional details. This may then serve as input for the component implementation.

The main content of a Story Step is some kind of PowerPoint slide that outlines what is going on in this step. This outline may contain any pictures or drawings or bullet items. This content is mainly read by humans and has just the task to help the involved domain experts and software engineers to get an idea of what is done in this step. On a white board, the participants will discuss the details of the Story Step content. When documented, some explaining text should be added to explain these things to the reader.

A very important part of the main Story Step content is the inclusion and visualization of some example data. In our example Software Story, we depict several paper sheets or PDF forms

showing notes, phone numbers, addresses, check boxes, and text fields. In Figure 9 we also mimic a screen dump showing an input form for some thesis and contact data. Such example data is extremely valuable for the requirements engineering task. Example data represented in forms that are familiar to the domain experts help those domain experts a lot to provide input to Software Stories and to explain which data they are dealing with. Similarly, the example data helps the software engineers to derive a data model for the desired software. The software engineers will most likely use some UML class diagrams to specify that data model for later implementation. While such class diagrams are very valuable for the software engineers and developers, domain experts usually do not understand class diagrams very well and they will not be able to spot faults in the class diagram design. Thus class diagrams will not help to clarify details and to resolve misunderstandings. Example data shown to the domain experts in a familiar way does a much better job. Deriving the formal data model is done by the software developers in a later step, easily. In our experience, example data contained in Software Stories is the most valuable part of a software story and thus we strongly suggest that Software Stories should always contain a lot of example data. Example data also helps you to find the right level of abstraction for your Software Story. If your Software Story does some kind of top down refinement, you may start with Software Stories like "Monday Morning, Albert tries to run SE Group Uni Kassel" containing Story Steps like "Albert manages David's thesis" which are quite coarse. When you refine such complex Story Steps to more detailed activities, frequently the question arises whether we have reached a sufficient fine-grained level of abstraction or whether we should still go on refining the steps. To our experience, as soon as you are able to give example data, you have reached the right level of abstraction.

5 Summary

Software Stories are a great means to do agile requirements engineering, analysis, design, and modeling with domain experts. Focusing on concrete examples helps the domain experts to explain and document their processes and by providing example data, the domain experts are able to provide great input for the data modeling step. To complement Software Stories during requirements engineering usecase diagrams may be used to group Software Stories and to give a general overview of the requirements. Thus usecase diagrams may help to structure the whole system while Software Stories explain the usecases in more detail. Thus, the usecases will help to structure the whole software development process and to group system functionality and to prioritize software development tasks. A thorough analysis of alternative scenarios may result in a larger number of Software Stories for a given System. Following the ideas of Story Driven Modeling [NJZ2013], Software Stories shall be turned into automatic (J)Unit [JUnit] tests. These Story Tests serve two main purposes. First a Story Test ensures that the described functionality is actually implemented and working at least for the example scenarios. And second, the Story Tests, help to ensure consistency across multiple related Software Stories. Without such a consistency check, multiple Software Stories may easily contradict each other on how a certain step is done and why a certain decision is made and which example data is used and stored in a certain step. By turning Software Stories into JUnit tests, the software engineers will identify such inconsistencies and they may revisit the domain experts to resolve such issues. Once you have achieved a consistent set of JUnit tests for your Software Stories, these JUnit tests ensure the consistency and completeness of all your requirements. Note, your final system will need more tests than just the Story Tests. You may also need a system design using component, deployment, and class diagrams. When designing algorithms, you may again use Software Stories or Storyboards for more fine-grained internal analysis. Such internal Software Stories may look much more technical and they may contain object diagrams or pseudo code fragments. Overall, Software Stories are just a great help for such requirements engineering, analysis, design, and modeling activities – especially in an agile development environment.

References

[Beck2000] Beck, Kent. Extreme programming explained: embrace change. addison-wesley professional, 2000.

[BPMN] Object Management Group Business Process Model and Notation. http://www.bpmn.org/

[FB1999] Fowler, Martin, and Kent Beck. Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[JUnit] Junit. http://junit.org/

[NJZ2013] Ulrich Norbisrath, Ruben Jubeh, Albert Zündorf. Story driven modeling. CreateSpace Independent Publ. Platform. ISBN-13: 978-1483949253. http://www.amazon.de/Story-Driven-Modeling-Ulrich-Norbisrath/dp/1483949257 2013

[Pohl2010] Klaus Pohl. Requirements Engineering: Fundamentals, Principles, and Techniques. Springer Publishing Company. ISBN:3642125778 9783642125775. 2010.

[RJB2004] James Rumbaugh, Ivar Jacobson, Grady Booch. Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education. ISBN:0321245628. 2004.

[Schwaber2004] Schwaber, Ken. Agile project management with Scrum. Microsoft press, 2004.

[TN1986] Takeuchi, Hirotaka, and Ikujiro Nonaka. "The new new product development game." Harvard Business Review (1986).