The XP of TAO
eXtreme Programming of Large, Open-source Frameworks

Michael Kircher
Siemens AG
Corporate Technology, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany
+49 89 636 33789
Michael.Kircher@mchp.siemens.de

David L. Levine
Dept. of Computer Science.
Washington University
1 Brookings Drive
St. Louis, MO 63130 USA
+1 314 935 7538
levine@cs.wustl.edu

ABSTRACT

The Adaptive Communication Environment (ACE) and The ACE ORB (TAO) are cutting edge, real-time software products that by nature support and grow from change. Therefore, eXtreme Programming (XP) would seem to be appropriate for these projects. In fact, their developers have been following many XP practices all along. We describe these practices here, and identify some that we do not follow rigorously.

However, three characteristics of ACE and TAO contraindicate the application of XP. These products are 1) large, 2) open-source, and 3) development frameworks, that is, not end-user applications. We explore the impediments to XP for such projects and how we have, or would like to, overcome them. In particular, we introduce *remote pair programming* as a potential augmentation to traditional pair programming.

Keywords
eXtreme Programming; Patterns and Frameworks; Distributed and Real-Time Middleware

1 INTRODUCTION

Many successful large-scale software projects rely on the open-source model [1]. The long lifetime and widespread usage of open-source products leads to incremental feature introduction, rapid development cycles, and, above all, change. The most notable feature of open-source is a potentially large and fluid development team.

eXtreme Programming (XP) is a natural development approach for open-source in many respects, because it encourages change and supports rapid evolution. However, the large, distributed, and variously committed open-source development team does not effectively support pair programming. In practice, we have found that XP can be successfully applied to a long-term open-source development project.

This paper contributes threefold. First, it documents the development process used in the ACE [2] and TAO [3] projects. In particular, it considers the interaction of XP and open-source. Second, we attempt to motivate other large, open-source, and/or framework development projects to consider XP. We include some suggestions for development process components. Finally, it discusses the current deficiency we see with XP applied to distributed development: that *remote pair programming* is necessary for large and/or open-source development projects. Section 2 documents our development process, and compares and contrasts it with XP. Section 3 discusses how XP can be successfully applied, in particular, to large and/or open-source projects. Section 4 introduces remote pair programming, and Section 5 concludes with what we have found to be the keys to successful application of XP.

2 DOC AND XP

The Center for Distributed Object Computing (DOC, or DOC group) at Washington University has long practiced XP. We have employed many XP practices before they were identified as such. We discuss how in this section. Our purpose here is not to justify whether or not the DOC group practices XP. Rather, it is to show that our project is different in significant ways from the textbook XP organization. In particular, our development team is large and distributed, because our products are open-source. Furthermore, our products are frameworks, and therefore not always simple and minimal. Finally, one of our leading products is standard-based, and therefore constrained in its interface.

How is DOC eXtreme?

The following is a checklist [4] stated by the "The Three Extremos" in the Portland Pattern Repository [5]. We compare our research group against it briefly to demonstrate our adaptation of XP.

- *Paradigm: Your project is extreme to the degree you see change as the norm, not the exception, and optimize for change.* Open-source framework development is in constant change, the DOC group would not have succeeded with poor support for change. There are many examples in later sections describing this, e.g. how changes in the standard API specification trigger change in our products.

- *Values: Your project is extreme to the degree that you honor the four values - communication, simplicity, feedback, and courage - in your actions.* We describe our conformance to this in the discussion of the four values of XP below.
- *Power sharing: Your project is extreme to the degree that Business makes business decisions and Development makes technical decision.* Business decisions are made by the director of the group and his research staff; development decisions are made by the domain experts of the core development team. On open-source projects, both the constituency of these domain experts and the development team itself can change often. That is one of the strength of open-source, by bringing many viewpoints into the development. It is also one of the challenges, because there need to be some constraints on the development process. We address that challenge by retaining source code control in the core development team and a few other select individuals.
- *Distributed responsibility and authority: Your project is extreme to the degree that people get to make the commitments for which they will be held accountable.* Masters and PhD thesis work supports this idea. The students get to make their commitments and are responsible for achieving them. Because their thesis work is very often tightly integrated with other work in the team, everybody in the team has an interest in their success. This ensures that everybody is motivated to support them if needed; they do not have to struggle if problems occur. People external to the core team helping with work on the open-source project are responsible for their work, yet they cannot be held accountable in the traditional way. The system works slightly differently; the motivation to build proper software or to support the product when bugs occur stems from the public contributors list. Nobody wants his/her name associated with something that does not work or is of obvious bad quality. Considering all this, it can be stated that our way conforms to the eXtreme way.
- *Optimizing process: Your project is extreme to the degree that you are aware of your software development process, you are aware of when it is working and when it isn't, you are experimenting to fix the parts that aren't working, and you consciously enculturate new team members.* In our group we have a continuous process of improvement, some of our techniques on process monitoring and insourcing are described in sections below.

