Using XP to develop a CRM framework

Hubert Baumeister

Abstract— **This paper describes our experiences with using XP practices within the EU-project CARUSO. The objective of CARUSO is the development of a framework for building customized Customer Relationship Management (CRM) applications. Originally, the project was planned with a traditional software development process in mind with a first prototype for evaluation by the customer and a second prototype building on the results of that evaluation. However, problems occurred defining the requirements for the framework for several reasons. First, our prime customer had only a vague understanding of how the software support for their CRM needs should look like, and second, CRM involves almost every business process in a company. To address these problems we used an agile software development process that allowed us to start from a simple CRM process (customer service), dividing it into user-stories, and clarifying the requirements on the framework as the user-stories were implemented.**

I. Introduction

In particular in the domain of e-commerce, having robust software systems satisfying the needs of the business, is getting more and more crucial to the survival of companies. In addition, when starting the development of a software system, usually one has only a rough idea of the final functionality of the system while, on the other hand, the software is needed already yesterday. Classical, heavyweight software processes, which first require a thorough analysis of the requirements and a detailed design before implementing, fail to deliver in time. Software that takes several years to design and implement may find themselves in a situation that it cannot cope with the current requirements of the company, or even worse that the company who initiated the software development does not exist anymore. To cope with these kind of problems, agile software development processes, like Scrum [12], Crystal [6], FDD [5], DSDM [7], and others have been proposed [1]. A quite recent agile method which has gained a lot of popularity is Extreme Programming (XP) developed by Kent Beck, Ward Cunningham, and others [3], [10]. XP is a lightweight process which incorporates methods to react to change while not sacrificing the quality of the resulting software. XP is most suited to small and middle-sized projects where the software has to adapt to changes in the requirements and the environment, and where the software needs to produce business value even if not all functionality is implemented. An example of this type of projects are e-commerce web-sites. It is important that the web-site is up and running already quite early to start making money, even if only a minimal functionality is present. While the web-site is running, new functionality is added and the requirements for the web-site are refined and changed due to the feedback from the customers.

We have used practices from XP in the CARUSO project [2], [4]. CARUSO is an EU-funded project [8] with the objective to design and implement a framework for building customized Customer Relationship Management software. The major problem with designing such a framework is finding the right components and their functionality because the requirements on CRM software are quite complex as they involve all the business processes of a company, like marketing, sales, service, etc., and all its IT systems.

We first started the project following a classical software development process which required us to analyze a good deal of these processes before starting the design of the system [13]. Because of the complexities involved in CRM this proved to be impossible.

Therefore, starting from a rough idea of the CARUSO architecture, we defined user-stories based on the CRM needs of REMU, a utility company and one of the partners of the project. During the implementation of these user-stories, the components of the framework and their functionality was discovered and implemented.

One of the components of this framework is the script engine which manages the execution of dialog-scripts. Dialog-scripts guide the dialog between a call-center agent and a customer by presenting the agent with text and questions she should ask the customer. Though more precise, the general requirements of the script engine were too many to be dealt with in a suitable time frame. Thus the problem was to decide which subset of the requirements were the most important in the context of the CARUSO project. By looking at the user-stories of the CARUSO project we were led to the definition of suitable user-stories for the script engine. In addition, the script-engine was implemented using test-first programming.

In the next section we describe the CARUSO project and the architecture of the framework in more detail. Sections III and IV show how XP practices were used in the CARUSO project. Finally, Section V provides a conclusion.

II. CARUSO

The CARUSO (Customer Relationship Support Office) project [4], [2] is a research and technological development (RTD) project funded by the European Union within the Information Society Technologies (IST) program of the 5th framework program [8]. Partners are REMU, a Dutch utility provider in Utrecht, DataCall, a German software house in Munich, and the Institute of Computer Science of the Ludwig-Maximilians-University in Munich. The project started in January 2000 and ends June 2002. The objective of CARUSO is to provide customized Customer Relation-

H. Baumeister is with the Institute of Computer Science, Ludwig-Maximilians-Unviersity (LMU), Munich, Germany. E-mail: baumeist@informatik.uni-muenchen.de .

This research has been partially sponsored by the European Union within the IST project CARUSO (IST-1999-12192).

ship Management (CRM) solutions for small- and middle-sized enterprises. This is achieved by designing and implementing a framework for constructing CRM applications. This framework consists of a set of generic components together with tools to customize these components.

