Experiences With an Industrial Long-Term Reengineering Project

Ralf Kollmann
BOSS AG
Lötzener Str. 3
D-28207 Bremen
ralf.kollmann@boss-ag.de

Abstract

In this paper, we discuss the experiences gained in a large-scale industrial reengineering project. The subject system is a medical data management software that has been continuously developed and maintained for about twenty years. About four years ago, it has been decided to subsequently reimplement the entire system in Java, while continuing maintenance of the legacy system.

The focus of this paper is less on technical details concerning reverse engineering tools, but rather on the overall approach for reengineering a large and complex legacy system. Central issues are the identification and definition of a suitable software development process, training for the developers, as well as the ongoing reverse engineering of the existing system. The latter has been carried out in parallel to continuous maintenance of both the legacy system and the reimplementation.

Keywords: Reengineering, Software Development, Legacy System, Industry, Data Reengineering, Java, Enterprise Applications, XP, UML.

1 Introduction

Legacy systems often manage to thrive far longer than the average modern software, for different reasons. Often, it is a matter of developing highly specialized systems only few experts truly understand. Neither is it easily possible to write them anew, nor are there many people who could do so. For these reasons, it is hardly possible to replace the system by standard COTS software. Also, the customers may have been committing themselves to IT technology at a rather early stage in computer development and the software has been integrated tightly into the company's processes over the years. In some cases, as e.g. in certain scientific environments, the ongoing progress in software technology has hardly any influence on the existing systems. However, in most commercially driven projects, software developers sooner or later face the need to replace their legacy systems with new software: be it because of increasing difficulties with maintaining the legacy system, or because of customer demands that are likely to be easier satisfiable with a modern programming language and development approach.

Most of these arguments do apply for the project discussed in this paper. It is about reengineering a large data management and administration system for medical information, which is employed by hospitals throughout Germany. The development of the system began in the early eighties, with development and especially maintenance continuing until today. The development team has been lucky to have a comparably low personal fluctuation, with some of the first-hour developers still being present. Therefore, detailed knowledge not only about the application domain, but also about the source code of the legacy system is still available. Maintenance of the old system is going to be continued for several years, until its complete functionality has been integrated into the newly reengineered system. Because of frequent changes in german health care laws, the software is subject to change on a regular basis. Deadlines for this kind of change are often as short as two or three weeks and are again required by law.

In the course of planning the long term future of the system, it has been decided to do a full reimplementation in Java as a client server architecture, with communication between client and database carried out via J2EE/EJB. The database itself had not to be redesigned, as a central requirement was to remain compatible with the legacy system: both the new Java application and the existing system had to run on the very same database, which consists of about 9500 tables with almost a million fields (logical views included). At the present state, the reimplementation in Java has a source code volume of about 1.8 MSLOCS.

This paper is organized as follows. In section 2, we discuss and motivate our different strategies for analysis of the legacy system. We then give a short introduction to the software development process employed by us and our experiences with it in section 3.

Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)
1095-1350/04 $20.00 © 2004 **IEEE**

To embed these methods into a greater scheme capable of providing long term perspectives not only for technical, but also for personal development, we shortly delve into the Capability Maturity Model (CMM) in section 4. Finally in section 5, we reflect on our experiences and central lessons learned. The paper ends with references to related work in section 6 and conclusive remarks in section 7.

2 Reverse Engineering vs. Requirements Engineering

Because of their varying future application, different strategies have been chosen to analyse the database and the legacy software. The proper approach towards reverse and requirements engineering had to be pondered, which had a significant impact on personnel requirements and team roles which we discuss in section 4.1.

Both the legacy system and the new Java system had to be maintained in parallel for a time span of at least five years. Changes to the source code of the legacy system would be limited to maintenance, especially concerning new laws.

