Applying Test-First Programming and Iterative Development in Building an E-Business Application

Hubert Baumeister and Martin Wirsing

Abstract— Software development for applications where time-to-market is critical has to cope with, among others, imprecise requirements and reliability of the resulting software. This paper describes the positive experiences with the techniques iterative development with small increments and test-first programming in developing software for a framework for building customized Customer Relationship Management (CRM) applications.

Keywords— Software Development, Extreme Programming, Testing, CRM

I. INTRODUCTION

In addition, when starting the development of a software system 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 implementation fail to deliver in time. Software that takes several years to design and implement may find itself 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 [1], Crystal [2], FDD [3], DSDM [4], and others have been proposed [5]. A quite recent agile method which has gained a lot of popularity is Extreme Programming (XP) developed by Kent Beck, Ward Cunningham and others [6], [7]. 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 ecommerce 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.

XP consists of a set of practices, each of them a prac-

tice also common to other software processes, but taken to the extreme. For example, testing is well-known also with other processes; however, taken to the extreme, testing means in XP that tests are automatic and written even before the code that they test is implemented. Another XP practice is pair programming which puts code-review to the extreme. In XP each line of production code is produced by a pair of programmers sitting in front of the same computer; while one programmer is writing the code, the other immediately reviews that code. Note that each of these practices may have negative effects when done in isolation; however, taken altogether the positive effects of the practices cancel the negative effects of other practices. For example, the XP practice simple design requires that the software always represents the simplest design necessary to achieve its functionality. When functionality is added, it may happen that to keep the simplest design the software needs to be restructured. This, however, may break already existing functionality. To ensure that this does not happen, automatic tests, to test the existing functionality, are needed [6], [8].

Another characteristic of XP is that the software development process is guided by user-stories. A user-story represents a functionality that is useful to customers. The customer defines the user-stories and their priority. The developers then take one user-story at a time and implement that story without taking into account the user-stories that have not yet been implemented. Thus the customer gets immediate feedback, as she does not have to wait for all user-stories to be implemented to get an impression of the system and furthermore she can change user-stories not yet implemented or can remove them altogether and replace them with new user-stories.

We have used practices from Extreme Programming in the development of software for our CARUSO project [9], [10]. CARUSO is an EU-funded project [11] with the objective to design and implement a framework for building customized Customer Relationship Management (CRM) software. The major problem with designing such a framework is finding the right components and their functionality because the requirements on Customer Relationship Management software are quite complex as they involve all the business processes of a company, like marketing, sales, service, etc., and all its IT systems. A classical software development process would have required us to analyze a good deal of these processes before starting the design of the system [12]. Because of the complexities involved in CRM this proved to be impossible. Instead we started with a rough idea of the CARUSO architecture and defined user-stories based on the CRM needs of REMU, a utility

H. Baumeister and M. Wirsing are with the Institute of Computer Science, Ludwig-Maximilians-Unviersity (LMU), Munich, Germany. E-mail: {baumeist, wirsing}@informatik.uni-muenchen.de .

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

company and one of the partners of the project. During the implementation of these user-stories, the components of the framework and their functionality were discovered and implemented.

One of the components of this framework is the script engine which allows the execution of dialog-scripts. Dialogscripts guide the dialog between a call-center agent and a customer by presenting the agent with text and questions she should ask the customer.

In the next section we describe the CARUSO project and the architecture of the framework in more detail. Then we take a closer look at Extreme Programming. Section IV and V present experiences with Extreme Programming in the context of the CARUSO project. Finally Section VI provides a conclusion.

II. CARUSO

The CARUSO (Customer Relationship Support Office) project [10], [9] is funded by the European Union within the Information Society Technologies (IST) program of the 5th framework program [11]. 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 Relationship 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 some 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 allows 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, workflowmanagement 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 backoffice 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 build 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 utilitybill. 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 to 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 dialogscripts. 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.

As with the functionality of the CARUSO components itself, the problem with the design and the implementation of dialog-scripts and the script engine was to find the right amount of functionality so that the scripts and engine are useful, but, on the other hand, an implementation could be obtained fast so that the engine could be used to get feedback. In addition, robustness was a top requirement, in particular when run through a web-interface.