A. Architecture

The basic design consideration of CARUSO was to build the framework from components. Microsoft's COM/DCOM component technology was chosen because Windows is the target platform for CARUSO and the framework should reuse and extend existing COM/DCOM components. The architecture of the CARUSO framework has five major parts:

- the kernel components
- interfaces to back-office systems
- front-office applications
- application builder tools
- administration tools

The kernel components provide the basic services to all front-office applications built with CARUSO. These components are:

- Communication Server
- Storage Manager
- Business Object Manager
- Script Engine
- Service Manager

The communication server is one of the central parts of any CRM application. Ideally any communication with the customer will be done using some of its services. In particular, the communication server manages the routing of incoming and outgoing messages, like phone calls, e-mails, faxes etc. Each incoming message will be sent to the most skilled agent depending on a variety of factors, e.g. which number the customer called (i.e. which service he requested), who the customer is (can be inferred from the calling telephone number), and what his contact history is. In addition, the communication server can be used to make outbound calls and it supports marketing campaigns using pre- and power-dialing.

The storage manager defines an abstraction layer on top of common relational databases so that an application programmer does not have to deal with the peculiarities of a particular relational database. On top of the storage manager functionality, the business object manager provides access to the business objects of an application by applying a user-defined mapping of these objects to relational database tables.

Finally, the script engine is used to run dialog-scripts guiding the dialog between a call-center agent and a customer. The script engine will be discussed in more detail in the next section.

For each back-office systems that the CRM application has to interface with, like ERP systems, workflow-management systems, etc., a software-component representing the back-office system needs to be implemented. The task of this component is to access the data stored in these systems, but also to initiate business processes involving these systems. The advantage of using components representing the back-office systems instead of directly calling the API functions of the legacy system is that all the methods that need to know about the API of the legacy system are grouped together in one place, which makes it easier to adapt these components to changes to the back-office system (like new versions of the software).

The CARUSO kernel together with the interfaces to the back-office systems provide the components used to build particular front-office applications. These are built with the help of the Application Builder Tools. These tools include a Data Modeler to define the business objects used by the business object manager, the Script Developer for developing dialog scripts, and tools to administer and monitor the resulting CRM applications.

B. Script Engine

A dialog-script guides the dialog between the call-center agent and the customer. It guides the agent through a set of questions and texts to be presented to the customer. The sequence of questions and texts depends on the answers a customers gives. In addition, to each transition from one question or text to another one can associate arbitrary actions, like updating databases or sending messages to other agents.

An example is the script used by REMU for changing the amount of monthly pre-payment for a customer's utility-bill. After the usual introduction, identification who is calling, and finding out about the service request, that is, that the customer wants to change the monthly pre-payment, the agent is first presented with the following question on his screen:

„*Your current monthly payment is [payment]. What should be your new monthly payment?*"

In this question [payment] is replaced by the actual monthly payment of the customer, which is retrieved from the customer-database. Then the agent types the answer of the customer on his keyboard. If the new payment is greater then the original payment or not less than 90% of the original payment, this new payment is accepted by the system without further questions and the customer database is updated with the new payment. The agent then is presented with the text:

„*Thank you, your new monthly payment is [payment].*"

In this example, we assume that the dialog is finished at this point, although, more likely, the agent would ask the customer if he could do something else for her.

In case that the new payment is less then 90% of the original payment an explanation is needed. Thus the agent asks:

„*The number you have given is too small. Please give an explanation.*"

The agent types the answer given by the customer and automatically this answer is forwarded by e-mail to some person in the back-office evaluating the request. The agent is presented with the following text to end the dialog with the customer:

„*Thank you, your request will be considered.*"

The task of the script engine is to execute given dialog-scripts. It keeps track of which questions have been asked and what answers were given, what the current question or text is, and performs actions when moving from one script item to another.

One of the design goals was to separate the logic of how scripts are executed from the user interface used to execute these scripts and from the storing of the answers. In particular, different programming languages were used to implement these different aspects. The engine itself was written in Java (Visual J++) as a COM component. One user interface was written as an ActiveX component using native Windows widgets, while a web-interface was implemented with the help of Java Server Pages. To process and store the answers given to the questions in a script, the business object manager and storage manager components were used.

III. Requirements Paralysis

Our first approach was to use a more traditional software development process. Two prototypes of the software were planned. The first prototype was scheduled after the first 18 months and should be evaluated by REMU. The result of this evaluation was intended to drive the second prototype which was scheduled for the next 9 months. Each of the two prototypes were planned according to the traditional development cycle: analyzing requirements, designing, implementing, and testing the system.