The Four Values

Kent Beck states four values that let you decide if you are doing XP right. We restate them here briefly and describe how we apply them to our development.

The four values of XP are 1) communication, 2) simplicity, 3) feedback, and 4) courage. The DOC group supports the four primary values of XP to various degrees.

Communication is very well supported both internally and externally. Within the group, email and impromptu conversations are the primary modes of communication. There are no scheduled meetings, due to the conflicting schedules of group members and success of the informal mechanisms.

Externally, email lists are used to communicate bidirectionally with users and contributors. There are currently four email lists, `ace-users`, `tao-users`, `ace-bugs`, and `tao-bugs`, and two lists that report changes in ACE and TAO bug report status. The email lists are gatewayed to a newsgroup, `comp.soft-sys.ace`, for convenient access.

The email lists work very well because they support asynchronous communication, they are persistent, and they are searchable. Asynchronous communication is especially important given the wide geographic distribution of contributors, e.g. across time zones. Persistence supports the informal use of the lists as a resource for design and implementation issue discussions. Several commercial sites store the list traffic in searchable form, which helps both new and old users and contributors find discussions of interest.

Simplicity has interesting implications for frameworks. The goal of a framework is to simplify application development. However, the framework itself is typically not simple or minimal. Because the cost of framework development effort is amortized over many projects, this tradeoff of simplicity for functionality and complexity is often acceptable. We discuss the issue of simplicity for frameworks further in Section 3.

The huge (> 600) user community tests ACE+TAO quality and conformance to standards, e.g. CORBA, on a very short-term basis. New beta kits are created roughly every other week. This keeps the *feedback loop* very tight. Bugs and non-conformances are reported quickly through the above mentioned communication media.

This feedback has the desired direct benefit to the code base. It has a further benefit in open-source projects: it links users into the development process. Many contributors to open-source projects started as users. Then, they fed back defect reports or enhancement requests. Close contact with, and rapid feed back from, developers encourages users to contribute fixes and enhancements.

The DOC group shows *courage* when developing the ACE and TAO frameworks. Due to the changes coming from customers, performance requirements or standards requirements, e.g. changes in the CORBA spec, main parts of the ORB architecture have seen huge changes. The core development team did not fear applying these huge changes.

Examples of where the team showed courage are:

- Reimplementing the TAO Real-time Event Service
- Restructuring the ORB Core [6]
- Adding support for pluggable protocols [7]
- Large refactorings on the code generation for implied IDL [8]

A good development process supports courage; you can always easily step back from something that proved to not work. Furthermore, a good development process is well defined and documented yet adaptable. We discuss development process aspects in the following section.

3 XP FOR LARGE, OPEN-SOURCE PROJECTS

Applying XP to Open-Source

Open-source development efforts differ from traditional efforts because the distinction between Business and Development is blurred. Business may expect ambitious product features and development schedule, while Development has constraints on resources and/or technology. The group that coordinates development often assumes both roles, especially early in the life of the product. The natural tension between these two roles may not be present. Therefore, formalities such as schedules may not be taken seriously, or used at all. And, the Planning Game is often one sided, because Development is not directly bound by a design contract.

XP relies on a tight feedback cycle, open-source development as well, though it can be hard to identify users. Lack of user identification breaks the feedback cycle. An effective remedy is to actively encourage and reward feedback, for example, by public acknowledgment in newsgroups and THANKS files.

Open-source projects rely on the contributions of (many) developers. Successful, large, open-source projects require many developers at various levels of effort. Over time, as more and more people contribute, the core Development group migrates more towards a traditional Business role. It serves as a gatekeeper for the source base, identifies desirable new features, assigns priorities, and schedules. One or more of these activities can be performed by third-party organizations, possibly for profit.