2.1 Data Reverse Engineering

The database was determined to remain more or less unaltered in its legacy state, and became subject to examination by data reverse engineering only. Analysis sessions were generally initiated by a meeting with those developers most familiar with the data structures in question, to get an overview of the data structures and their representation in the database tables. Of particular value here was the developer's knowledge about database-specific caveats (e.g. intended redundancies, legacy data required to consider), which the developers had gained from their work, but which had not been thoroughly documented. Afterwards, it was usually required to get down to the data descriptions themselves. For this, several COTS database browsing programs were available, which also supported generation and reverse engineering of data definition statements. Additionally, a small analysis tool had been written in the beginning of the project, which was used to disentangle known semantic relationships between central database tables.

To decouple the database structure from the business logic layer, one of the first new implementations was a two tier framework for object relational mapping. The bottom tier provided standard conversions between database fields and Java data types, including a set of domain-specific types. The upper tier allowed the declaration of compound types and coherences between types at the domain level. The framework was subsequently expanded to also contain methods for generic construction of standard SQL statements. This approach saved huge amounts of time lateron especially for the business logic developers, whom it enabled in many cases to deal with persistence with hardly any involvement in database or EJB expertise.

2.2 User Interface Requirements Engineering

Having been written in RPG, the legacy software system hosted a huge number of user input masks. While the knowledge about the structure of the RPG code was broadly available within the RPG team, it was this part that would be replaced by the new system in its entirety. However, the opportunity was used to not only transfer the old functionality to Java, but to actively question the old functionality and especially its workflows. It was decided to disregard the RPG code completely and base the new user interfaces on requirements analyses of the old user input masks as well as newly set up user stories. The documented use cases and user stories were used both as basis for the design as well as for effort estimations. Two sources proved particularly valuable here: our hospital IT personnel, having knowledge of both the hospital domain as well as Java programming, came up with loads of ideas and requirements details. Additionally, a lot of constructive remarks originated from customer hotlines and especially from our beta customers. Many of them did not only help by finding bugs, but also had proposals for new functionality and ideas on how to improve existing functions.

Because of the large number of software modules developed concurrently by different teams, the desire arose quickly to maintain a survey of the relationships between them, as well as of the general progress towards project completion. To fulfill this need, regular presentations have been held. Apart from the developers themselves, feedback on usability was given by testers and stakeholders. For example, since many customers were strongly accustomed to keyboard usage, it became soon obvious that one of the primary usability requirements would be the establishment of consistent keyboard bindings: in addition to mouse based interaction e.g. by means of drag and drop, the new software was required to be usable solely by keyboard.

2.3 Effort Estimations

The application of standard cost estimation procedures like Function Point [1] and Cocomo [5] has proven appropriate only limitedly: these methods consider primarily the information flow of functions and focus more on the software than on the process and the development environment. In contrast, we considered it important to count in other, sometimes hard to grasp factors. For example, the analysis of existing programs and especially database reverse engineering revealed themselves as items resulting in considerable effort and especially a large uncertainty factor.

Although the responsible domain experts were generally present in the company, it occurred that their work had to be interrupted because of a seminar or an urgent hotline requiring their presence on site. When present, maintenance of the legacy system would often cause significant delays to the development plan. Further, most new developers still needed to invest a considerable amount of time to get into the data structures. This issue got less important however, with time passing.

One advantage of our kind of reverse engineering project was that the largest part of the required set of use cases was well-known in the team, which avoided an over-lengthy analysis (or authoring of unnecessary user stories, respectively). Therefore, apart from use case analyses, effort estimations have generally been based on experience. So far, we did not do extensive meta analysis of our estimations, as the results were generally considered sufficiently precise.

3 Defining the Software Development Process

It became obvious rather early that we would aim for a compromise between different development processes. Many facts spoke for a disciplined, heavy methodology. However several developers, including the author, had made good experiences with XP in earlier projects. So the basic idea was to pursue an approach somewhere between heavy methodology and agile development, as inspired e.g. by [16] [7], and find a compromise between the required formality and the desired agility [6]. Although we did not actually follow the extreme programming methodology [4] in all of its details, those associated paradigms that appealed to us were taken over as far as possible. The goal was to integrate and apply agile methods, preferably from extreme programming, where possible.