A major problem with this approach was that for the first design of the system we had the tendency to look for a complete set of requirements to start with, because any requirements that were not considered in the first design could require changes to the design and implementation that would be too expensive to do in the later phases. In CARUSO we first tried to model the business processes that would be influenced by a CRM software. This was not feasible as CRM usually has to interact with almost every business process of a company and each company has different kind of business processes. Further, it proved to be very difficult to find a common data model for a customer suitable for several companies even in the same vertical market. Also, REMU had only an imprecise understanding of their CRM requirements. REMU was referring to CARUSO as their customer care dream. As is common with dreams, CARUSO was supposed to do everything; but because of the complexity of CRM, no concrete requirements were given, since nobody knew where to start. This resulted in the problem that to start with the design and implementation of the system, we needed to have precise requirements, while, on the other hand, the real requirements would only be known when a first version of the system was available to gain some experience.

The way XP addresses this problem is that analysis, design, implementation, and testing is done for each user-story in turn without taking those user stories into account which have not yet been implemented. The result of these steps is a system implementing exactly this user-story. This allows for immediate feedback by the customer. Each new user-story is dealt with the same way. Not taking into account all user stories that have not yet been implemented is important as the user-stories may change because of the experiences gained with the resulting system.

A. User Stories

Therefore, within the CARUSO project, we first focused on the business process most important to REMU, which was customer service. This included support cases like complaining, getting information about products, and changing customer data. One of the most important, but also most complicated support case was the moving from one place to another by the customer.

So the first step was to build a small pilot to show REMU how these service requests could be handled. While this pilot could show some sample screens, it did not yet implement any serious business logic. However, it proved sufficient for REMU to produce a set of support cases they want to have handled and to define how these should be handled.

- User-story 1: Identifying Customer
- User-story 2: Change Billing Address
- User-story 3: Change Monthly Pre-Payment
- User-story 4: Handle Complaints
- User-story 5: Handle Requests for Information
- User-story 6: Move In / Out

Each user-story added to the functionality of each of the software-components. At the end of the last user-story, a first prototype of the CARUSO framework was available.

B. Example: Script Engine

Using the script engine as an example, we show how each of the iterations guided the design of the scripts and the script engine. To handle the support case of the first iteration, identifying customer, no script was necessary. In the support case for the second iteration, changing the customers billing address, the script consisted of a simple question and processing its answer without any branching. The support case for the third iteration, changing the monthly payment, involved branching on conditions and performing actions, like sending an e-mail to the back-office. Furthermore, parameters like `[payment]` had to be introduced into the text of the question. These parameters were replaced by their actual value during the execution of the scripts.

For the support case of the fourth iteration, handling customer complaints, we discovered that at several points in the script it was necessary to schedule the visit of a technician at the customers house. This involved asking several question which could be considered as a script of its own and led to introducing scripts as part of other scripts. Also we noticed that the first question of a script may depend on information REMU had about the customer. For example, if the customer had a complaint about district heating and has a service agreement with REMU, then immediately a technician would be scheduled to visit the customer. However, if the customer does not have a service agreement, further questions would be asked and she would be referred to, for example, the house owner. This led to the introduction of a script item which could be used as the first item in a script and as source for branching but would not be displayed on the screen.

The support cases of the last two iterations did not require any further extensions to the scripts and the script engine.

IV. Test-First Programming

To achieve the robustness required of the script engine, automated tests and test-first programming were used. This ensured that each each functionality had its associated test and that everything that we want the software to do was document as a test. This also helped in better understanding the functionality to be implemented, because the test made precise the functionality we expected.

We wrote tests for:

- Intended Functionality
- Assumptions About the Code
- Border Cases
- Discovered Bugs
- Interaction between COM/DCOM Components

Test for intended functionality and assumptions about the code are quite similar. However, the test for intended functionality tests for the results the code should produce if everything is okay. Testing assumptions about the code may also document failures, for example, what happens if a function gets passed a wrong argument. While this probably shouldn't happen at all, in some cases it is important to document what would happen. Other assumptions on code include unexpected behavior (whether correct or incorrect) of library components.

Writing the test for the border cases, e.g., if an argument to a method is null and similar cases, made precise (and documents) what should be the result of such situations.

Bugs were an indication that we forgot to test and implement some functionality. Further, tests for bugs ensured that later revisions of the software did not introduce the same bug again.

One major problem was to understand the interaction between COM/DCOM components written in Visual Basic and Java. One of the user interfaces was written in Visual Basic while the script engine was implemented in Java. Therefore access from Visual Basic to methods and objects in Java was needed. Problems occurred with how data types in the Visual Basic were mapped to data types in Java; in particular, how values of type Variant in Visual Basic were mapped to values in Java. Tests were important to document our assumption on how this mapping works.