Applying XP to Framework Development

XP prescribes simple designs. In particular, software should have the fewest possible classes and methods. Frameworks must support multiple applications, and therefore may not be minimal with respect to any one. While frameworks may (greatly) simplify application code, frameworks themselves can be very complex, and not minimal, internally. Therefore, development of frameworks themselves is atypical for XP.

Nonetheless, XP can be used to great benefit when building frameworks. While XP tries to avoid costs for not yet needed functionality, the cost and effort of developing and maintaining good frameworks can be quite large, and must usually be amortized over many application products.

To place framework development in proper perspective, its immediate purpose must be considered. If a framework is being developed for internal use, then simplicity dictates that it only provides the necessary functionality for the immediate target application(s). If additional or modified functionality is required later, then the framework can be refactored. If a framework is being developed for external use, then it is an end-user product in its own right. The business demands on the product must drive its evolution, whether the product is a standalone application, framework, operating system, or any other software artifact.

How do Standard APIs Relate to XP?

In this section we discuss how XP can be applied to build software conforming to standard APIs.

The requirement to conform to a standard API, such as CORBA or POSIX, has a two-fold impact on the developers. On the one hand developers can profit, because they do not have to go through the tough job of domain analysis necessary to come up with the API. On the other hand it can also limit the freedom in implementing the semantics of standard APIs.

As a result, developers can focus largely on internal design/implementation issues, rather than doing time consuming and "mushy" upstream activities, such as trying to figure out what the customer requirements are. In such situations, XP and open-source are based on the notion of *rapid feedback loops* and *community development*, which are really powerful!

Besides the impact on the effectiveness of the developers, standard APIs also have an impact on the metaphor understood in the team. Standard APIs limit the scope the metaphor can come from. There is less of a need to translate domain concepts into software abstractions, such as metaphors; the domain *is* software abstractions.

XP and Design Patterns

Due to the restriction on where metaphors can come from, a replacement needed to be searched for. Ideas, concepts, solutions need to get transported in some kind between minds. As every member in our team is very familiar with design patterns, we use them to communicate our ideas. Design patterns are used to describe the internal mechanisms, the under-the-hood of the outside standard API. They are the common language based on a concise and documented vocabulary.

Pattern languages can be an excellent replacement for metaphors. They communicate a global view, as the metaphors are intended to do.

Some of the design patterns used in the architecture of TAO are: The Reactor pattern [9], used for event dispatching mechanism; the Acceptor and Connector pattern [10], for abstraction of connection management; and the Service Configurator pattern [11] for easy and dynamic reconfiguration. At a higher abstraction level, the Broker design pattern [12] describes the general interaction between clients and servers with their proxies and stubs in between.

Applying XP in a University Research Group

University research groups usually consist of one or more professors, affiliated staff, PhD students, Masters students, students volunteering on research, and visiting researchers. There is no reason to expect that XP cannot be employed in such environments.

However, our experience with XP has revealed some interesting insight, which we discuss below.

Because Masters or PhD programs take between 2 to 6 years, there is a constant fluctuation of people associated with it. New students need to get *insourced* all the time. Besides this, there are also shorter cycles, like exchange students, or visitors, staying anywhere between 2 and 12 months. Intensive mentoring and support from the whole team help to insource new students within a very short time. Every member of the team is always open to questions. Ideas get transported via a common language: design patterns. The first activity of a new member is to study the various design patterns relevant to his/her work. Good source code documentation, external documentation, and research papers about former research projects are necessary to get new members quickly up to speed.

So far we have described the insourcing process within the core development team. In the remainder of this subsection, we describe the insourcing process of the developers external to the core team. People external to the core development team are usually either employees of companies, or research staff and students of other universities, working with the open-source products. Most of them start out as simple users, but soon report the first bug, or enhancement request. After a while they usually dig deeper and deeper, and get more and more involved. Soon they contribute their first pieces to the open-source product.

A requirement for the success of such distributed development teams is that everybody follows the basic principles of the XP, which are:

- *Rapid feedback*: To drive rapid evolution of the system.
- *Assume simplicity*: To best allocate programmer resources given the economics of software as options.
- *Incremental change*: For overall development efficiency (and sanity).
- *Embracing change*: To preserve options while delivering what is most needed.
- *Quality work*: To leverage the natural tendency to take pride in the efforts of an individual, and of the team.