At first, the developer team appeared to be too large to successfully apply extreme programming techniques in their purest form. We discuss our experiences on this matter in section 5.1. When it came down to collaboration between departments, a certain amount of formalism appeared unavoidable. In addition to non-formal talks, communication between development teams was encouraged by weekly meetings. Communication between developers and quality management occurred both informally (i.e. walking down the floor), as well as via official forms. The need for "heavy methodology" was most evident during the release phases (c.f. section 3.2).

3.1 Modelling Standards for Documentation and Communication

While the UML is seen by some as a graphical programming language, we understand it primarily as an efficient means for documentation and especially communication between developers. Most of the time, we have created diagrams in paper and pencil style. Two people would work together on a diagram, discussing while drawing (very much like pair programming). Important sketches have been captured with a CASE tool. In addition to paper and pencil, some people (including the author) used the CASE tool Poseidon CE [11] directly for diagram design.

In an effort towards collaborative design, we made quite good experiences with computer supported UML design: a small group of up to four people would join for a design meeting and bring with them a notebook and beamer. During design discussion, one participant would capture the results and decisions as UML diagrams and text documents and beam them to the wall, thus enabling everybody to view the current state.

However, we found that exaggerated use of diagrams should be avoided, as drawing too many UML diagrams appeared to cost too much time and therefore money. This is by no means an argument against the use of UML diagrams. We just noticed that a healthy balance between diagram-based design and implementation is appropriate. Otherwise, there may be a chance to get lost in details, and some developers actually observed a reluctance to start the actual programming (often argumented along the lines of, "the design is not yet detailed enough"). This feels a bit like the old waterfall process model, where implementation is not started unless the design is "complete". We decided to avoid this behaviour wherever possible in favour of a more iterative approach, with a sound balance between all development activities [15]. Apart from serving as documentation, diagram-based design and other forms of sketches have been used primarily if there was an apparent need to clarify ideas and eliminate misunderstandings.

During the evaluation and experimentation with different analysis methodologies, we also learned to clearly distinguish between volatile approaches for structuring of ideas and persistent documentation-oriented methods. In [7] for example, a common use of index cards is described as follows:

The decisions coming from [...][analysis and design] are typically written down on index cards which are discarded once they have been committed to code.

We chose to use UML diagrams and use case cards to capture central parts of architecture and behaviour [13][12]. Rather than discarding these documents, we set up an archive for them. Use cases and especially diagrams were considered part of the documentation, and often their value greatly exceeded the written word. It has shown that this approach was particularly useful when doing refactorings at a later time, where they allowed to quickly grasp and remember complex architectures and coherences.

3.2 Iterative-Incremental Development

Our basic project schedule had three major releases per year. The number of these had not to be changed because of the administrative tasks associated with each of them. For each release, a precise delivery plan had been worked out that consisted basically of these phases:

- Development phase (including unit tests, semi-automated UI tests and DB tests). The quality management team received interim releases throughout this phase.
- Quality assessment. A separate team that was not involved in development ran a well-defined suite of tests consisting of the aforementioned automatic tests as well as manual tests of handling, workflow and exceptional situations. Development of new functionality and handling of QM change requests have been performed in different CVS branches during the assessment by QM.
- Beta phase. Selected customers received a beta release for application under realistic conditions and gave feedback to the quality management team and developers. This included both bug reports as well as ideas for new or changed functions.
- Release. When all pending changes had been integrated, the final release was ready for delivery to the customers. The delivery deadline had been fixed in advance and had to be kept in mind all the time. Postponing the date was not an option.

The process took up some time and effort for administration, but we have made quite good experiences with it. Especially, we were able able to hold the administrative work at a constant level and add improvements between releases. To adapt the process to XP-style short iterations, we decided to introduce an internal schedule. The corresponding deadlines would be relevant primarily for the developers and testers, providing a kind of "Ariadne thread" through development. The upcoming tasks for each iteration were captured with project management tools by means of Gantt charts. This made it easier to order tasks chronologically, manage causal dependencies and continuously track the progress.

