Could an Agile Requirements Analysis be Automated?—Lessons Learned from the Successful Overhauling of an Industrial Automation System

Thomas Aschauer[1], Gerd Dauenhauer[1], Patricia Derler[1], Wolfgang Pree[1], Christoph Steindl[2]

[1] C. Doppler Laboratory Embedded Software Systems, Univ. Salzburg
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at
www.cs.uni-salzburg.at

[2] Catalysts GmbH,
Prager Str. 6, 4040 Linz, Austria
steindl@catalysts.cc
www.catalysts.cc

Abstract. This paper sketches a recent successful requirements analysis of a complex industrial automation system that mainly required a talented expert, with a beginner's mind, who has been willing to dig into the domain details together with a committed customer and a motivated team. With these key factors and the application of an appropriate combination of well-established and some newer methods and tools, we were able to efficiently elicit, refine, and validate requirements. From this specific context, we try to derive implications for innovative requirements analysis. We argue that in projects that go beyond simple, well defined, and well understood applications, automated requirements analysis is unlikely to lead to a successful specification of a system.

Keywords: requirements analysis, agile development, use cases, automation systems

1 Introduction

Our research group cooperates with an industry partner that is a dominant player in the area of a specific kind of test automation systems that are used, for example, in the automotive industry. These automation systems need to be tailored to customer demands. For the software solution our research partner currently offers, this tailoring process is not supported well. Thus we were asked to develop a system that radically improves the customization and operation process of such systems.

The inherent complexity of the domain and the vagueness of the original requirements document we were provided with were major challenges for the requirements engineering process. We chose an agile, prototype driven approach with

short feedback cycles. In conjunction with an unbiased team, which consisted of a top software scientist and four motivated software engineers, we were able to successfully elicit and analyze the requirements and to come up with an innovative solution. We are confident that it is able to solve the current system's shortcomings and to sustainably improve our partner's competitive advantage.

The main contribution of this paper is twofold. First it presents a successful requirements analysis process for an industrial innovation project. Second it argues that in this particular case automatic requirements analysis methods were not applicable. As such it serves as a reality check for natural language processing methods in requirements analysis.

The remainder of this section briefly introduces the target domain and gives a short overview of the customization process in the current system. Section 2 describes the project context, the initial requirements and the team structure. The actual requirements analysis process and the development of the prototypes are described in section 3. Section 4 presents a case study on how the team's understanding of one particular requirement grew over time. Section 5 concludes that automated methods for analyzing requirements are not likely to have succeeded for this particular project setting.

1.1 The Domain of Test Automation Systems

This section briefly introduces the application domain of test automation systems. Typically, a test system is used to acquire measurement data from operations of a device under test. The resulting data is required, for example, for research and development or for quality assurance. Various variants of test systems are used in industry.

An automated test system typically comprises the following parts: a device under test (or device for short), automatic test equipment (or equipment for short) that simulates force, a mechanical link between the device under test and the automatic test equipment for force transmission, measurement equipment ranging from simple temperature sensors to sophisticated measurement devices, actuators such a throttles, I/O systems as interface to an automation system that controls the test procedure, and conditioning devices controlling supply for air, oil, water, etc.

This system structure is depicted in Figure 1: Boxes represent hardware components, the block arrow represents the mechanical link, solid lines represent electrical connections between components, and the dotted line represents media supply for air, oil, water, etc.

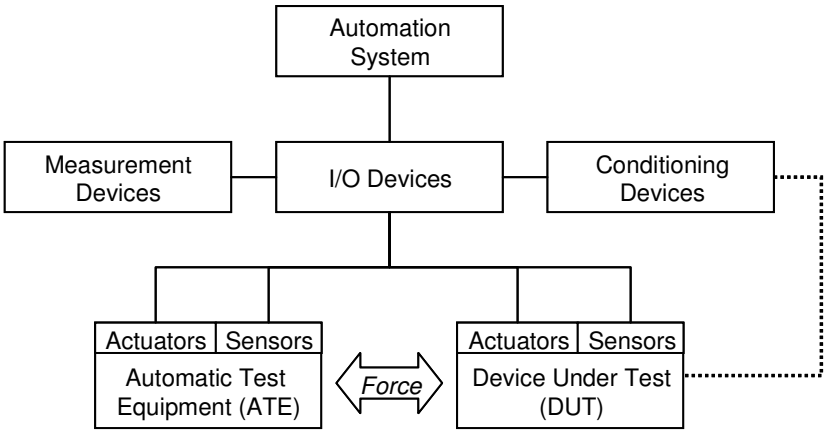

Fig. 1. Typical automated test system structure.

A typical test procedure for an automated test system has a duration ranging from minutes to hours or even days. During that time, up to millions of measurement values are recorded, which can amount to several gigabytes of measurement data. The automation system's software is responsible for controlling the device and the equipment in real-time, for executing test procedures, and for collecting and recording the measurement data. Evaluation of the measurement data is performed by separate post-processing tools.