In addition, also access from Java to other COM/DCOM components was required, as all the data manipulated by the scripts were handled by objects outside the script engine COM component. For example, all data gathered by the script engine was stored using the business object manager components. Again tests were written to document and test our assumptions.

Usually one adds functionality in a way that keeps the old design untouched to ensure that the old functionality still works. However, this results in duplicated and complex code. Therefore, when adding new functionality to the script engine, we changed the old design to produce the most simplest design that implements the old and new functionality. This approach requires to re-implement some of the old functionality using the new design. The tests helped us assure that we correctly re-implemented the old functionality.

Tests helped us improve the portability of the script engine. While intended to be used as a COM component in Windows, we wanted to use the script engine also as a pure Java application to maintain platform independence. Thus a first version of the engine was developed under Linux. When moving from Linux to Windows, tests showed us that almost everything works with the exception of a few tests related to reading and writing scripts in XML. Investigations showed that these failures were related to the different line end conventions of Unix and Windows.

A more subtle problem occurred when moving from one computer running Windows 2000 to another computer running the same operating system. All tests passed but one. The failing test revealed a broken library we distributed with the script engine. The computer on which the development took place used a correct version of the library instead of the broken one. Because of having the tests we found the bug which otherwise might have been discovered only at the customers site where fixing this bug would have been quite expensive.

The code size of the tests equals almost that of the production code, 42 classes with 5.111 lines of production code versus 37 classes and 4.837 lines of test code. JUnit [9] was used to test the Java part and the connection from Java to other COM components, and VBUnit [11] was used to test the connection from Visual Basic to Java.

V. Conclusion

In this paper we have presented our positive experience with some of the XP practices in the context of the CARUSO project, although not all XP practices could be applied because of the distributed nature of the project (as common with EU-projects, the project partners were from different countries) and political reasons. It proved very helpful to divide the development task into user-stories guided by the CRM needs of REMU to get a precise understanding of the requirements of the framework and to get feedback on its use.

Similarly, this approach helped the implementation of the script engine, which otherwise would have taken much longer to design and implement, because we would have taken into account a lot of sensible requirements which were not needed for CARUSO.

Although no big design phase preceeded the implementation of the engine, the design proved quite stable with respect to future requirements. While within the CARUSO project the design of the script engine reached a stable state, the engine is being extended at the moment to cope with requirements coming from outside of CARUSO. A company needs an implementation of dialog-scripts involving forms in addition to plain questions. It showed that these new requirements could be implemented with only minor modifications, although no particular effort was made to ensure that the design of the script engine was able to cope with future changes.

Also, the engine is quite stable and only a few bugs were found since the engine is in use. We believe that this is due to the automatic tests and due to the fact that tests were written before the actual code. Writing the tests also helped us discover problems when moving from one platform to another and even when moving from one computer to another running the same OS. We think that without the tests it would have much harder to find and deal with these problems.

References

[1] Agile Alliance. The agile alliance. www.agilealliance.org, 2001.

[2] Hubert Baumeister and Piotr Kosiuczenko. CARUSO: Customer Care and Relationship Support Office. In Ricardo Gonçalves and Adolfo Steiger-Garção, editors, *Product and Process Modelling in Building and Construction, Proceedings of the Third European Conference on Product and Process Modelling in the Building and Related Industries, Lisbon, Portugal, 25-27 September 2000*, pages 115–120. A. A. Balkema, Rotterdam, Brookfield, 2000.

[3] Kent Beck. *Extreme Programming Explained.* Addison Wesley Longman, 1999.

[4] CARUSO Consortium. CARUSO web-site. www.caruso-crm.uni-muenchen.de, www.caruso24.com, 2001.

[5] Peter Coad, Eric LeFebrve, and Jeff De Luca. Feature-driven development. In *Java Modeling in Color with UML.* Prentice Hall, 1999.

[6] Alistair Cockburn and Jim Highsmith. Crystal methodologies. www.crystalmethodologies.org, 2001.

[7] DSDM Consortium. Dynamic systems developement method. www.dsdm.org, 2001.

[8] European Union. Future and emerging technologies initiative global computing. www.cordis.lu/ist/fetgc.htm, 2001.

[9] Erich Gamma and Kent Beck. JUnit. www.junit.org, 2001.

[10] Ron E. Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed.* Addison Wesley Longman, 2000.

[11] Bodo Maass. VBUnit. www.vbunit.org, 2001.

[12] Ken Schwaber. Scrum development process. www.controlchaos.com, 2001.

[13] Ian Sommerville. *Software Engineering.* Addison Wesley Longman, 2000.