Often, the time frame for a single iteration is defined to be about two weeks (e.g. [17][7]). Because of the experiences already gained with the problem domain and the existing software, we learned that we were well able to plan some time into the future. Also, the existing release plan gave a firm time frame for finishing the next major version. We finally decided to use four week iterations as a smallest common denominator in the team, with the option to use shorter iterations for individual tasks.

During each increment, short interim meetings would be used to synchronize the schedule with reality and probably adjust it. At its end, a developer meeting was held to discuss the finished schedule. Based on the experiences from previous increments, a new schedule for the upcoming month was set up.

After gaining some experience, we became quite successful at estimating the amount of work we could finish in one increment, which helped to improve morale and especially the motivation to continuously engage in collaborative planning sessions. Increments were passed as interim releases to the quality management, giving them the chance to get up to date with the state of development. This way, the amount of manual testing to be performed during the final quality assessment phase could be significantly reduced.

3.3 Customer Involvement

Customer involvement is a crucial issue in most software development projects, which is addressed in extreme programming by means of an on-site customer. The tasks fulfilled by this role are manifold [9], with a central problem being the consolidation of different stakeholder needs. This is even more the case, when there is a large number of customers with individual, probably conflicting interests. In most situations, we have been able to manage the respective requests by means of a persistence-based configuration management rather than by different implementations.

The role of an on-site customer has been fulfilled by several developers, who either had been working in the IT departments of hospitals for several years, or had acquired a broad domain knowledge while working within the company. Generally, a tight communication between customers, developers, quality management and management has been aspired.

A valuable source of customer feedback came via our support hotline: apart from common requests, many customers would add individual requests for changes in the software. If fulfillable, they have either been implemented as standard part of the software, or as individual enhancement for the customer.

Currently, we consider the introduction of another way of involvement: apart from the usual beta release procedure, selected customers will be invited when certain mile stones have been reached in the development. The idea is to present the current state to get additional feedback, especially for improvements concerning usability and workflow. Also, based on requests from customers, an agreement has been made recently about customer involvement in a new certificate-based testing process. After successful internal beta testing, the software and a test protocol are handed out

to the respective hospital IT department. For each software module, a personal contact person from second-level support is assigned. After successful test by the IT department, again a test protocol is created and key users from operating departments are given an introductory seminar on usage, parametrisation and tuning of the software. The final test is performed by the operating departments and an acceptance report is created. Now, the training for all involved application users is carried out.

4 Personnel and Process Development

Extreme programming provides a vast range of practices for modern software development. To get the best benefit of these, the need for some kind of framework for introducing the development process and reflecting upon the development of the new skills was identified. Especially in the early phases, we encountered the need to introduce a broad range of new skills to many people in a short time frame. The desired level of knowledge and skill would increase continuously: for example for the RPG programmers, the initial focus was on learning the basics of imperative programming and object orientation, as well as the Java APIs. Once learned at a sufficient level, additional skills such as design patterns and refactoring would be added.

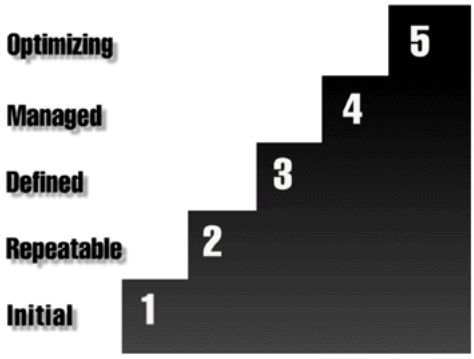

Figure 1. Maturity Levels of the CMM

We decided to use the *Capability Maturity Model* (CMM) as a base for a tailored skill development process. The CMM consists of different models for assessing the quality of a company's organization in a certain domain, as well as for giving guidance on how to improve it. The relevant models in our context are the "CMM Integration for Software Engineering" (CMMI-SW) [21], which is further subdivided into a staged and a continuous representation, as well as the *People Capability Maturity Model*® (P-CMM) [8]. The P-CMM can be seen as a special CMM for management and development of personal skills. Characteristic for CMMs is the definition of five (or six, depending on the point of view) distinct levels, each of which is is assigned to certain practices and requirements.