An automated test system can be operated as stand alone system or in a larger context, the so-called *test-factory*. A test-factory is a set of separate automated test systems, with possibly different capabilities, that share common infrastructure such as measurement data archiving. The overall goal of a test-factory is to optimize the throughput by scheduling test orders accordingly.

1.2 Problems of the Current System

Our work is based on the cooperation with a dominant player in the field of test automation systems. Our research partner offers a software solution that can be applied to all kinds of test systems through customization to specific requirements. The software evolved over the last two decades during which its code base, mainly written in C++ and C, grew to about 1.5 million lines of code.

The automation system software consists of a number of specialized subsystems such as a hard real-time kernel executing the device under test and the automatic test equipment, and a subsystem for measurement data acquisition. The interactions between these subsystems are established through variables in globally shared memory. Thus the subsystems have to be configured consistently. Due to the evolution of these subsystems, they all use their own configuration file formats for customization, ranging from plain text files to binary files.

Configuration parameters describe properties of hardware and software. Properties of hardware are, for example, the device's weight. Software properties described in

configuration files include the characteristic values for the equipment's controller, the safety limits for the force affecting the device, device driver settings for certain measurement devices, or user defined formulas and scripts to be executed by the automation system.

A major hurdle for users that have to customize the system is that the current tool directly reflects the automation system's software structure and low-level design decisions in the user interface. Moreover, all parameters are presented in a tabular form. The main view basically is a plain table, where each table row represents the associated subsystem of the automation system and is used to navigate to a detailed view for the subsystem. In other words, the configuration interface is presented as a set of interrelated spread-sheet pages.

As an example, the automation system has a separate subsystem for proportional-integral-derivative controllers (PID-controllers). When a user needs to modify the parameters for a specific controller of the equipment, the user would need to navigate to that subsystem, browse through all PID-controllers in the system to find the right one, and then modify the corresponding parameters. This customization system forces engineers that are used working with the parts of the test system such as the device, the equipment, etc. to understand the internals of the software to be able to set parameters correctly.

In the customization process, engineers have to modify parameters in configuration files that are loaded by the automation system at system startup. In a typical setup there are about 10,000 configuration parameters with about 120,000 values to be set correctly. The creation of a consistent set of configuration files is a time consuming and error prone procedure which may take weeks or even months for a complex test system setup.

The current systems offers very limited support for the initial creation of these configuration files. Only the skeletons representing the structure of the test system can be created by a tool, the details have to be filled in manually. The time it takes to get the system running depends on the experience of the engineers in charge. They often use a form of ad-hoc reuse by using configuration files from previous similar projects as templates.

Once a set of configuration files is created, it has to be kept synchronized with the automation system software. When software is updated during the lifetime of a test system, its configuration files have to be modified accordingly. Again, tool support for the update process only barely exists. Updating configuration files has to be performed manually. As a consequence, customers update their automation system software to major revisions only when absolutely necessary, because the process of modifying configurations files is time consuming and error prone. Our research partner therefore has to invest a lot of development effort in the maintenance of many different software revisions in parallel.

2 Project Setup

Our research partner identified the necessity for improving the current customization process. Due to the fact that multiple tries to overcome the current system's

shortcomings by the company itself have failed for different reasons, a research project in cooperation with our research institute was initiated. This section briefly describes the initial requirements and the project team structure.

2.1 Initial Requirements

The main mission goal is to develop a system that radically improves the usability of customization and operation of test automation systems. In the beginning, we were provided with a rather haphazard requirements document consisting of about 20 items. The list includes specific functional requirements as well as some general non-functional requirements such as maintainability and security issues. The most important requirements are summarized as follows:

a) Introduce components, i.e. named sets of parameters, that naturally map to domain entities such as device under test, automatic test equipment, PID-controller, etc. and that describe both their visualization and their parameters.
b) On-site extensibility, meaning that new functionality can be added to the system without the need to recompile any source code.
c) Provide different parameter views including guidance through customization tasks. The basic idea is that of separation of concerns [1], meaning the splitting of various aspects of a system into independent parts that can be dealt with independently. As an example, there should be a separate view for hardware-related parameters, such as the weight of the device, and another separate view for software-related parameters, such as the characteristic values of the PID-controller for the equipment.
d) Support users in mastering the complexity of test system setups, e.g. by hiding those parameters that are not needed for a specific task. For example, a service task concerned with finding the defect part between the automation system and a certain device does not require knowledge about the simulation model for the device.
e) Provide a context-aware work environment that supports the user in specifying only valid parameter values for a component by evaluating the component's context.
f) Do as many checks as possible as early as possible. Inconsistent measurement and consumption frequencies, for example, can be detected by comparing parameters of connected components when the connection is established, whereas the existence of a piece of hardware in a test system can only be checked when the system is connected to the actual test system.
   Furthermore, ensure that these checks can be integrated in different products to avoid duplicated implementations.
g) Replace configuration files by parameter sets, i.e. by components.
h) Provide an *operations view* describing a component's visualization and the parameters that are modifiable during the operation of a test system.
i) Compatibility to existing systems, which means supporting a wide range of tools and technologies.

j) Maintainability of components, which means support for versioning, change tracking, comparison, and interoperability between different systems and also between different software versions.

