Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2006 Proceedings

Americas Conference on Information Systems (AMCIS)

December 2006

XePtance: Supporting Distributed Acceptance Tests in Extreme Programming Projects

Roger Müller
ERCIS - European Research Center for Information Systems

Björn Eilers
ERCIS - European Research Center for Information Systems

Christian Janiesch
ERCIS - European Research Center for Information Systems

Herbert Kuchen
ERCIS - European Research Center for Information Systems

Jörg Becker
ERCIS - European Research Center for Information Systems

XePtance: Supporting Distributed Acceptance Tests in Extreme Programming Projects

Roger A. Müller
ERCIS – European Research Center for Information Systems
roger.mueller@ercis.de

Björn Eilers
ERCIS – European Research Center for Information Systems
bjoern.eilers@ercis.de

Christian Janiesch
ERCIS – European Research Center for Information Systems
christian.janiesch@ercis.de

Herbert Kuchen
ERCIS – European Research Center for Information Systems
kuchen@ercis.de

Jörg Becker
ERCIS – European Research Center for Information Systems
becker@ercis.de

ABSTRACT

Acceptance tests are of major importance for the successful roll-out of new software products. Any brilliantly engineered software unable to suit the user's expectations will fail in practice. Tight time-to-market and limited resources makes acceptance testing difficult. This is especially true if multiple locations for development and testing are concerned. We argue that a solution must allow an off-site customer to test the software in question whenever he finds the time to do so. This requires tool support as well as simplicity in design. Within these constraints, we propose on a framework and elaborate on a prototypical software solution that facilitates distributed acceptance testing in Extreme Programming projects. We utilize technologies to allow the deployment over the internet even in strictly organized company networks. We provide further evidence of the feasibility by deploying the software prototype to evaluate the design.

Keywords

Extreme Programming, acceptance tests, agile IS development.

OPEN ISSUES IN AGILE IS DEVELOPMENT

An insufficient involvement of the prospective users into the requirements engineering process as well as the (acceptance) testing of prototypes are the main reasons for delays and in the worst case the complete failure of software projects. Incomplete or inaccurately formulated specifications and software that does not meet the customer's requirements are the result (Johnson, Boucher, Connors and Robinson 2001, The Standish Group International 1995).

In the development process of information systems several persons are involved who differ in knowledge, abilities, and status. Especially end users and developers often do not speak the same language. It has always been a principal task of the IS discipline to overcome this defect (Bostrom and Kaiser 1981, Coughlan and Macredie 2002). Attempts to create artifacts that aim at solving the communication gap in the requirements specification phase mainly origin from the area of information modeling. Promising approaches include configurative multi-perspective modeling (Becker, Delfmann, Dreiling, Knackstedt and Kuropka 2004), domain-specific modeling (Luoma, Kelly and Tolvanen 2004), and on a more general level meta-modeling to facilitates the process of model adaptation (Marttiin, Lyytinen, Rossi, Tahvanainen and Tolvanen 1995).

Several approaches to integrate the customer into the software design phase and testing have been proposed (for an overview cf. Balzert (Balzert 2000) and Liggesmeyer (Liggesmeyer 2002)). Acceptance testing is generally used as a final test by customers to check if the software complies with their needs. However, at the time this final test is conducted it is usually too late to find an easy fix for rather basic problems. Thus, a failed acceptance test usually results in overspending in IT projects.

Extreme Programming (XP) (Beck 2000, Beck and Andres 2004, Beck and Fowler 2000), an agile method for small to medium sized software development projects, faces this problem by not only incorporating regular, formalized acceptance tests into its process, but also integrating an ongoing acceptance test by allowing and encouraging the customer to test the software whenever he sees fit. This rigorous and time-consuming process is intended to minimize the cost of (re-)developing pieces of software that proved useless to the customer.

One of the main advantages of XP is the flexibility in the specification of the desired software product and the strong bond with the customer that goes as far as to put a customer on the development team. However, this strong bond is undermined by the business requirements of the customer, whose other obligations often make a permanent deployment with the development team difficult or next to impossible.

As this difficulty is a well known problem for XP, it seems to be the next logical step to develop software that eases the dialectics between the customer's obligations and the development team's needs.

Thus, our research question is as follows: *How can acceptance testing in XP projects be software supported in a distributed environment?*

In this paper we present a framework and a prototypical software implementation called XePtance which allows distributed acceptance testing in heterogeneous environments. The software can be used to install concurrent versions of new software remotely and transparently to the customer and allows him to give efficient feedback by means of screenshots enriched by comments to further describe the errors encountered and a relevant part of the program state at the time of the screenshot.

A qualitative research method was chosen to guide the research process: design research. As Hevner et al. state: "Design science […] creates and evaluates IT artifacts intended to solve identified organizational problems" (Hevner, March, Park and Ram 2004). Thus, this research method is applicable to our research question. Furthermore, action research has certain ties to our research endeavor as one of the important aspects of action research is that the researcher usually faces a dichotomy: problem solving interest and research interest (Avison, Lau, Myers and Nielsen 1999).

The structure of this paper is the following: In section 2, we elaborate on the XP process and explore alternative approaches to acceptance testing. In section 3 a more advanced acceptance testing framework is proposed. We fortify our approach with a case study utilizing a prototypical software implementation in section 4. The paper concludes summarizing the findings and pointing out further work.