When using the capability levels of the continuous representation [20], levels four and five are considered rather theoretical: Although generally attainable, from an economic point of view, most companies might consider reaching level three sufficient.

It may be notable that our motivation for delving into the matter of process improvement was solely to evaluate and adopt (where appropriate) the practices of CMM-levels two and three, especially concerning the P-CMM. Although no official certification has been aspired at the current stage, achieving the aforementioned levels might become a medium term goal in the future.

4.1 Initial Training Measures

To suit the personnel needs of the new project, the following measures have been taken:

- Employing new Java developers
- Java training for already employed RPG developers
- Java training for already or newly employed application trainers and IT personnel with hospital experience

First of all, a score of new developers with profound Java experiences has been hired over a period of about 30 months. For these, training in the required medical background as well as the company's existing IT products has been arranged. A good understanding of the legacy systems was considered paramount, as these were the subject of reengineering for the new developers.

To better benefit from the extensive knowledge and experience of already present RPG programmers, some have been retrained to join the team of Java programmers. As expected, learning the new language as well as object orientation and further concepts like extreme programming took considerably more time to yield productive developers than teaching the basic knowledge of the hospital domain to a Java programmer. Further, the daily business generally required the former RPG team to remain involved with their previous occupations for quite some time. To start productive developing, the Java developers would need about two to three weeks of preparation, in contrast to a time span of about three to four months for the retrained RPG developers. The decisive difference between both approaches was of course the way of getting things done. Even with a certain background in medical applications, the Java team would have to apply common analysis techniques in cooperation with in-house domain experts. Regularly held meetings (at intervals of one to four weeks) would interrupt the periods of development. For the RPG team, the

approach was actually reversed: while having detailed domain knowledge, the major problems were rather of technical nature. Pair programming with experienced Java programmers proved to be a good way to address this issue. This way, both groups could benefit from each other, with the Java developers learning more of the domain, while the former RPG team would efficiently get familiar with and learn the programming language.

As the legacy system was strongly database-centered, the members of the RPG team with their detailed knowledge of the database structures were primarily assigned for database programming, for which they had to learn J2EE techniques in addition to standard Java.

The third group of Java developers also came with a detailed knowledge of the hospital domain, but from a very different perspective: both application trainers and computer personnel hired from hospital IT departments had their focus on user interfaces and the requirements from application end users. This group was exposed to similar efforts of learning the programming language and environment, as was the RPG team.

4.2 Team Level Workforce Practices

P-CMM level two propagates the introduction of workforce practices at the team level (rather than organisation wide). As an example, we have applied this approach when introducing unit testing with JUnit. First, a seminar on unit testing was offered to all interested developers (or those yet unfamiliar with the topic). While everybody was encouraged to experiment with JUnit, a single team has been selected to use unit tests consistently as central part of development. As JUnit is easy to learn and comfortably integrated into our IDE, it did not take much time until the first success reports came in. Then, in subsequent steps, the seminar has been repeated to cover all developer teams. Developers already familiar with the topic have been used as knowledge multipliers to facilitate the first steps for the others. By now, unit tests have been adopted as a mandatory practice in all teams.

4.3 Continuous Learning

Learning is a lifelong process. This may be even more so for computer science with its fast-paced technological advances than for most other subjects. While sometimes frowned upon at school, we found that the opportunity for learning can be a strong motivator in business life. Accordingly, in addition to the need to keep the developers up to date, continuous training was provided to raise the motivation and thus increase productivity.