In addition, we also received a huge amount of user documentation, system requirements specifications for the existing system, and UML diagrams. The latter consisted of use case diagrams and use cases describing functionality at the level of specific technical details. These documents evolved along with the existing system during the last two decades. They were, however, hardly up to date.

2.2 Project Team and Location

Due to the importance of the project for our customer, the company is fully committed to it and we report to one of its executives. The project is set up around one of the company's most respected experts, who is also fully committed to the project goals. The project leader has more than one decade of experience in the domain and long time experience in successfully managing projects of comparable complexity, including innovative software development projects. Later on we realized that this particular project leader is like an *advocate* for the project, in the sense as Wile described knowledgeable advocates as crucial for the success of their domain specific language experiments [2].

Company representatives with in-depth domain knowledge as well as product managers were available in the requirements analysis phase. Additionally, we had access to employees that formerly were associated with competitors and also to developers of the current system.

The initial software development team consisted of one top software scientist as team leader, and four young software engineers with little or no project experience. The team leader has extensive software development experience, social skills training, and an additional solid background in automation systems, but had no prior knowledge of the particular automated test system.

During the course of the project, the team grew in size by two software developers and two domain engineers with background in automation systems and the target domain.

The project team intentionally resides at a different geographical location than our partner, which emphasizes the company's intention to strike a new path in the development of their software solutions.

3 Prototyping-based, Agile Requirement Analysis

Considering the ambiguity in the provided requirements document (cf. section 2.1), the fact that the team had no prior knowledge of the domain and the overall vision of the project seemed somewhat unsettled, the right methods for the requirements engineering task had to be chosen.

We decided to stick to an agile approach for the following reasons: First, the short feedback cycles would allow us to quickly respond to changes in the requirements and

to misunderstandings of the original requirements document. As stated by Hirsch, "the desired properties of the end product can not be known until at least part of the solution is built" [3]. Second, the project leader's intuition gave him the feeling that for an innovation project, a front-up design method would not lead to success. Third, the team leader had previous, successful experience in applying agile methods.

This section chronologically describes the project phases, beginning from the initial phases of paper prototyping to the current phase. In addition, the planned project phases are sketched to depict the different approaches necessary in the different phases.

3.1 Phase I: Paper Prototyping (September 2006 – February 2007)

Since the project team was completely new to the domain, we started the project with a 5 day workshop. We approached the problem from the user's point of view, first developing a global context with the user roles and their targets, then detailing the tasks of the users – completely unrestricted by the existing system. During the first workshops we looked also at systems from three competitors.

We wrote down the discussions in detailed workshop protocols, and we visualized the scenarios on slides, some with animations so that they resembled how a system could actually work. Some of these presentations were prepared from one workshop day to the other, so that we could start with a recapitulation of the previous day, and extend on it.

Figure 2 shows a conceptual drawing for how a *perfect parameterization system* would show the physical parts of a sample test system. Basically, boxes represent components which are pieces of hardware or software that are connected to other components. Concepts such as grouping, abstraction by hierarchically structuring components, and different ways of connecting components were applied and refined using these drawings.

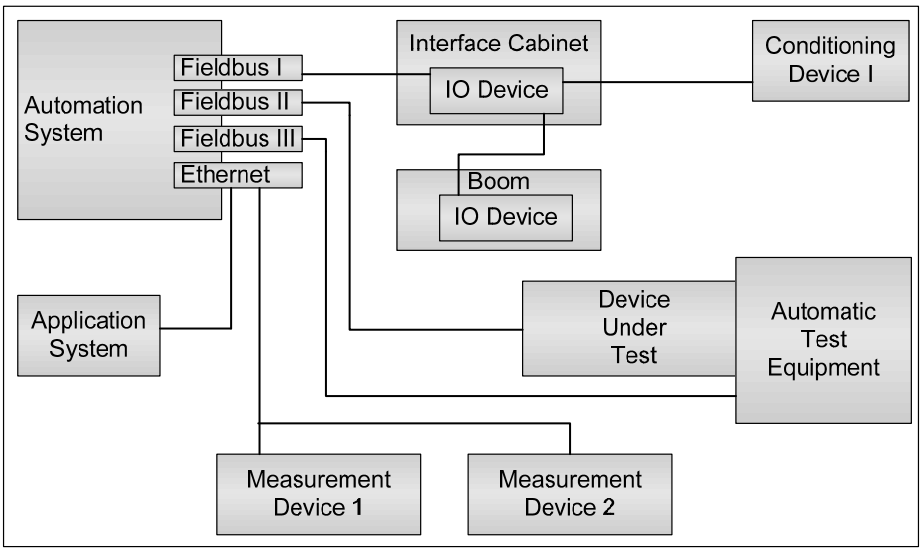

Fig. 2. Conceptual drawing showing the physical parts of a test system.