AGILE SOFTWARE DEVELOPMENT AND RELATED WORK

Extreme Programming Software Development Process

Numerous approaches to agile development of information systems exist (cf. e. g. Baumeister 2004, Mugridge and Cunningham 2005). XP is an agile, light-weight software engineering method originally introduced by Beck. It features an iterative, incremental approach which is best described by a set of rules proposed by Beck (Beck 2000, Beck and Andres 2004, Beck and Fowler 2000) and explained in more detail by Jeffries (Jeffries, Anderson and Hendrickson 2001).

First of all, XP relies on a short release cycle which emphasizes the iterative and incremental aspect common to all agile methods. A single release cycle is called *iteration* in XP terminology and lasts for a fixed time span, usually six to eight weeks. The proceedings for each single iteration are detailed at its beginning in a *planning game*. The game tries to combine the customer's insight to his desired product and the experience of the programmers to determine the difficulty and the time required for the implementation of the desired product. The customer expresses his requirements in a metaphor and in (or literally on) *story card*s which detail just one aspect of these requirements. The developer uses his experience to split the story cards into *task cards* which in turn describe one aspect of the story card of the customer. Every aspect can be implemented on its own and can be timed efficiently. Task cards which do not fit in a single iteration time slot are discarded based on priorities given to the original story cards by the customers.

Before releasing the implementations of the task cards, an acceptance test is performed by a customer who should be assigned to the developers' team on a permanent basis. The acceptance test should partially be formulated by a customer in the beginning of the iteration and be implemented in the form of automated test cases which the implementation can be tested against. Additionally an *on-site customer* should continuously check if the implementation is heading in the direction he desires by testing intermediate releases of the software whenever available.

The implementation in XP is done in so called *pair-programming*, which essentially means that two programmers share one computer – one doing the programming, the other one supervising. By constantly switching programming partners and roles, everyone in the team should gather knowledge of the whole code. In turn this enables a *common code base*, where all

programmers have equal rights and responsibilities over the whole code. To keep concentration up on the programming team, a steady speed (which was originally labeled *40h week*) should be sustained throughout the program.

If some part of the code turns out to be inefficient or inappropriate, it should be *refactor*ed (Fowler, Beck, Brant, Opdyke and Roberts 1999). To conserve the semantics of the code refactored, all parts of the program have extensive automated regression tests which should in part be programmed before actually coding to provide a kind of programmer's specification to the produced code. Though new code should be *integrated continuously* into the common code base, it has to pass the whole regression test suite before being submitted. As stated above, the tests for the regression test suite are written before the code. This is because the test is used as an informal specification for what the code should be able to do (Crispin and House 2002).

Approaches to Distributed Acceptance Testing

Currently, there are a number of applications available which provide some functionality to support XP projects that do not have an on-site customer (for an elaborate overview of framework and testing tools cf. Pancur, Ciglaric, Trampus and Vidmar 2003). However, as these have not been developed specifically for XP projects, they cannot be integrated into the XP software development process without drawbacks. OPUS and Bugzilla represent two current efforts to facilitate distributed acceptance testing (Gunzer 2004, The Bugzilla Team 2005).

OPUS: The Online Program Update Service (OPUS) by Ivotec (cf. www.ivotec.de) is an application to support and simplify the handling of software updates. The application works with a client/server architecture with two different clients (*OPUS Developer* and *OPUS Customer*) and a centralized server that is owned and run by Ivotec – the server usage is paid for through a monthly fee. The developer client is used by the development team to upload software updates like patches and new licenses to the OPUS server and to define the access rights for each customer. The customer client is used by the software users to automatically search for new software updates and to automatically install them on the customer's system. Unfortunately, OPUS has some drawbacks: OPUS is a proprietary system that only works with Microsoft Windows. However, there is a number of industry sectors that traditionally depend on other platforms and operating systems (e. g. Apple computers in the Desktop Publishing business). Furthermore, it does not allow the user to give feedback about the quality of the software updated by the OPUS system.

Bugzilla: Bugzilla is a software tool originally developed to support user feedback and to track bugs during the development of the Mozilla browser (cf. www.bugzilla.org). Through a web interface, it gathers user feedback on its server component and offers several kinds of reports. It provides for a review-system before submitting a bug for development and automatically scans for incongruencies in the bug database. However, as it has no option for software updates, it covers only one side of the story.

TOWARDS A FRAMEWORK FOR DISTRIBUTED ACCEPTANCE TESTING

Customer Integration in XP Projects

Almost half a decade of IS development has not produced a method or procedure to keep projects within budgets, development schedules, and to guarantee a minimum of product quality. The fast-paced and ever-changing business environment today makes it even harder to grasp the requirements put forth by the users. Specifications change in the course of the project and adaptations have to be made.

Static project management did not prove to cope with small- and medium-sized projects and more agile procedures with on-site customer integration have been proposed. Tight IT project budgets do not alleviate the situation, especially if the customer is off-site and has to be brought in for testing.

Especially, the on-site-customer practice is one of the most valued practices of XP. The opportunity of having a customer at hand for rich and easy feedback is appealing to most of the developers. If properly integrated into the overall software development contract, the feedback provided can aid the developers to produce software that is of more valuable to customers. Furthermore, the developers can share some responsibility for the quality of the software by tightly integrating the customer and allowing him to make the decisions at critical points in the software development process. Usually it is assumed that such behavior can lead to decreased costs with respect to changes late in the development.