Apart from the basic training mentioned above, we established different forms of knowledge propagation: seminars, open forums and digital archives. The seminars were divided into two categories. Some were inevitably required by everyone working in the respective business, featuring topics such as understanding of hospital structures and workflows. Technical issues included e.g. IBM iSeries fundamentals. The second group of seminars, although strongly encouraged, had a more optional character. Here, central topics addressed for example improvement of object oriented development skills as well as introduction of more advanced concepts like refactoring, design patterns and “best practices”.

Arising questions, solutions to solved problems and new findings were discussed in the weekly developer meeting, with important issues having been archived digitally. A knowledge database with a web interface was used to store documentation and slides, while a mail archive was set up for storage of important mails.

The acceptance of the seminars was usually very good, and often, additional dates were required to give everyone interested the chance to attend. Also, people came around asking questions and giving feedback on the seminars. Concerning the discussion forum and especially the knowledge base, resonance was rather feeble. We noticed that comments came usually from the same small circle of “very interested persons”. Many developers disregarded the knowledge base, and some did not even know where to find it, despite regular announcements. In the attempt to explain this behaviour, we suspected that the increased effort of self-study might have had a discouraging effect, especially in combination with the lack of a direct channel for clarifying questions. Although all articles in the knowledge base (let alone archived mails) were labeled with an author tag, the readiness to ask questions seemed far greater, when a contact person was immediately available. For us, this was another evidence for the importance of personal communication.

4.4 Practices on the “Defined” Level

Beyond level two (“Repeatable”), the focus in P-CMM is on the development of an organization-wide infrastructure on top of the established workforce practices. At the current stage, we have focused on the practices discussed above, leaving this issue for future development.

5 Lessons Learned

While we learned many things during this project so far, there are some issues which seem outstanding concerning their relevance to its outcome:

- Selection of an appropriate methodology
- Acknowledgment of the importance of communication

- Ensuring the propagation of the chosen methodology in the team

We discuss our experiences with and attitude towards these points in this section.

5.1 Agile vs. Heavy Methodologies

Especially concerning release deadlines and release management, it became very obvious to us how developing in a large team requires a solid amount of discipline from everyone. Not to the least, this can (and had to) be enforced by putting up conventions that have to be abided by. Thus, one of the premier lessons was to accept the demand for certain degree of formalism when working in the large team towards a release date. This goes to a good amount into the direction of the quality management paradigm of being able to accomplish the same task repeatedly with the same (or with improving) quality. In small flexible teams, the story was entirely different. Communication ways were extremely short, often nil in effect. Within such an environment, agile methods like pair programming and collaborative planning could be applied well and efficiently.

Many approaches have shown how it is possible to integrate agile and heavy methodologies (e.g. [16], [7]). Because of the different attitudes towards developing in small teams and global coordination, we went for a hierarchical approach. While application of agile methodology in general has not been enforced, authoring of unit tests was considered mandatory for all developers. Further, our selection of XP methods (including unit tests, refactoring, pair programming and collaborative planning) has been applied successfully and with great effect by several developer teams.

5.2 On the Importance of Communication

Communication between different developer teams occurred usually over the same channels as within a team (i.e. informal talks and meetings). Additionally, the communication between developer and quality management department has been done via standardised email forms (e.g. progress notification and request for testing).

An author whose name I do not remember wrote once that the success rate of IT projects tends to be inversely proportional to the communication distances of the team members. The experiences made in our project more than affirm this statement. Actually, in some situations it has been noticeable how even the distance to a bureau down the corridor or on a different floor subtly reduced communication and sometimes indirectly increased development times.

An obvious consequence was to order the offices in such a way that people working on the same or similar subjects are situated in close proximity.

5.3 The Hundredth Monkey

Being based on a myth, the hundredth monkey is a metaphor that represents the hypothetical one element in a group which makes it reach a critical mass. Being based on psychology and communication, this metaphor plays a significant role in the introduction and acceptance of new approaches or techniques in a large team.