We held several workshops in a row, with approximately a month in between, which gave us time to understand the existing system. Thus we were able to conceptualize the requirements step by step. This process was documented by writing a glossary comprising about 90 terms as well as by analyzing and writing some 130 use cases. At that time the project focus to develop a system that would eventually replace the parameterization tool of the current software solution was clearly communicated to the team.

Due to the radical departure from the original system, we knew that we had to present the ideas in an easy-to-grasp way; hence we decided to develop a mock-up prototype that would allow showing how various users, in their various roles, would use the system. For that we specified scenarios such that we could exactly define the click paths through the prototype for every user. Numerous concepts and ideas were proposed and discussed in simple drawings on paper, in slide presentations and figures drawn with common drawing tools. These drawings exemplified how the software could appear for each scenario.

Similar to the drawing in Figure 2, the mock-up prototype provided a view representing the physical components of a test system. Figure 3 shows the corresponding screen.

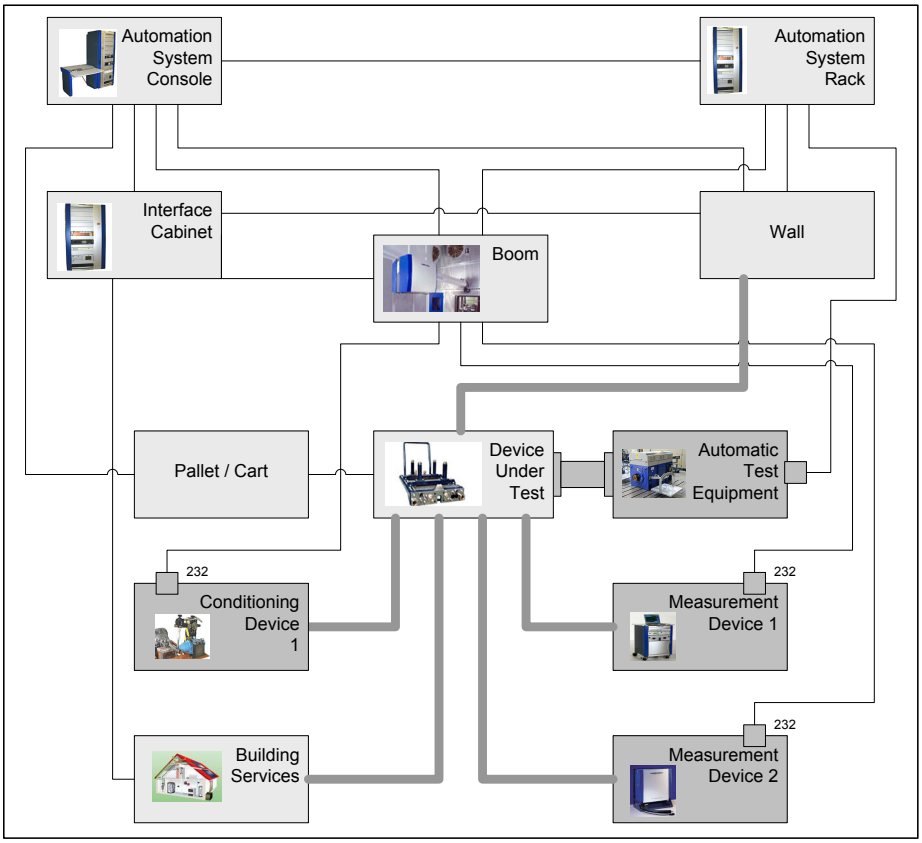

Fig. 3. Physical view of test system in mock-up prototype.

The development of the mock-up prototype served as a vehicle to document and elicit customer requirements and to gain domain understanding, such that we were able to derive *16 core features* which represent the essence of the system's functionality. Each of those core features was meant to be orthogonal to the others; together they yield a powerful system to solve the underlying problem. The most essential core features are summarized as follows:

- Definition of the concept of domain components. Domain components are defined as sets of parameters that are grouped into self-contained units.
- Support for configuring domain components, that is, setting their parameters or connecting them to other domain components.
- Support for combining domain components in groups or hierarchies. By allowing domain components to be built hierarchically, that is, by layering component systems as described by Szyperski [4], complexity can be managed. Collapse and expand mechanisms help hiding unnecessary details.
- Support for management of domain components in libraries, which might be predefined by our customer or be user-specific.

- Support for comparing domain components and helping users to discover similarities and differences.
- Support for versioning of domain components.
- A mechanism for undo and redo management and for creating and recording macros at the user interface layer.
- Provide a mechanism for guiding users performing predefined tasks or for resolving problems. Furthermore, allow users to provide their own experience in form of guidance for other users.
- Management of different views of domain components and corresponding access rights.

For the actual demonstrations we decided to have one person clicking through the prototype, while another person would do the talking, watching the audience, being able to answer questions and to improvise, and to lead the questions back to the click paths that we had prepared beforehand. Eventually the team presented the prototype, which was enthusiastically received by the customer's top management in January 2007.