However, documented cases of XP projects show that it is often impossible for a company to grant a development team full-time, on-site access to an employee who acts as a representative for the customer. This is mainly due to financial considerations, as the working hours of the employee add up to the project costs. Especially for small and medium enterprise

an employee competent enough to give valuable feedback to the XP team might not be dispensable due to other commitments in the company (Rumpe and Schröder 2001).

XP is not only demanding much time and patience from the customers but is also putting a high demand on the developers. They are supposed to continuously release small bits of software, so the customer can test it. It is rather obvious that releasing early and often is only worth the extra time and effort if the customer actually finds the time to check the releases for their fit to his needs. Thus, this practice is tightly coupled with the on-site customer.

If the customer tests the continuous releases, the final acceptance test should be easy to pass for each iteration on the condition that the requirements were stable throughout the iteration and that the requirements were adequately formulated in the first place. As acceptance tests are not fully automated in most software engineering situations, the same has become true for XP – acceptance tests are no longer required to be strictly automated, and part of the acceptance testing (at least in the more general meaning) is done on the fly.

Seeing the relevance of the tight coupling of customer and developer, the implementation of a feasible solution is long overdue. As we will expound and discuss means to resolve a conflict between a proposed theory (XP) and its application and as we are willing to put our research project to practice and to iteratively refine it in reflection of learned lessons, we argue that our approach is design science in the sense of Hevner et al. (2004).

Requirements for Customer Integration in XP Projects

The foremost task is to discover the means to keep the tight coupling of customers and developers even when the customer is not on site. Based on the general assumptions given above, we will analyze and detail the requirements for a software tool that can bring the customer and the development team tighter together with respect to releases and the prompt testing of these releases without stretching the time-table of the customer overly. The prompt testing could refer to manual acceptance tests or continuous tests on a current release of the software.

Basically, there are four ways how a customer can limit the time he spends with the developers to an economic value.

- The customer does not have a full-time but a part-time assignment to the developer team. The obvious advantage (spending less time with the development team) is overshadowed by the (equally obvious) disadvantage that the customer still has to be physically present while consulting the development team. This means either getting to the team on a fixed schedule (which does neither reflect the business urges of the customer nor the flexibility needed from the development team) or that he has to pop in spontaneously, which will – earlier rather than later – result in the customer not showing up at all due to other duties.
- There is no on-site customer. The development-customer relationship is replaced by a document-driven information exchange similar to the document output produced by classic and rather static software development models, e. g. the waterfall model. The disadvantage of the approach, the inflexibility and slow response times, is in certain situations compensated by the added assurance of the customer stating his wishes in a clear and reproducible manner, i. e. by written documentation of his suggestions. But generally, the drawbacks of the communication gap in requirements engineering mentioned in the first section apply.
- There is no on-site customer. The customer offers some remote access to domain experts in the issues concerning the project by phone. This allows for some information on implementation decisions, but essentially removes the release tests from the picture.
- There is no on-site customer. Instead, the customer receives releases by the development team and checks them whenever he finds time to do so. The test process is tool-supported, easy to use and produces little overhead for the customer and can be integrated into his daily routine.

We argue that the last way might produce the best results and could easily be seconded by phone calls (to get prompt personal feedback where needed) or documents (for extra assurance in contract questions). It allows asynchronous, document supported, efficient (to both customer and developers), distributed, and tool-driven acceptance tests.

Design Issues for Software Supporting Distributed Acceptance Testing

In the following section we propose an architecture for a software tool that complies with the requirements given in the previous section. Apart from the general requirements two major design issues have to be taken into consideration: the simplicity of design and the independence from technological constraints.

Simplicity

The target audience for the software tool is the domain expert. Thus, we do not assume a familiarity with any development related issues and try to keep the interface as simple as possible, so the domain expert can familiarize himself with the tool on short notice. The simplicity should also lead to a more efficient testing process so that the expert can use even short intermissions in his regular work cycle for efficient testing. This also implies that the testing client gathers as much information about the current program state as possible with the complexity hidden from the user. For more detailed information about the information gathering process please refer to section 4.

Another important simplicity issue is versioning. Traditional versioning systems require the user to manually resolve versioning conflicts. Furthermore, the installation process has to be initiated manually by the end user. Both are not an option. Instead we propose a "push button" technology, which automatically resolves versioning issues and starts the installation and initialization process right after the current version is downloaded to the computer. The installer should be capable of initializing secondary systems, e. g. databases.

The simplicity restriction applies only to the end user interface but not to the developer interface. The developers have some GUI support (e. g. the server handling), but essentially their input to the system is script-based which adds flexibility in trade-off for comfort. For more detailed information on the developer's interface please refer to the case study in section 4.

Target Platforms and Middleware

For the ease of use for the development party, the whole system has to be platform independent. The platform independency can be achieved either by implementing multiple clients or by implementing a client which runs on a virtual machine which is available for a multitude of platforms. Considering this, either Java or the .NET platform comes to mind. As the field-tested Java is handily available for a multitude of platforms, it was chosen as the platform to implement the clients and server.

Related to the issue of the target platform is the choice of the target middleware. While the transport layer of the middleware should be flexible, secure, and interchangeable to provide for the client's needs, the middleware itself should be field-tested and support as many platforms as possible. All these aspects are fulfilled by CORBA (The Object Management Group 2000), which has an interchangeable (and thus flexible) transport layer which can be secured by simple SSH tunneling (cf. www.openssh.org for in-depth information) and which has not only open source implementations but also commercial implementations from renowned companies such as IONA (cf. www.iona.com).