Of course, everybody is going to ponder whether a newly introduced methodology is useful for their own work and if it is worth the time and effort to get familiar with. However, it can be observed that once a certain number of people have made positive experiences with it, the propagation is often carried out considerably faster. People seem to trust the experiences others have made and are easier convinced to come aboard, resulting in a kind of snowball effect. A similar effect can be observed in marketing, where certain products fill only a niche position until a certain volume of propagation is reached. Then, propagation may increase explosively.

The relationship to marketing is not so farfetched. After all, the technology promoters aim at “selling” their new idea to the colleagues in the team. Apart from the general benefit, they gain some reassurance that their work is actually appreciated by the other team members. This requires at first that there are developers who are motivated to delve into and quickly grasp new technologies and their possible advantages. On the other hand, it requires the other team members to be open for innovation and willing to leave the well-trodden paths. Such activity, if carried out focused and without going out of hand, is a bless for every developer team and should therefore be encouraged. When a clear view on the subject has been acquired and it has been decided that the technology is worth adopting, the experiences can be made public in a presentation or at a meeting. Once again, communication is paramount. The ”pioneers” probably know already about the most prominent caveats and trip wires, thus flattening the learning curve for others.

A word of caution seems appropriate, though. If this kind of prospecting is carried out in an uncontrolled manner and without tight communication between all involved, it can easily happen that knowledge already present in the team escapes ones attention, resulting in superfluous efforts.

6 Related Work

Related studies have been made e.g. on the integration of heavy and agile methodologies, as well as on industrial reverse engineering experiences and analysis of maintenance processes.

Van Deursen et al. compared software development based on extreme programming and maintenance of legacy

systems, and examined whether the application of XP techniques is useful in legacy maintenance [22].

Lucia et al. applied statistical control techniques to a software company's massive maintenance processes and analysed the data of an empirical study carried out in the effort. Focus is on the data about the single phases of the process and the distribution of effort between them [2].

Jeyaraman et al. reengineered a large legacy system into a web based J2EE system, using a variant of the Rational Unified Process (RUP) [10]. Here, too, the SEI CMM standards have been considered.

Antonini and Canfora adjusted a legacy system to meet metrics-based quality standards by means of reengineering techniques and automated tools [3].

Concerning projects focusing on the technical aspects of reverse engineering in an industrial environment, Moise and Wong performed a reverse engineering case study on an industrial-scale software application [18] using the Rigi system [19].

7 Summary and Conclusion

In this paper, we have compiled our experiences made with a large-scale reengineering project, whose goal was to completely reimplement a twenty-year-old legacy system in Java.

After giving a survey over the initial situation and motivation, we have discussed the road to take, especially concerning the chosen methodology. From a software developer's point, a central issue of the project has been the consolidation of heavy methodology and agile methods. Because of the large number of persons involved, it has been unavoidable to introduce a certain degree of formalism in the project. Although we had a large amount of informal communication between departments, there still existed "official" channels, e.g. for handling and tracking of customer support requests.

We have learned that a certain formality has been essential for quality management, enabling us not only to repeat successful projects and thus maintain quality, but also for learning from mistakes and thus improve the existing processes.

On the other side, we have learned how to work with agile methods in small teams, and how to successfully employ them in larger teams built out of smaller ones.

Sadly, we did not find the philosopher's stone of IT in the methods employed by us. However, by combining the advantages of different approaches, we established a solid software development process that enabled us to successfully proceed on the double road of reengineering and reimplementing a large legacy system, while maintaining both old and new system at the same time.

Acknowledgments

I would like to thank my employer BOSS AG, who supported this work and made it possible to share the experiences gained in our project in this way.

Thanks to the anonymous referees for their constructive comments, which helped to improve this paper.

References

[1] A. J. Albrecht. Measuring Application Development Productivity. In *GUIDE/SHARE: Proceedings of the IBM Applications Development Symposium (Monterey, Ca.)*, pages 83–92, 1979.