We captured the demonstrations as video sequences with a couple of introductory slides and an animated demo part. Those videos had several advantages:

- They allowed everyone to get some insights about what the project was about. With the prototype, getting these insights would not have been possible, since the prototype required in-depth knowledge of the prototype's implementation and the predefined click-paths. Only a small percentage of click-paths a typical user would do were implemented.
- They allowed us to preserve the presentations, so that we ourselves could have a look at them later on, e.g. when new people were to join the team.
- They allowed us to explain the system without having to conserve the executable or without installing the executables; hence it was easier to share.
- They forced us to get the user stories right and consistent. We had to remove all vagueness from the ideas.

However, the videos also had some drawbacks, e.g.:

- Even though we asked for feedback on the last slide shown in the videos, we did not get valuable feedback from the customer.
- We were told not to distribute information about the project in this format any more, since some of the customer's employees spent much time on them. Moreover, the videos caused disturbance in the current system's development team, since this was one of the first sources of information about our project that was made available to them. As Ramos et al. point out [5], the introduction of radically new software and the vision of a future work reality associated with it is never free of emotions.

The system to be built was split in two layers, a generic framework layer and an application layer built on top of the framework. The framework, developed by our team, provides a platform for building custom applications that can be developed by application engineers with profound domain knowledge, but without programming skills. While the framework incorporates generic domain knowledge, such as sensors,

measurement values, etc., the application incorporates specific domain knowledge such as the types of equipment applicable for a specific device and test system.

During the first project phase, the software development team gradually gained understanding for the customer's demand of a component-oriented framework for test automation systems. The different interpretations of the term *component framework* were one of the major causes of confusion between the customer representatives and the development team: The development team had components in mind as defined in software science, that is, components describing a unit of composition with contractually specified interfaces and explicit context dependencies only. Such a software component can be deployed independently and is subject to composition by third parties [6]. Furthermore, the development team had a *technical* component framework in mind, in the sense of e.g. the OSGi platform [7], while the customer representatives thought of a framework for modeling and assembling components specific to the *domain*, such as DUT and controllers. Domain components are somehow related to software science components, that is, they are units of composition, they explicitly describe dependencies, and they are units of deployment. For the further success of the project it was crucial to overcome this misunderstanding.

3.2 Phase II: Working Prototype Based on a Domain-Specific Language (March 2007 – September 2007)

Early in the project, we considered a domain-specific language (DSL) as crucial basis for describing test system components. Motivated by our project leader, the DSL was designed as a generic one for describing automation systems. This means that the DSL offers, for example, a means for describing data types of values, the construct of a generic component for grouping values and also for grouping associations between these components. The language for describing test systems is an extension of the generic one. We refer to the generic language as CDL (Component Description Language) and to the test-system-specific language as tsCDL.

For the refinement of CDL and tsCDL we applied an informal approach. Starting in April, we evaluated tools and methodologies that would best fit this task. We first used the Unifid Modeling Language (UML) syntax [8] to sketch and iteratively refine the key test automation system concepts such as electrical plugs, wires, mechanical connections, sensors, and actuators. It turned out that a simple UML diagram drawing tool with limited UML capabilities was better suited for this purpose compared to a full-featured UML editor.

In addition to the UML-based CDL and tsCDL refinement, we came up with a textual representation of CDL and tsCDL. We used both, the UML-based and text-based versions of tsCDL to describe test system components such as devices, data acquisition units and also complete test systems. The definition and refinement of the textual syntax of the language and the UML-based version were intertwined.

The following code fragment in Figure 4 sketches the description of a test automation system. The sample test automation system consists of three components, the automatic test equipment *Ate*, an I/O hardware called *IO*, and the automation system called *AuSys* . The component *Ate* is of type *AteType127*, the component *IO* is

of type *IODevice*, and the *AuSys* component is of type *AutomationSystemPC*. The test automation system description also states which locations are relevant in this case: a location called *Floor3* and a *ControlRoom*. The second line in the *RELATIONS* section harnesses the location description by stating that the *Ate* component is located on *Floor3*. The *Ate* component is hierarchically composed of other components, such as the *BendingBeam*. The *BendingBeam*'s plug *Plug2* is connected to plug *X17* of the *IO* component, which is specified in the first line of the *RELATIONS* section.

```
COMPONENT TestSystem
  COMPONENTS
    Ate   : AteType127
    IO    : IODevice
    AuSys : AutomationSystemPC
  END
  LOCATIONS
    Floor3
    ControlRoom
  END
  RELATIONS
    Ate.BendingBeam.Plug2 CONNECTS IO.X17
    Ate AT Floor3
  ...
  END
END
```

Fig. 4. Sample test automation system described textually in the tsCDL.

Splitting the domain-description language in a generic one (CDL) and a test-system-specific one (tsCDL) is an example of an architectural aspect that could not be derived as requirement from the information we received from the customer. Nevertheless, this extra effort of coming up with both description languages turned out to be crucial for other system parts that rely on them. For example, we only needed to implement an interactive visual editor for CDL, not the much richer tsCDL which is constantly changed and extended. This is also true for the persistence layer for storing and retrieving domain components. A key objective for the project was the compatibility between components stored in different software versions. Another key objective was that software upgrades must not result in costly database schema migrations. As such, the persistence format has to be stable and should not change often during the further evolution of the software. Our experiments corroborated that we only need to define a database schema for the CDL, not for the the tsCDL.