III. EXTREME PROGRAMMING

Extreme Programming (XP) is an agile or lightweight software development process developed in the past 5 years and which has gained a lot of popularity during that time [6], [7]. It consists of 12 practices that together allow to develop robust software which provides immediate business value to the customer. These 12 practices are:

- The Planning Game
- Small Releases
- Metaphor
- Simple Design
- Testing
- Refactoring
- Pair Programming
- Collective Ownership of Code
- Continuous Integration
- 40-Hour Week
- On-Site Customer
- Coding Standards

In our view, one of the most important practices is the Planning Game. The customer formulates her requirements as user-stories, where a user-story represents a piece of functionality which is useful to the customer. These stories are estimated by the developer and the customer decides, based on these estimates, which are the most important stories that should go into the next release. It is important that business people define the stories and their priorities, that is, make business decisions, while the developers estimate how long it will take them to implement that story. Thus the customer does not put functionality in a release which the developers don't believe they are able to deliver and, on the other side, no functionality is developed not needed by the customer.

In the next step the developers plan the iterations that make up a release by dividing the stories into tasks. After each task is implemented, the code will be immediately integrated into the system. Because of this Continuous Integration the customer has a running system available to test and get feedback from. One thing that could happen is that the customer, based on this experience, discovers that some of the originally planned user-stories have to be modified, are not needed anymore and have to be replaced by new user-stories, or their priorities have changed. In this case the customer may reschedule the user stories after consulting with the developers to make sure that the overall estimated resources within a release does not increase. The customer gets to decide which of the user stories are more important to him in case the estimated effort is higher than what the team can do in one release.

To gain this flexibility, the software development process has to be turned upside down. The classical development cycle, analyzing requirements, designing, implementing, and testing the system, even when done in iterations, usually does not permit this flexibility. This is because to analyze requirements, several user stories are analyzed at once. After producing a model of the requirements, for example, a use case model, this model is the starting point for designing the system, for example, using class-, interaction diagrams and state charts. Based on these models the software is implemented. Regarding flexibility, the problem with this approach is that each of these steps requires to look at several user-stories at once. This makes it hard to replace one user-story by another because each of the planned user-stories has already influenced the design.

Another problem with this approach is that for the first design of a system one usually tends to look for a complete set of requirements to start with, because any requirement that was 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. However, the danger here is that to start with the design and implementation of the system, all requirements have to be known, while, on the other hand, the real requirements are only known when a first version of the system is available to gain some experience.

This analysis paralysis showed up at several times during the CARUSO project; on the level of the framework itself, but also when designing the script engine and other parts of the system. For the framework itself, the problem with the requirements for a CRM framework is that CRM covers every aspect of a company, like their business processes and IT structure. For example, 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. Within the CARUSO project we solved this problem by focusing on one business process – customer service – first and building a pilot for that (cf. next section).

The way Extreme Programming addresses the problem of requirements paralysis 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.

To make such a software development process work, however, other practices have to be in place. Extreme Programming requires that the system implements the simplest design that makes the system run. In particular, one aspect of simple design is no code duplication. Any functionality in the system should be implemented only once. This makes changing and adapting the system to incorporate new functionality easier. To keep the system in the state implementing the simplest design, it is necessary that the system will be refactored when otherwise the design would suffer. Refactoring is only safe if it is assured that old functionality after refactoring still works [6], [8]. The XP practice of writing automated tests (functional tests, also called acceptance tests, and unit-tests) is a guarantee for that. Also, a common metaphor for the overall structure of the system is needed to define a common ground for the design of the system.

Another important practice, which allows XP to adapt to change, is that the customer should be on-site. This means that the customer is available for questions regarding the software all the time. The practices pair programming (all production code is written by two people sitting in front of one computer), collective ownership of code (all code belongs to all programmers, and every programmer is responsible for its quality), and coding standards refer to how teams of programmers work together. Within the CARUSO project these practices were not applied because software development took place at two different locations and each software development team had only one programmer.