The middleware will be used not just by the server and the clients, but can also be used by the program that is tested. It can use an IDL CORBA interface to efficiently and transparently transfer program execution related data back to the server for review by the development team in an asynchronous way.

Architecture of the Software

From the architectural point of view the framework can be split into three clients (including a feedback API) and one server, as depicted in Figure 1.

The server offers three services: First, it provides a layer of security through the insecure internet, by the means of offering a *SSH Server* to which the clients can connect. Second, it offers a connection to the clients that these can use for data exchange with the server. Third, the program that is tested can connect to the server and send part of its internal state. The API used is simply called *Feedback API* in XePtance terminology.

The *Customer's Client* receives updates from the server over the internet and if applicable, makes screenshots and collects remarks from the tester. The *Programmer's Client* is used to save, manage, retrieve the feedback, which is provided in PDF format, and enter new versions of the software into the system. The *Administration Client* is used to create and modify projects and user management. Both development clients operate in an intranet environment. Programs implementing the Feedback API can send data about the current state of the program back to the server to be included in the report.

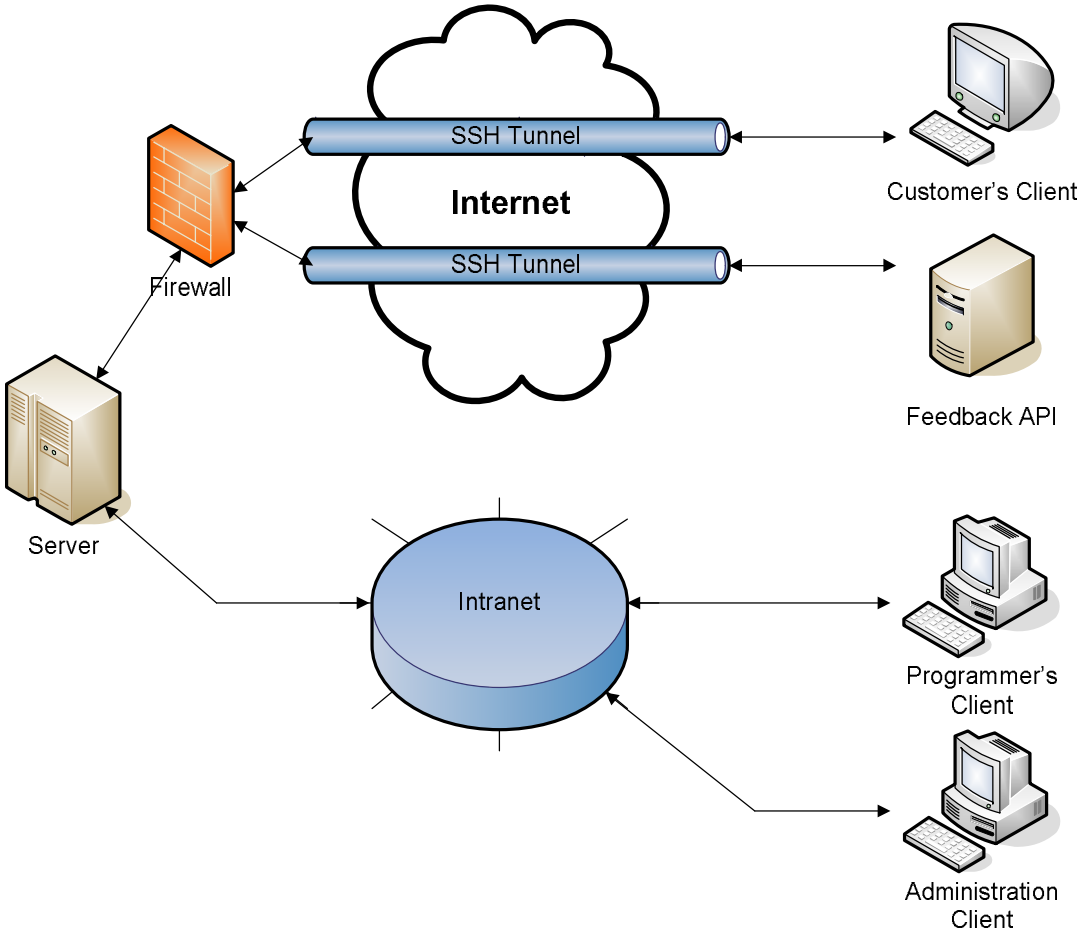

Figure 1. Architecture of a software for distributed acceptance testing

DESIGN ISSUES AND CASE STUDY

XePtance is the prototype of a distributed client/server system to support off-site customers in XP projects. The server stores different iterations of the software in development as well as related test results and further feedback. The following case study illustrates the core features of XePtance in detail:

> *A small enterprise needs a new invoicing application for their order-to-cash process and has concluded a contract with a near-shore development centre. The software should allow user-defined invoice templates, contain preview functionality and support multilingual invoices and different media, like e-mail invoices (with attached PDF files), printed invoices, and EDI transfer.*
>
> *Because of economic reasons and a limited software life-cycle, XP will be used for development. As there are no free capacities to send an employee to the development centre, it was decided to use the XePtance prototype for testing and customer reviews. Testing will be done by two employees: one being responsible for overall acceptance testing, the other one being a future end user who will partake in testing to comment on the usability. Both employees will do the testing alongside their normal working hours.*

For the project infrastructure setup, the Administration Client is used. The administrator can maintain users (on both developer and customer side) and create or remove the projects themselves (cf. Figure 2 for a screenshot of the main administration dialog).