Beside these, we found some additional principles very useful when communicating with each other:

- *Respect each other*: We found it essential to respect the opinions of others when discussing problems, solutions, designs, etc. If people would not show respect for each other, some would turn away in distress.
- *Honor the work of others*: When somebody achieved an important milestone, finished a tough job, or just did refactoring to support the next steps in development, others honor the work by letting them know what a good job he/she did. Such behavior can be a fundamental engine driving people to do a good job the next time as well.

We have seen pair programming in our research group before it was actually made that popular by XP. Students discovered pair programming as a fun way to do their research.[1]

They want to have as much fun as possible while working, just as they want to create quality work. Solving problems as team of two, while have some popcorn sessions and cans of diet Coke^{TM} is just more fun than solving them alone. Within a short time, students realized that they were faster. Not only that, but they also had less errors in their code and learned from each other. Having snacks during these sessions relaxes, and helps developers sustain their enthusiasm for each programming session.

The working environment, the DOC research lab, supports pair programming in an ideal way. At the beginning, this was more of a coincidence, than planned beforehand. The room is large enough to accommodate around 10 people, and everyone has a reasonably fast machine on his/her desk. Fast compilation machines are centrally available, via remote access. Remote access to compilation machines is very common to our team, due to the many platforms the products have to support. Developers are responsible for unit testing their code on various platforms. Usually these are two or three different platforms, it is well known which compilers are the most restrictive and/or non-conforming to the programming language (C++, usually) standard.

The Planning Game

The goal of XP planning is to establish a mutually respectful relationship between the customer and the development team. It abstracts two participants, Business

[1] We have collected some examples in http://cs.wustl.edu/~doc/ACE_wrappers/etc/DOC-way.html

and Development. Applied to the domain of software developed at a university, Business is sponsors who fund the work; Development is clearly the research group itself. Because it is clear that the research group has great interest in getting funding not only for the current projects, but also for future projects, the same kind of interests are prevailing as in every usual software development process. The planning game can be applied the same way to research groups.

Our story cards are stored as Bugzilla [13] entries. The entries can be made literally by anyone with web access. They might contain customer requests, bug reports, or just ideas for refactorings. Additional information is entered continuously, often to the level of specific tasks necessary to address the request or problem. Every time a change is made to one of the entries an email notification is sent to the involved persons, including the customer if prevailing.

Business (the core DOC group development team) and Development (domain experts in the core development team) periodically assign priorities to the entries, this way doing iteration planning. Customers get feedback on the progress of their requests 1) via email notifications on changes of the Bugzilla entries and 2) via issuance (and announcement) of new beta kits.

We found that *on-site customers* are not strictly needed in our environment. There are three reasons for this.

- *The nature of framework software*: is that it alone does not provide any value. It needs to be integrated into applications. But this can be done effectively at the customer sites, too, and does not require to be done at the location of the framework development team.
- *Rapid feedback loops*: by making progress information and results quickly available to customers via mailing the email lists/newsgroup and beta kits, respectively.
- *Close email contact*: involves the customer directly in discussions about the schedule, supported features, and planned enhancements.

This allows us to have the profit of customer involvement and immediate feedback without having an on-site customer.

DOC Group Roles

In the DOC group, everybody is a programmer; there is nobody who only does monitoring or supervising. The following list enumerates the rest of the roles and how we fill them.

- *Customer*: The sponsors funding the research work are the customers in our case. They have an interest in getting work done, whereas mostly the scope and time can vary.
- *Tester*: We have a dedicated tester. This is the person who broke the last set of builds before the release of a new beta version. By this rule, everybody -- well almost everybody -- gets a chance to play the role of a tester. Internally to the group, the tester has a nickname, the Build Czar or Build Master. This comes from the tradition that the one monitoring the automated tests is also responsible for creating the next beta version. While all team members are responsible for building and testing prior to committing changes, the Build Czar ensures that builds in fact remain clean.
- *Tracker, Coach*: The head of the research group, together with the affiliated staff are the trackers, they watch the schedule and make sure progress is made to keep customer promises. They also play the role of coaches, making sure the process works as a whole.
- *Consultant*: Everybody with a great new idea, technique or technology plays from time to time the role of a consultant by introducing them, bringing them to the attention of the group, though the consultant is never named as such.
- *Big Boss*: This role is greatly played by the director of our research group. In general, developers make their own decisions. However, when there is uncertainty or lack of agreement, they may request a decision from the Big Boss.