The design of CDL and tsCDL as well as the development of the interactive visual editor, the persistence mechanism for CDL components and their versioning were inherently difficult to plan. Initial attempts to establish a development process, such as Scrum [9], were abandoned since the estimated efforts turned out to be unrealistic and the process became an overhead without any benefit.

The major milestone of Phase II was the development of a prototypical first version of the software system incorporating the most essential features of the 16 core features identified in the previous phase.

3.3 Decompression Phase III (September 2007 – February 2008)

After presenting the results of the second phase to the executives of our customer, the team entered a short *decompression phase* [10], which means the team performed a retrospective to improve subsequent phases. The retrospective revealed the following key success factor: Defining concrete scenarios helped to focus the development of the prototype. Furthermore, the scenarios had to be defined in detail, so that all vagueness had to be eliminated and the concepts had to be sound and understandable from the user's point of view.

The following factor has been identified as restraining to the project success: Due to the inherent complexity of the application domain and its peculiarities, the team depends on a variety of information sources. We did not consistently question the quality and the completeness of the information we got.

3.4 From Research Prototype to Product (Starting February 2008)

The software system has reached a level of maturity so that domain engineers can use it to model real-world test system components. The foundation, based on CDL and tsCDL, is stable and additional features are continuously integrated enabling domain engineers to model the various aspects of test system components as they are found in test system products. The goal of a milestone in August 2008 is to demonstrate that the software system is capable of modeling, configuring and operating a real test system. The long-term plan is that the newly developed software system will be shipped as product to customers in 2010.

The additionally required features are derived from the feedback of the customer's domain engineers. We have established a bi-weekly release cycle now. The release planning incorporates the requests of domain engineers in the form of user stories, describing the expected behavior in terms of the user interface. These stories are usually a few lines of text and the effort to implement them ranges from one person-day to about one person-week.

These requests of domain engineers represents one source for our release planning. The other sources are the initial requirements as presented in section 2.1 and the refactorings suggested by the development team itself. We treat the identified refactorings of the existing code as user stories.

4 Case study: Understanding the Versioning Requirement

We exemplify how our understanding of the requirements evolved over time by picking one of the 20 initial requirements which we consider as a representative example. The initial list of requirements contained the following text, which was summarized as the last bullet-hole item in section 2.1:

«12. Maintainability: it must be possible to version parameters and parameter sets; Change logging, i.e. who changed what and when; Export/import among test fields, also language independent; Search/find; Difference of parameters and

parameter sets; Undo; Interoperability of previous software versions with data in newer version and vice versa»

We were quite aware that this key requirement was intentionally phrased quite vaguely, for example, the "and vice versa" phrase. Therefore, we tried to de-scope some of the requirements for the initial project Phase I:

«12. Maintainability:…»

« ➔ we will propose a concept for the operation until the end of 2006
➔ the domain model and meta-model will allow for versions
➔ since the implementation would require major changes to the existing system, we won't perform them until the end of 2006»

So for a while we turned back to the more challenging requirements and developed concepts, prototypes etc. as explained in section 3.1 above. One of the 16 core features identified in Phase I was the following:

«13. Updating of components with a transport mechanism for changes:
It is possible to deliver application components in a new version and deploy them. Macros can be used as transport mechanism for changes.
There will be a language for describing:
- *How old data shall be migrated*
- *Whether the new version must be deployed or can (optionally) be deployed*
- *Whether user interaction / acknowledgement is necessary or whether the update shall be performed silently*

Updating application components is not about updating software, but about updating descriptions of components (together with the underlying data).
Updating of system functions would require a software update which we do not address in the first release. »

Even then we thought that updating would "simply" mean that we need some flexible mechanism to get data of older versions migrated to the schema of the new version. During a workshop with another project team of the customer in February 2007 they presented the following requirements or conclusions:

«'Import mechanism is enabled to do needed data migration'
'Migration Framework is a MUST!'
'Be migration aware'
'Versioning - Implemented within our storage services'»

We took those statements again as hints that we will only need to import old data in new versions of the software. We acknowledged the need for a migration framework and versioning but deferred the topic nevertheless, believing that we will also be able to implement it in the persistence layer with some import / export filters.

In March 2007, we augmented the requirements with use cases. We identified the following use cases:

- UC Versioning 1 – Select a version
- UC Versioning 2 – Browse version log
- UC Updating 1 – Define data migration
- UC Updating 2 – Perform data migration

However, we sketched only the main scenario for the versioning use cases, and left the updating use cases undefined. Back then, leaving everything open was the best we could do, since any detail would have been speculation.

At the end of March 2007, we had an architecture workshop with the customer where the development manager of the existing system mentioned that the new system will have to sustain the concurrent operation of automated test systems in multiple versions. We considered it sufficient if our software were able to cope with new and old versions of the data.