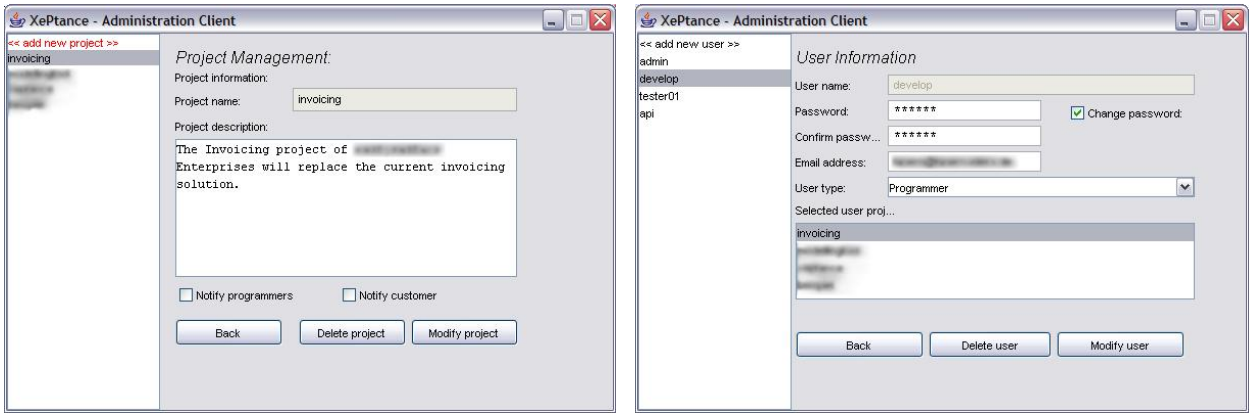

Figure 2. Administration Client

The development center has finished the first iteration of the invoicing software. Previewing works; printing however is still disabled. This version is published on the XePtance server.

This process is conducted in several steps via the Programmer's Client:

- The files that make up the project are selected in a tree view.
- The installation process is described on task level (e. g. create directory, execute database query etc.). These tasks are compiled into an installation script later on.
- Additionally, it is possible to create an uninstall script that is executed whenever an iteration is to be removed. This needs to be done if an iteration is faulty or if the prototype will be replaced by the fully functional version.
- Finally, the install and uninstall scripts are compiled into an XML script (that is compatible with the open source make tool Ant), the project files are compressed and transferred to the XePtance server (cf ant.apache.org for more information on Ant).

Figure 3 depicts the dialog of the Programmer's Client.

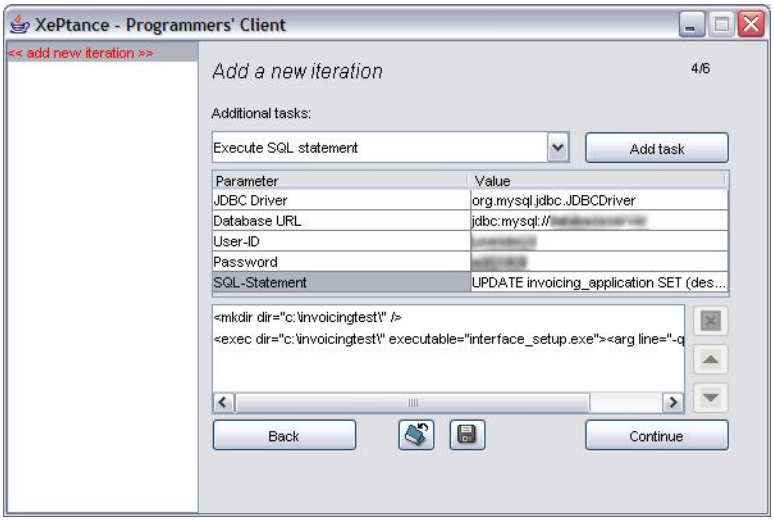

Figure 3. Programmer's Client

Afterwards, the new iteration is available for the customer to download. Depending on the setup the Customer's Client automatically checks if a new iteration is available on the XePtance server and will ask the customer whether he wants to install it or not (cf. Figure 4). If this option is disabled the customer has to check for updates manually.

Figure 4. Customer's Client – installation of a new iteration

In case the customer accepts a new installation, the client will download the new iteration from the server and install it on the customer's computer. After this, the test of the new iteration can begin – supported by the Customer's Client.

> *The acceptance tester begins with his check of the software by clicking the "Test iteration" button. However, nothing happens. After investigating further, he finds out that the software crashes during start-up. He takes a copy of the command-line error message and sends it back to the developers via the feedback tool which is part of the Customer's Client (cf. Figure 5).*

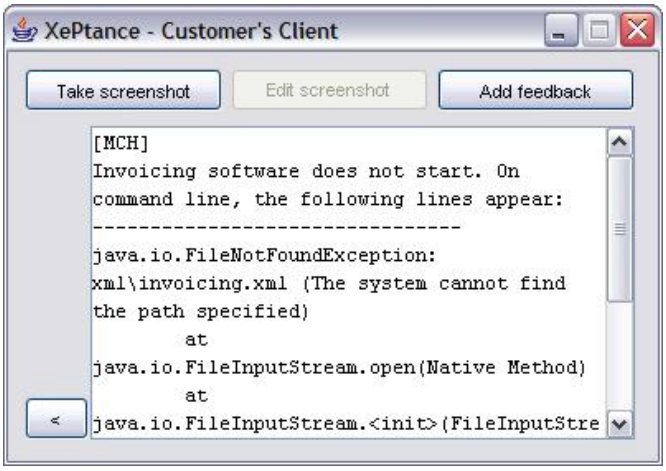

Figure 5. Customer's Client – gathering feedback

Any feedback sent by the client is gathered on the server. It can be accessed by the developers through the Programmer's Client which displays all feedback available for an iteration and can download a feedback report. This report can consist of all feedback or only of all new feedback messages that have been transferred onto the server since the generation of the last report (cf. Figure 6 for the feedback management dialog).