Code Ownership

In our group we have collective code ownership as XP does. Generally, everybody is fixing anything that does not work or needs to get enhanced. This holds true especially for ACE. Regarding TAO we have some exceptions, here the ownership is more for resource (people) allocation and efficient use of those resources.

In some source code areas only one or two persons are making changes. There are two reasons for this. One is the research of these persons, the other is, that it is just more efficient because the person is already familiar with it. Proper documentation can minimize the risk and effort needed when other people are forced to do changes.

Insourcing

In software development some people talk about outsourcing, some kind of fashion in big companies, nowadays. Kent Beck talks about *insourcing*, the process of getting new people involved in development. For a research group, especially at a University, it is mandatory to do insourcing. Students come, students graduate, there is a continuous fluctuation in the group. There is always somebody new to the group, getting insourced. This process needs to be optimized for a research group to succeed. New members of the group get slowly introducing in the environment; they get their first chances on small work packages, mostly with mentoring from experienced students. Pair programming brings them quickly up to speed. Finally, after some weeks they start programming on the hard stuff, e.g. the ORB core, in the case of TAO, themselves.

Insourcing at the level of open-source development gets even more challenging. It is more rapid and often of shorter term. People sometimes stay only for the term of a product evaluation and/or project. Key prerequisites are proper documentation, including tutorials and source code documentation.

Testing
Developing ACE and TAO, we always have a test handy to verify our actions. In the case of ACE, we have a special use case for the framework in mind, which triggered the extension/refactoring of it. In the case of TAO, we have test cases telling us when we comply with the CORBA specification.

Our tests use Perl [14] scripts. We use Perl because it is available for nearly all platforms that we support. An OS-independent testing tool is essential because there are so many platforms, with sometimes subtle differences. Monitoring the output of a Perl script is not as easy as watching a bar coloring green or red [15], but it is sufficient. We use conventions such as program exit status of 0 for success and 1 for failure to enable automated testing.

Unit tests are written for almost each class in ACE. TAO is completely based on ACE, which assures its high portability. Unit tests in TAO test functionality not already tested in ACE.

Functional tests are provided by the customers and our research group. The customers regularly run their functional tests -- in many cases the test is part of their actual application -- against the latest beta kit.

Our functional tests demonstrate compliance with the requirements specified by customers. These functional tests are collected in our *regression test suite*, which gets run periodically, as often as every three hours, and on demand. The suite executes on multiple platforms concurrently, to make sure the developed code is runnable on all platforms. The suite searches output for potential problems and emails these to the group.

Refactoring
These days, ACE has pretty much matured; most refactoring is done on TAO. Refactorings are triggered by new versions of the CORBA specification, requests for support of wider areas of the CORBA specification, and performance optimizations - real-time ORBs just cannot be slow.

Focusing on real-time environments, refactorings aim these days at subsetting efforts, meaning things need to get more and more modular. The footprint needs get small enough in order to fit applications using the ORB on small devices. One of the most common refactorings is the removal of dependencies to enhance modularity, which helps reducing the footprint. Such refactorings are often not trivial.

Changing interfaces is very problematic for frameworks already in wide usage. Sometimes it might even be reasonable to set up a new framework. Because ACE is in widespread use, major interface changes are unacceptable. However, interface changes are desired to support lighter weight subsets for embedded systems. Our approach in this instance is to initiate development of a separate product, ACELite, though that is not a commonly applicable solution for framework software.

40-Hour Week
The 40-hour week guideline holds true in every environment, the DOC group is surely no exception. One thing that might be different is that people tend to be in lab quite a long time, as many as 60 hours a week. But that does not mean that they are working all these hours. The lab of the DOC group can be seen not only as a working place, but also as a living place. Students do their homework, email friends, have discussions, and these are not only about software development.

When people do not follow the guideline and work long hours, e.g. due to exams coming up or the creation of a major release, the care and confidence in developing code decreases dramatically. Learning from these experiences, the students gain something for their later working life. It prepares them to be sensitive to working for excessively long periods of time.