In Phase III at the beginning of December 2007, we discussed the topic in detail with our advocate. The discussion was summarized with the following versioning requirements:

- In a test field, multiple test systems will be in use with various versions of the new software system.
- The new software system must be able to process old and new components.
- It shall be possible to migrate components in old versions to newer versions, such that test systems with new versions of the software system can use the old components.
- If possible, it shall be possible to use the new components even on old test systems, possibly just in a read-only mode.

We discussed the implications of those requirements on the various layers of the system and how changes in each layer would affect upper layers. Analogies from books on database refactoring were drawn, e.g. the idea of *scaffolding code* in the database, which transparently enables one version of the software to work with several versions of the data model. As described by Ambler [11], this can be achieved by introducing views and triggers in the database layer. Furthermore, we drew analogies from related scientific papers, dealing for example with the problem how to co-evolve a model when the corresponding meta-model evolves, as described by Wachsmuth [12].

We identified two principal approaches to deal with version changes: to track all transformations, i.e. a priori, versus to derive modifications from delta detection, i.e. a posteriori. The latter approach was ruled out by construction of examples that showed its deficiencies.

However, we still did not really accept the need for bidirectional compatibility, i.e. that new versions of the software can work with old and new data, *and* that old versions of the software can work with old and new data.

At the end of December 2007, our advocate kept pushing towards bidirectional compatibility. In January 2008, we finally accepted the challenge of bidirectional compatibility and gave it a try, i.e. we did a so-called *spike* in eXtreme Programming terminology [13]:

- We refined the implementation from the user's point of view.
- We implemented the solution, which required several extensions of the persistence layer and upper software layers.
- We demonstrated to the customer how data model transformations can be defined and how a new version of the system can then automatically transform data from the old format into the new format. Furthermore we demonstrated how an old version of the system can automatically transform data from the new format into the old format, given that a bidirectional mapping between old and new meta-model exists.

Summarizing the case study,

- we considered versioning and updating as a black box for a long time

- we ignored repeated hints by the customer, or we did not understand them
- we placated our advocate for a long time

Finally, we worked through the problem within three calendar weeks, and we came up with an appropriate solution for a problem that the customer has had for decades but that resisted several previous attempts to be solved. In the end, all the extensions did not have a negative impact on the existing architecture. We think that it would have been impossible to derive the requirements of this aspect from documents we received from the customer.

5 Limits of Automated Requirements Analysis

This real-world project corroborates, in our point of view, that requirements analysis can barely be automated if the stakeholders do not have a clear understanding about a software system. In this case it was the feeling of the customer that the current system could be improved significantly. The customer and its team were somehow trapped in the existing system. Knowing too many details and worrying about significant changes made it virtually impossible to come up with appropriate requirements for an overhauled system. The required creativity cannot be expected from tools. To quote Deming [14]: "As a good rule, profound knowledge comes from the outside, and by invitation. A system cannot understand itself."

The beginner's mind [15] allowed the team to profoundly analyze the features of the current system as well as its strengths and weaknesses. This is a quality already pointed out by Berry [16]. He describes a computer-system-savvy person without any knowledge of the domain as the person asking ignorant, not stupid, questions to expose tacit assumptions made by domain-expert stakeholders assuming incorrectly that all other domain-expert stakeholders understand. By making those assumptions explicit, conflicts in the understanding are discovered at an early stage in the software development.

5.1 Could Automated Support for Requirements Analysis have been Beneficial?

Reflecting on potential use of automated approaches to requirements analysis, we identified two areas where application of such approaches could have been beneficial in our case: term extraction and preventing ambiguity. For a recent overview of state-of-the-art approaches to requirements engineering in general see Cheng and Atlee [17].

Since the project team was completely new to the problem domain, automated support for extracting the domain specific terms could have been applied. As Kof points out [18], a thorough understanding of domain concepts is essential and a precise definition for each concept is required. An approach to semi-automatically extract ontology from requirements documents is proposed. Such an approach or similar ones are, however, likely to have failed in our case for the following reasons:

- As pointed out in section 2.1, the initial requirements document we received consisted of only 20 items that just briefly described the system to be built. Domain specific terms occurred in the document, but due to the document's limited size the usage of a semi-automated or an automated tool for term extraction is not likely to have produced substantially better results than performing this task manually. In the paper prototyping Phase I, as described in section 3.1, we created a glossary for the essential domain concepts.
- Along with the initial requirements document, we also received a huge amount of documents related to the current system, such as requirements specifications and user documentation. When the team sifted through these documents, it soon became obvious that most of the information was not relevant in the early project phases. It still is in question whether the majority of the material will be of any use at all since it deals with specific technical details and peculiarities. Applying a system for term or ontology extraction on these documents would have been a challenge on its own due to the size of the documents. It is not clear how such a system could have helped in the decision which concepts to ignore, and which not to ignore, in particular if one keeps in mind that the number of essential concepts is very small compared to the overall number of concepts.