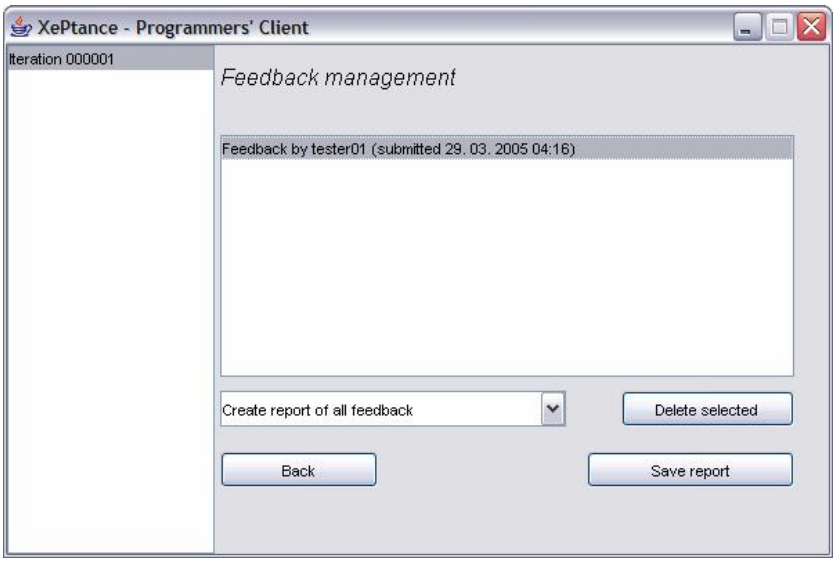

Figure 6. Programmer's Client - creating a feedback report

The selected feedback data is then compiled into a PDF document which can be viewed with any PDF viewer. An example of such a report is depicted in Figure 7. The exemplary report shows the feedback the customer sent to development when he could not run the software.

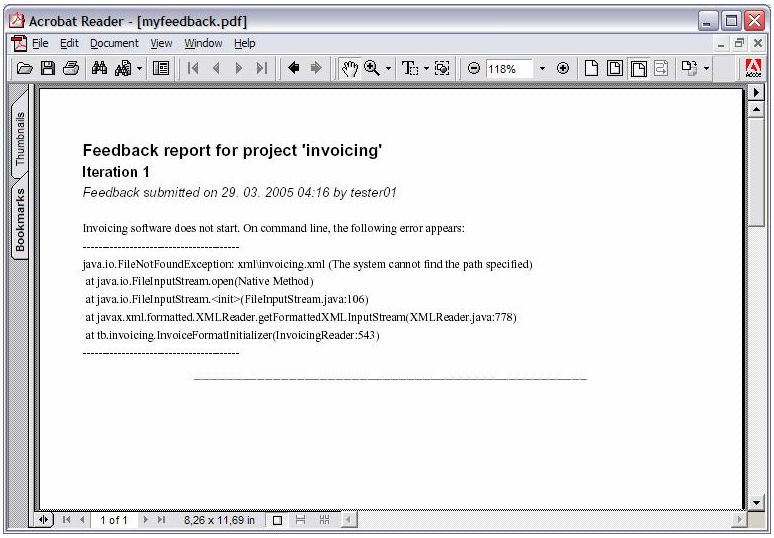

Figure 7. Exemplary feedback report

> *Development has fixed the bug and uploaded a new version to the server. The customer successfully installs the new version on his computer and runs it. However, upon calling on the Preview function, he detects some bugs: The preview screen is unreadable and the button to leave the screen is always disabled. To clarify what he means, he uses the built-in screenshot utility and marks the errors he has found.*

In order to allow a thorough explanation of a problem, the Customer's Client provides for taking screenshots of the user's desktop and editing them afterwards. A screenshot is uploaded to the server in conjunction with text-based feedback. Figure 8 shows an example of this screenshot functionality.