XP for Large Projects
XP is intended to be used by small-to-medium size teams [16]. However, our experience with ACE+TAO suggests that it could be successfully used on large projects, which we loosely define as involving 20 or more people. In this subsection, we address the question: Does XP scale?

Communication is one of the most critical issues for large projects. It is well known [17] that communication overhead can dramatically reduce the productivity of a development team. Besides the largeness, there is also the distribution aspect of an open-source development team. It is impossible to convene people for meetings. Asynchronous communication media such as email and newsgroups are preferred, because they do not require simultaneous availability. To not get chaos this way, some people need to play the role of a moderator, these are mostly the people of the core development team. Timely replies are admired by the submitters, and therefore this is an unnamed rule in our communications: respond as quickly as possible. This encourages further communication, and assistance with solving problems or designing enhancements. In addition, due to the asynchrony, question/answer sets might have to bounce multiple times before an issue can be resolved.

Source code control is invaluable for any serious software development project. The DOC group relies on CVS [18], as an example. We find it invaluable for documenting every file change, for maintaining file versions, for concurrent development, and for support of branches. Another important use of configuration management tools on large projects is bug isolation. While we have not yet resorted to an automated defect isolation approach, e.g. delta debugging [19], we have used simple scripts to crudely isolate specific problems.

Defect tracking is essential for large software development projects. The DOC group uses Bugzilla [13] for tracking problem reports and enhancement requests. We currently do not require the use of problem reports, for historical reasons (all changes are documented in ChangeLog entries). However, a refined development process might well require a report for *every* change. Tracking systems support searching and categorization, contributing to a better development process.

The third component of our development process is a clear definition of the process. The core of this process is a set of steps that must (well, should) be followed for every software change. While this sequence is not necessarily novel, we show it here to provide an indication of the rigor in our process.

1. Every change to ACE+TAO must have a bug report. *Changes* include fixes, enhancements, updates, and so on.
2. Create a bug report.
3. Accept the bug report if you are going to implement the change.
4. Implement the change in your workspace(s).
5. Test the change sufficiently to demonstrate that it both does what is intended, and doesn't break anything. The test may be as simple as building and running the ACE tests on one platform, or as complicated as rebuilding and testing all of ACE and TAO on all platforms that we have.
6. Create an appropriate ChangeLog entry.
7. Commit the change using a ChangeLogTag commit message.
8. Respond to the requester of the change, if any. This must be done *after* committing the change.
9. Make sure that the requester is listed in the THANKS file.
10. Update the bug report to indicate resolution.
11. Monitor the next round of build/tests for problems with your change.
12. Respond immediately to reports of problems with your changes.

Coding standards are necessary to support rapid familiarization and refactoring [16]. Above all, standards must emphasize communication. We have found this to be extremely important on large, open-source projects. Consistency between developers is even more important given their sometimes vastly different experience and goals. To encourage new contributors, there must be a low entry barrier to understand existing code. And, consistent coding style assists the gatekeepers in evaluating contributed code for inclusion in the product.

A serious software development project must contain a configuration management component. It must provide read/write access to some developers, but read-only access to others. The source database must reside at a well-know location, and be accessible via email, the web, an intranet, and/or other convenient means.

Ideally, the configuration management approach relies on a problem/feature-tracking component to rigorously follow up on every change to the product. To avoid duplication of change documentation, for example, we continue to maintain that in ChangeLogs. Source code control commit messages consist simply of a *ChangeLogTag*, or link to the appropriate ChangeLog entry. We wrote a short Perl script to view the source code control (CVS) commit messages, expanded to include the appropriate ChangeLog entries. In some cases, problem reports contain more detailed or unnecessary information for the ChangeLog. Therefore, ChangeLog entries contain a problem report identifier (Bugzilla Bug ID).

With a large user community, typical for open-source products, structured feedback is essential. When new beta versions are released users get notified via the aforementioned email lists. Triggered by this, most of them download and build the new version. Integration with their tests and applications then shows whether existing bugs were properly fixed, or if even new bugs were introduced. For several years, we did not impose any structure on the form of bug reports or queries. Necessary information was often missing, and therefore required one or more requests for more information from the user. We added a problem report form, which requests such data as host (and target, for embedded systems) platform type, compiler, phase at which the error occurred, etc. The structured form draws out the necessary information up front. In addition, it is much easier for developers to rapidly find what they are looking for in a bug report when it is structured.