  For example, an automatic analysis of the documentation would likely have identified the *normname* as one of the most relevant concepts in the domain just by the number of references. Normnames are, however, just a necessity of the current system's implementation: A normname is the unique name of a variable in the global shared-memory which is used to connect the different functions and subsystems, as mentioned in section 1.2. These global variable names are one major shortcoming of the current system that we could get rid of in the new system.
- Important concepts of the new system were completely missing in the current system; they were only described by general terms in the initial list of requirements. The versioning and compatibility requirement as described in section 4 is an example. Term extraction techniques would not have been helpful for understanding these requirements either.

Another area where application of natural language processing tools would have been conceivable is in preventing ambiguity in the documents we generated. For example, Fantechi et al. [19] present an approach that analyzes use cases written in natural language and provide certain metrics for measuring aspects related to ambiguity. These might have improved the consistency of the use case documents we created in Phase I as described in section 3.1. Because these use cases were not the final specification of the system to be built, but just a vehicle to further understand the requirements and to structure the problem domain, fewer ambiguities in these documents would have just been a minor benefit.

For a project of this type, i.e. searching for a creative and revolutionary solution, the successful application of automated techniques is unlikely. Typically, the customer would not present a fully specified requirements document and expect a development team to return a working program after a certain amount of time, within a predefined budget. Instead, in an iterative process with frequent workshops, demonstrations and presentations, the customer can see how the project is evolving

and how the team performs. Moreover, the team can gradually gain better understanding of the customer's *real* demands.

5.2 Conclusion

We assume that none of the tools that automate requirements analysis could lead to a successful completion of the requirements analysis for our project, because the available inputs from the customer are too haphazard and the terminology is not precise enough—a situation that is typical for many real-world software projects. In such a context the sketched agile requirements analysis with short feedback cycles together with the communication vehicle of a throw-away prototype has turned out to be an appropriate requirements analysis method. We are convinced that no automated system would have been able to support, let alone accomplish something close to such a successful requirements analysis and specification based on the available natural language descriptions of the requirements, the current system and its envisioned features.

References

1. Dijkstra, E. W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs, New Jersey (1976)
2. Wile, D.: Lessons Learned from Real DSL Experiments. Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03), IEEE Computer Society (2003)
3. Hirsch, M.: Moving from a Plan Driven Culture to Agile Development. Invited talk at ICSE '05, The 27th International Conference on Software Engineering, St. Louis (2005)
4. Szyperski, C.: Component software and the way ahead. In Foundations of Component-Based Systems, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, New York, NY, 1-20 (2000)
5 Ramos, I., Berry, D. M., Carvalho, J.: The Role of Emotion, Values, and Beliefs in the Construction of Innovative Work Realities. In Soft-Ware 2002: Proceedings of the First International Conference on Computing in an Imperfect World. Springer-Verlag, London (2002)
6. Szyperski, C., Pfister, C.: Workshop on Component-Oriented Programming, Summary. In Muehlhaeuser, M. (ed.): Special Issues in Object-Oriented Programming – ECOOP96 Workshop Reader. Heidelberg: Dpunkt Verlag (1997)
7. OSGi Alliance, Open Services Gateway initiative , http://www.osgi.org/.
8. UML, Unified Modelling Language, http://www.uml.org/.
9. Rising, L., Janoff, N. S.: The Scrum Software Development Process for Small Teams, IEEE Software, vol. 17, no. 4, pp. 26-32 (July/August, 2000).
10. Gamma, E.: Agile, open source, distributed, and on-time: inside the eclipse development process. Keynote talk at ICSE '05, The 27th International Conference on Software Engineering, St. Louis (2005)
11. Ambler, S. W., Sadalage, P. J.: Refactoring Databases : Evolutionary Database Design (Addison Wesley Signature Series). Addison-Wesley Professional (2006)
12. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In Ernst, E. (ed.) Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07). LNCS vol. 4609, Springer, Heidelberg (2007)

13. Beck, K.: Test-driven development: By example. Addison-Wesley Publishing (2002)
14. Deming, W. E.: The New Economics for Industry, Government, Education - 2nd Edition. MIT Press. ISBN 0-262-54116-5 (2000)
15. Suzuki, S.: Zen Mind, Beginner's Mind, Weatherhill (1973)
16. Berry, D. M.: The Importance of Ignorance in Requirements Engineering, Journal of Systems and Software (1995)
17. Cheng, B. H., Atlee, J. M.: Research Directions in Requirements Engineering. In 2007 Future of Software Engineering, International Conference on Software Engineering. IEEE Computer Society, Washington, DC (2007)
18. Kof, L.: Natural Language Processing: Mature Enough for Requirements Documents Analysis? In Natural Language Processing and Information Systems, 10th International Conference on Applications of Natural Language to Information Systems, Alicante, Spain (2005)
19. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: pplication of Linguistic Techniques for Use Case Analysis. In Proceedings of the 10th Anniversary IEEE Joint international Conference on Requirements Engineering. IEEE Computer Society, Washington, DC (2002)