[2] S. S. Andrea de Lucia, Antonello Pannella. Empirical Analysis of Massive Maintenance Processes. In T. Gyimóthy and F. B. e Abreu, editors, *Proc. 6th European Conf. Software Maintenance and Reengineering (CSMR'02)*. IEEE Computer Society, 2002.

[3] P. Antonini, G. Canfora, and A. Cimitile. Reengineering Legacy Systems to Meet Quality Requirements: An Experience Report. In Hausi Müller and Mari Georges, editor, *Proceedings International Conference on Software Maintenance (ICSM 1994)*, pages 146–153. IEEE Computer Society, 1994.

[4] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[5] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[6] B. W. Boehm and R. Turner. *Balancing Agility and Discipline*. Addison-Wesley, 2003.

[7] G. Booch, R. C. Martin, and J. Newkirk. *Object Oriented Analysis and Design with Applications*. Addison-Wesley, 1998.

[8] B. Curtis, W. Hefley, and S. Miller. *The People Capability Maturity Model: Guidelines for Improving the Workforce*. Addison-Wesley, 2002.

[9] A. v. Deursen. Customer Involvement in Extreme Programming. *ACM SIGSOFT Software Engineering Notes*, 26(6), 2001.

[10] V. R. G. Jeyaraman, Kumar Krishnamurthy. Reengineering Legacy Application to E-Business with Modified Rational Unified Process. In *Proc. 7th European Conf. Software Maintenance and Reengineering (CSMR'03)*, pages 143–. IEEE Computer Society, 2003.

[11] Gentleware AG. Poseidon Community Edition 2.0, 2003. `http://www.gentleware.de`.

[12] R. Kollmann. *Design Recovery Techniques for Object-Oriented Software Systems*. PhD thesis, University of Bremen, Department for Computer Science and Mathematics, 2003.

[13] R. Kollmann and M. Gogolla. Metric-Based Selective Representation of UML Diagrams. In T. Gyimóthy and F. B. e Abreu, editors, *Proc. 6th European Conf. Software Maintenance and Reengineering (CSMR'02)*, pages 89–98. IEEE, Los Alamitos, 2002. Best Paper Award.

[14] R. Kollmann, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In E. Burd and A. van Deursen, editors, *Proc. 9th Working Conf. Reverse Engineering (WCRE'02)*. IEEE, Los Alamitos, 2002.

[15] P. Kruchten. From Waterfall to Iterative Lifecycle: A tough transition for project managers. Available at Url `http://www-106.ibm.com/developerworks/rational/library/content/Rational%Edge/rosearchitect/ra_spring2000.pdf`.

[16] C. Larmann. *Applying UML and Patterns*. Prentice Hall, 1997.

[17] R. C. Martin. *eXtreme Programming Development through Dialog*. IEEE Computer Dynabook, 2000. `http://www.computer.org/seweb/dynabook/DevThruDialog.htm`.

[18] D. Moise and K. Wong. An Industrial Experience in Reverse Engineering. In *Proceedings 10th Working Conference on Reverse Engineering (WCRE 2003)*. IEEE Computer Society, 2003.

[19] U. of Victoria. Rigi, 2003. `http://www.rigi.csc.uvic.ca`.

[20] Software Engineering Institute (SEI), Carnegie Mellon University. Capability Maturity Model Integration for Software Engineering (CMMI-SW, V1.1), Continuous Representation. Technical report, 2002. `http://www.sei.cmu.edu/publications/documents/02.reports/02tr028.html`.

[21] Software Engineering Institute (SEI), Carnegie Mellon University. Capability Maturity Model Integration for Software Engineering (CMMI-SW, V1.1), Staged Representation. Technical report, 2002. `http://www.sei.cmu.edu/publications/documents/02.reports/02tr029.html`.

[22] A. van Deursen, T. Kuipers, and L. Moonen. Legacy to the Extreme. In *Proceedings of the first International Conference on eXtreme Programming and Flexible Processes in Software Engineering - XP2000*, 2000.