Problem report forms for XP projects should contain product version identification. As part of the ACE and TAO kitting process, version numbers are automatically inserted into the appropriate forms. This is especially important for XP projects, with their many releases. There is no need to conserve version identifiers; a new one should be assigned to each iteration[2].

[2] We use an automated release script to assign a new version identifier, update version information in the product, assign a source control tag, and create the product kit.

The large user community of an open-source product is invaluable for testing. Bugs are found almost immediately after release. A large, distributed, heterogeneous test "organization" stresses a system much better than a static regression test suite. Furthermore, the testers often track down and fix the problems, saving both the effort of the core development team and clock time. The user community triggers a continuous introduction of fresh ideas and techniques to the development team. So there is a huge base of coaches, though coaching in a limited way.

Our experience is that XP can scale well. The reliance on metaphor, testing, and self-documenting code contributes to scalability. And a streamlined, tool-supported development process is essential to scalability. Even more important are the XP values: communication, simplicity, feedback, and courage; all support the cooperation that is necessary for the success of large development projects.

The one component of strict XP that does not scale is pair programming. We consider a possible augmentation in the following section.

4 REMOTE PAIR PROGRAMMING

One of the more prominent features of XP is pair programming. Unfortunately, open-source development does not support pair programming for the following reasons:

- *Remote developers*: Many of the participating programmers are physically distributed all over the world. The Internet is connecting them via email and configuration management.
- *Transient developers*: Often programmers outside the location of the core development team are part of it just for short term, mostly for the time of a project. The terms these transient developers are participating are often too short to profit of pair programming.
- *Developers interested in only a small portion*: Programmers contracted by a company participating in the open-source development can focus only on a small portion of development. They have special constraints regarding the time and effort they can spend developing for everyone. Exchange of information is limited to this part of the solution.

Pair programming profits from the fact that there is not only one mind working, but two. Two minds develop ideas around the same set of problems, but from a different kind of perspective. Inductive reasoning suggests that more than two minds could be even better. On the first thought one would say yes, remembering Brooks law, one says usually, maybe, or no. The problem is the communication overhead, which gets bigger and bigger, the more and more people are involved. This is the theory, now come some facts we experienced.

In the following paragraphs, we elaborate why programming environments, such as open-source environments, are successful. Clearly, traditional pair-programming does not always work well, for example, with distributed development teams. Still, two or more people can successfully work on the same set of problems. The deficiency one notices first is the presumably high overhead in communication. On the one side, from the separation of locations, eye-to-eye communication is just not possible, on the other side from the number of people working on the ideas. It is clear that three people have to communicate more than two. The solutions to this are development environments, as for example the design fest theme *Concept Development* at OOPSLA 99 [20] suggested. Such environments bring people together from various locations, via editors, chat channels, and eventually voice channels to develop *concepts*, e.g. program designs and implementations, as a team. This requires minimal communication overhead. The communication overhead is small due to the asynchrony and common documents. Documents are shared in real-time, obviating meetings, which are otherwise necessary for communicating them. People at different locations can work on a problem as if they all would sit in front of one machine.

Our successful experiences are based on such kind of development, though our environment has not been that complete, as described above. Using our configuration management tool and email, we provide almost immediate updates on the common documents, including source code. Developers working together are sometimes only people of the core team, but sometimes also developers from outside, people who reported a bug, for example.

Due to asynchrony, of email for example, we gain some decoupling over the traditional pair programming approach. But there are also some limitations to remote pair programming. These are:

- *Weaker concentration*: The remote pair programmers are not physically adjacent, and therefore likely not as involved in the programming process. If there is sufficient communication delay, then pair programming could degrade to code review. Good support can help avoid this degradation to the extent that it can supply the immediate communication offered by pair programming.
- *Speed-up*: Pair programming can speed up development, because two people make fewer mistakes and often have a better design in mind than just one. So fewer mistakes reduce the time of debugging. Furthermore, the better design pays off when doing refactoring, maintenance in the long term. Can it be shown that open-source development using remote pair programming can further speed up development due to interleaved schedule of the participants working on subsets?
- *Learning/Coaching* can be done effectively using pair

programming, due to the tight feedback loop between someone new to development in the project and someone experienced. With open-source development, there is naturally not such an optimized feedback loop. However, remote pair programming might alleviate this and leverage it to the same position in coaching.