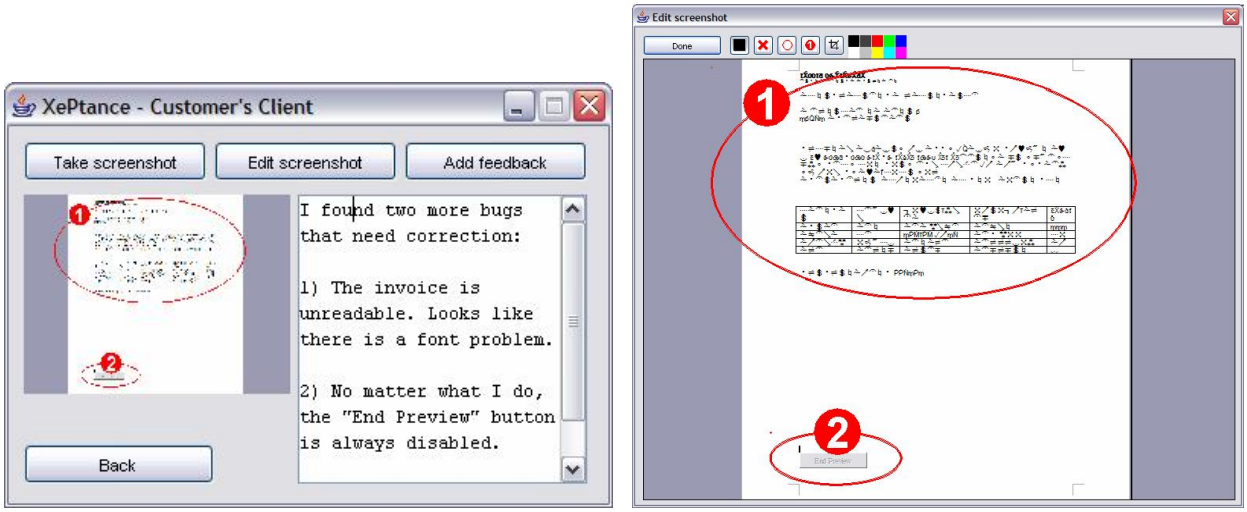

Figure 8. Screenshot functionality

The screenshot is inserted in the report on top of the feedback itself (cf. Figure 9).

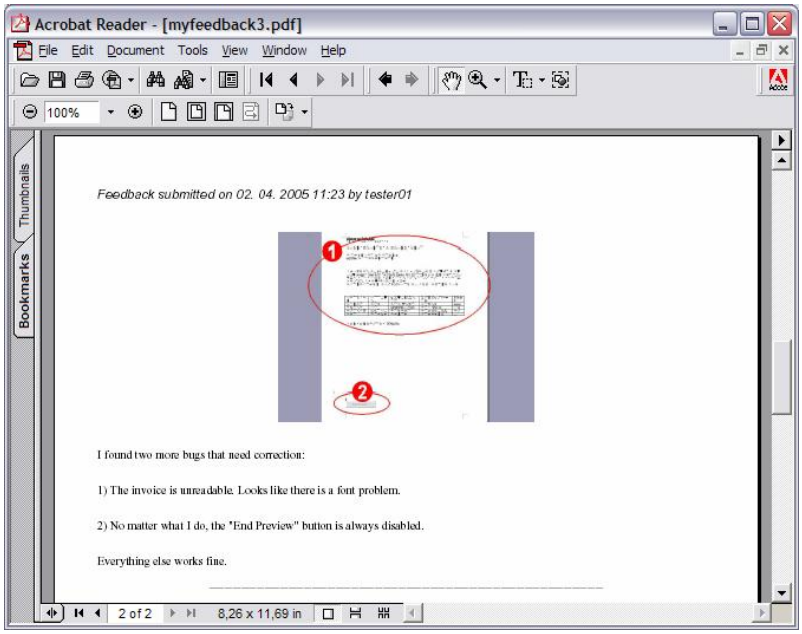

Figure 9. Report with screenshots

> *After the problems with the preview functionality have been fixed, printing functionality has been added to the tool. As the software runs stable in the first iteration of its publication, the invoicing software is accepted by the customer and given to the end user. There are some design issues, but he is in general quite content with the result – with one exception: From time to time the invoices are not printable, a repeatedly occurring error that at first cannot be resolved by the development team. They use the Feedback API to directly gather data from the invoicing software.*

The Feedback API allows a code-based gathering of data from the tool. It allows sending the content of variables to the server as well as the sending of short messages to indicate, e. g., that a specific method has been called. As the client/server communication is based on the CORBA middleware, the Feedback API is supported by all programming languages that

support the use of CORBA (this includes, but is not restricted to, C, C++, and Java). The content of the variables is shown in the report as follows:

```
class Invoicing; method printInvoice; int pageWidth = -143999
```

> *The reason some invoices could not be printed was that the dimension of one variable was wrong. The developers fix this error and publish another iteration of the software. It is installed on the computers of the client and tested for the last time. Because no other issues occur, the software then is rolled out in the enterprise.*

In addition to the above-mentioned features, XePtance offers the following features: The software and feedback on the server may contain sensitive internal data of the company. Thus, the user can use a tunneled and encrypted connection through the internet to the XePtance server. Moreover, the Programmer's and the Customer's Clients will not be running on the developer's and customer's computers all the time. Therefore, the server has the ability to send notifications via e-mails to both parties if new feedback respectively iterations are available on the server.

SUMMARY AND FUTURE WORK

In a distributed and heterogeneous environment, ignoring the input of off-site users is not an option. The differing demands cannot be incorporated into any software without the integration of the end users in acceptance tests. Our framework allows the customers to securely remote access the software with a flexible repository that allows both, to update of the software and to give feedback via a simple yet powerful engine. This improved the design of the case study software significantly.

If requirements are uniform within the different locations where the software is to be deployed, a distributed testing tool might not be necessary at first. However, we designed software that encourages a strong bond between developers on the one hand and the end users and domain experts on the other side. The ease of use for the end users and the flexibility in testing for the developers should encourage the deployment of this software, as it will ensure a software product that is up to the requirements of the software customers. The deployment of the software is estimated not to increase IT project costs but to keep them stable in opposite to increased travel expenses and delays due to the customer's business schedule.