IV. ITERATIVE DESIGN

Dividing the requirements into user-stories and then doing analysis, design, implementation, and testing for each user-story were used on two levels in CARUSO. On the first level it helped to define the scope of CARUSO and allowed us to detail the particular requirements REMU had. The problem was that CRM touches every business in a company, including marketing, sales, production, and services. REMU was referring to their a Customer Care Dream. However, because of the complexity of CRM it proved difficult for them to define precise requirements. After some discussion we found that for REMU the most important processes they need CRM software for were their customer service. This included service requests, like complaining and getting information about products, and changing customer data. One of the most important, but also most complicated service request 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. This allowed us to define 6 user-stories based on these support cases.

- 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: Give Information
- User-story 6: Move In / Out

Each user-story resulted in a running system and 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.

A. 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.

V. Test-First Programming

To achieve the robustness required of the script engine, automated tests were used and before any functionality was implemented, the test for that functionality was implemented first. In particular, if we noted that a method was needed during the implementation of some functionality, we first implemented a test for that method before implementing the method itself.

One advantage of first doing the tests was that this ensured that each functionality had its associated test and that everything that we want the software to do was document as a test. Another advantage was a better understanding of the functionality to be implemented, because the test made precise the functionality we expected.

It was also important that tests were automated, that is, that they could be run automatically without user interaction. This made it easy to always run all tests, which ensured that we did not unintentionally break any already existing functionality.

We wrote tests for:

- Intended Functionality
- Assumptions About the Code
- Border Cases
- Discovered Bugs

• Interconnection of COM/DCOM Components Written in Java and Visual Basic

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 should happen 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 the software or 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, bugs were documented to ensure 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. This proved quite easy; however, problems occurred with how values of type Variant were mapped to Java objects. Tests were important to document our assumption on how this mapping works.

In addition to achieving a robust implementation of the script engine, tests proved helpful, and where even a prerequisite, when new functionality needed to be added to the engine. Since all requirements and assumptions about how the script engine was working was encoded in tests, it was easy to add new functionality without breaking code that worked before even when adding the new functionality required changing the internal structure of the engine to keep the design simple.

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. To still keep a good design, the new functionality would have to be known at design time, which hinders flexibility as argued in Section III. Therefore, when adding new functionality, we changed the old design to produce the most simplest design that implements the old and new functionality. This approach, however, 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 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.

VI. CONCLUSION

In this paper we have presented our positive experience with some practices from Extreme Programming in the context of the CARUSO project. 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, however, were not needed for CARUSO. Although no big design phase preceded 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] Ken Schwaber, "Scrum development process," www.controlchaos.com, 2001.
- [2] Alistair Cockburn and Jim Highsmith, "Crystal methodologies," www.crystalmethodologies.org, 2001.
- [3] Peter Coad, Eric LeFebrve, and Jeff De Luca, "Feature-driven development," in *Java Modeling in Color with UML*. Prentice Hall, 1999.
- [4] DSDM Consortium, "Dynamic systems development method," www.dsdm.org, 2001.
- [5] Agile Alliance, "The agile alliance," www.agilealliance.org, 2001.
- [6] Kent Beck, Extreme Programming Explained, Addison Wesley Longman, 1999.
- [7] Ron E. Jeffries, Ann Anderson, and Chet Hendrickson, *Extreme Programming Installed*, Addison Wesley Longman, 2000.
- [8] Martin Fowler, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.
- [9] Hubert Baumeister and Piotr Kosiuczenko, "CARUSO: Customer Care and Relationship Support Office," in 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, Ricardo Gonçalves and Adolfo Steiger-Garção, Eds. 2000, pp. 115–120, A. A. Balkema, Rotterdam, Brookfield.

- [10] CARUSO Consortium, "CARUSO web-site," www.carusocrm.uni-muenchen.de, 2001.
- [11] European Union, "Future and emerging technologies initiative global computing," www.cordis.lu/ist/fetgc.htm, 2001.
- [12] Ian Sommerville, Software Engineering, Addison Wesley Longman, 2000.