Remote pair programming is a prerequisite to true support of distributed XP. High-speed video, audio, and data transfer, with reasonable but not exceptional quality, seem necessary. We plan to experiment using TAO's AV Streaming Service; its predictable and high performance are ideally suited to the remote pair programming application.

5 CONCLUDING REMARKS

Based on our experiences, we have found XP to be quite suitable for large, open-source, framework development projects. We have found that XP can be applied successfully and beneficially to such projects. XP works with projects that can incrementally grow to be large. It works with open-source projects, given their rapid feedback cycles. And, it works for framework development, even if the frameworks are not simple.

Successful open-source development with XP is based on:

- high-standard coding guidelines combined with a proper gatekeeper process,
- tight feedback cycle via active encouragement and reward, and
- well-maintained user groups with fast responses.

Building frameworks with XP we had good experiences with:

- using design patterns and pattern languages as metaphors, and
- making the prevalence of standard APIs an advantage, instead of a limitation.

University environments additionally need to:

- optimize insourcing,
- use pair programming for mentoring, and
- have collective code ownership.

Especially for large projects we found the following two mechanisms valuable:

- asynchronous communication, and
- easy to follow development process.

One practice that we would like to augment is pair programming. Large projects, including ACE and TAO, often involve distributed development. This is especially true with open-source projects, which encourage a very large number of developers. The vast majority of these developers has intimate knowledge of only a small portion of the system, and often is directly involved with development for short time periods. Therefore, it is not feasible to practice traditional pair programming. We would also like to explore alternatives, including advocates in the core development team and remote pair programming.

ACKNOWLEDGMENTS

We would like to thank Frank Buschmann, Lutz Dominick, Jeff Parsons, Christa Schwanninger, and Nanbor Wang for their insightful comments on drafts of this paper.

REFERENCES

[1] T. O'Reilly, *The Open-Source Revolution, Release 1.0*, http://www.edventure.com/release1/1198.html, Nov. 1998.

[2] D. C. Schmidt and T. Suda, *An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems*, IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems), vol. 2, pp. 280–293, December 1994.

[3] D. C. Schmidt, D. L. Levine, and S. Mungee, *The Design and Performance of Real-Time Object Request Brokers, Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[4] Wiki Web, *Are You Doing XP*, http://www.c2.com/cgi/wiki?-AreYouDoingXp, 1999.

[5] Wiki Web, *Wiki Web*. http://www.c2.com/cgi/wiki?WelcomeVisitors, 2000.

[6] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, *The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging*, in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[7] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, *The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware*, in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[8] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, and M. Kircher, *Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks, C++ Report*, vol. 12, Mar. 2000.

[9] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[10] D. C. Schmidt, *Acceptor and Connector: Design Patterns for Initializing Communication Services*, in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[11] P. Jain and D. C. Schmidt, *Service Configurator: A Pattern for Dynamic Configuration and Reconfiguration of Communication Services*, in The Pattern Languages of Programming Conference (Washington University technical report #WUCS-97- 07), February 1997.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Pattern*s. Wiley and Sons, 1996.

[13] The Mozilla Organization, *Bugs*, http://www.mozilla.org/bugs/, 1998.

[14] L. Wall, T. Christiansen, and R. L. Schwartz, *Programming Per*l. O'Reilly, 2nd ed., 1996.

[15] Erich Gamma and Kent Beck, *JUnit*, http://www.xProgramming.com/software.htm, 1999.

[16] K. Beck, *Extreme Programming Explained: Embrace Chang*e. Reading, Massachusetts: Addison Wesley Longman, Inc., 1999.

[17] F. P. Brooks, *The Mythical Man-Mont*h, Reading, MA: Addison-Wesley, 1975.

[18] SourceGear Corporation, *CVS*. http://www.sourcegear.com/CVS, 1999.

[19] A. Zeller, *Yesterday, My Program Worked. Today, It Does Not. Why*? in *Software Enginering – ESEC/FSE '9*9, Lecture Notes in Computer Science Vol 1687, Springer Verlag, Sept. 1999. Also published as ACM SIGSOFT Software Engineering Notes, Vol. 24, No. 6, November 1999.

[20] OOPSLA 99, *Design Fest*. http://designfest99.instantiated.on.ca, 1999.