Though XePtance has proven itself as a valuable testing asset, it is still a prototype and several extensions seem appropriate:

The software should be enabled to support the complete test workflow from error recognition to their resolution (e. g. bug tracking, reminder, and scheduling). As many software projects differ in content and organization, it should be possible to allow the users to add additional, customizable information. For bigger projects, support for project subgroups is also to be considered. Furthermore, the Programmer's Client could be extended to support the developers with the creation of new program versions (e. g. by generating only incremental uploads, an automatic pre-selection of files that have been used in the last iteration etc.). Another possibility to support the developers' work would be to integrate the Programmer's Client into one of the leading IDEs (such as Eclipse, NetBeans or Borland JBuilder), thus coupling of creation of new iterations even tighter with the development process (Pancur et al. 2003). The Customer's Client could be updated to also upload test suites corresponding to the iteration they have uploaded to see if the automatic test results meet their approval. There are many systems whose program logic is not installed on the user's computers, but on a central server (e. g. any ERP software like SAP R/3). Therefore, the software should also support a user notification if a remote component has been updated instead of downloading each iteration to the system. For programming languages that support an automatic evaluation of their environment (e. g. Java with the Reflection API), the Feedback API could be extended to gather all relevant data automatically. As many applications have output formats other than the display screen (e. g. Excel spreadsheets, PDF documents, EDI message dumps), another possibility would be to enable the software to upload these documents, too.

REFERENCES

1. Avison, D., Lau, F., Myers, M. and Nielsen, P. A. (1999) Action Research, *Communications of the ACM*, 42, 1, 94-97.
2. Balzert, H. (2000) Lehrbuch der Software-Technik, Spektrum Akademischer Verlag.
3. Baumeister, J. (2004) Agile Development of Diagnostic Knowledge Systems, Dissertation, University of Würzburg.
4. Beck, K. (2000) Extreme Programming. Embrace Change, Addison-Wesley.
5. Beck, K. and Andres, D. (2004) Extreme Programming Explained, 2nd Edition Edition, Addison-Wesley Professional.
6. Beck, K. and Fowler, M. (2000) Planning Extreme Programming, Addison-Wesley.

7. Becker, J., Delfmann, P., Dreiling, A., Knackstedt, R. and Kuropka, D. (2004) Configurative Process Modeling - Outlining an Approach to Increased Business Process Model Usability *Proceedings of the 15th Information Resources Management Association Conference (IRMA)*, New Orleans, 615-619.
8. Bostrom, R. P. and Kaiser, K. M. (1981) Personality differences within systems project teams: Implications for designing solving centers *Proceedings of the 18th Computer Personnel Research Conference*, Washington, D.C., 248-285.
9. Coughlan, J. and Macredie, R. D. (2002) Effective Communication in Requirements Elicitation: A Comparison of Methodologies, *Requirements Engineering*, 7, 2, 47-60.
10. Crispin, L. and House, T. (2002) Testing Extreme Programming, Addison-Wesley Professional.
11. Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. (1999) Improving the Design of Existing Code, Addison-Wesley Professional.
12. Gunzer, F. (2004) Mr. Holland's Opus: OPUS - Online Program Update Service von Ivotec, *Entwickler*, 01, 15.
13. Hevner, A. R., March, S. T., Park, J. and Ram, S. (2004) Design Science in Information Systems Research, *MIS Quarterly*, 28, 1, 75-105.
14. Jeffries, R., Anderson, A. and Hendrickson, C. (2001) Extreme Programming Installed, Addison-Wesley.
15. Johnson, J., Boucher, K. D., Connors, K. and Robinson, J. (2001) Project Management: The Criteria for Success, *Software Magazine*, 21, 1, 3-8.
16. Liggesmeyer, P. (2002) Software-Qualität, Spektrum Akademischer Verlag.
17. Luoma, J., Kelly, S. and Tolvanen, J.-P. (2004) Defining Domain-Specific Modeling Languages: Collected Experiences, in Juha-Pekka Tolvanen, J. Sprinkle and M. Rossi (Eds.) *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM 2004)*, Vancouver.
18. Marttiin, P., Lyytinen, K., Rossi, M., Tahvanainen, V.-P. and Tolvanen, J.-P. (1995) Modeling Requirements for Future: Issues and Implementation Considerations, *Information Resources Management Journal*, 8, 1, 15-25.
19. Mugridge, R. and Cunningham, W. (2005) FIT for Developing Software Framework for Integrated Tests, Prentice Hall.
20. Pancur, M., Ciglaric, M., Trampus, M. and Vidmar, T. (2003) Comparison of Frameworks and Tools for Test-driven Development, in M. H. Hamza (Ed.) *Proceedings of the 21st IASTED International Multi-Conference on Applied Informatics (AI 2003)*, Innsbruck, 980-985.
21. Rumpe, B. and Schröder, A. (2001) Quantitative Untersuchung des Extreme Programming Prozesses. Technical Report ViSEK/006/D of the TU Munich, Munich.
22. The Bugzilla Team (2005) The Bugzilla Guide - 2.18 Release, http://www.bugzilla.org/docs/2.18/pdf/Bugzilla-Guide.pdf.
23. The Object Management Group (2000) The Common Object Request Broker: Architecture and Specification, http://www.omg.org/cgi-bin/apps/do doc?formal/00-10-33.pdf.
24. The Standish Group International (1995) The Chaos Report, http://www.standishgroup.com/sample research/PDFpages/chaos1994.pdf.