Introducing Agile Development into Bioinformatics: An Experience Report

By David Kane, SRA International (david_kane@sra.com)

Abstract

This experience report describes our efforts to introduce agile development techniques incrementally into our customer's organization in the National Cancer Institute and develop a partnering relationship in the process. The report addresses the steps we have taken not only to deploy the practices, but also to gain customer support for them. It addresses variations we have used to adapt to our customer's environment, including our approach to involving customer personnel at remote locations. We also address challenges we still must face, including how best to manage a product-line with agile development techniques.

1 Context

Our team is responsible for software development at the Genomics and Bioinformatics Group in the Laboratory of Molecular Pharmacology of the National Cancer Institute in Bethesda, Maryland. [1] The group, lead by Dr. John Weinstein, includes experimental biologists as well as those who focus on bioinformatics exploration. Dr. Weinstein has a long history of developing tools and techniques for the biology community in general, and cancer researchers in particular. [2][3][4][5][6] The group's approach has been to encourage a close collaboration between the experimental and computational components of the lab. The needs of the experimental biologists drive the priorities and directions of the tool development undertaken by the group. Although the internal tool needs of the group provide the primary drive, the tools are generally made available to other researchers. The assumption is that the kinds of problems encountered by the experimental biologists are similar to those that are also encountered by others in the field, and in fact, investigators around the world use the tools developed by the lab.

In 2000 the lab experienced staff turnover, and in October 2000 a new team of software developers from SRA International was hired. [7] This was the first time that the lab had incorporated contractors to any significant degree. The team develops new software, maintains previously developed tools, and integrates tools and components from other labs. The team works as part of a staff augmentation contract for which there are no software deliverables. The work started with a staff of 1.2 engineers, and reached the current staffing level of 4 engineers and a system administrator in August 2001. The team works most directly with bioinformatics specialists in the customer's organization, but also with the other biologists in the lab. The team often works directly with external collaborators of the lab as well.

The new software tools being developed by the team are primarily written in Java. Some of the tools are web-based. Two new tools have been released since the team has been in place, GoMiner and MatchMiner. [5][6] Other tools are applications, both GUI and command-line. The development and operating environments are a mix of Windows, Mac OS X, Linux and Solaris. The legacy applications maintained by the team are mostly written in PERL and S-PLUS. The most important of these applications are CIMMaker and MedMiner. [3][4] The team also maintains the group's web site. [1]

2 Forces

There are a number of forces that push the overall effort. The applications built and maintained by the team support the large and complex domain of the biology of cancer. It has been estimated that there are 30,000 to 60,000 human genes. And because each gene can produce different mRNA splice variants, each of which codes for a different protein, there appear to be well over a hundred thousand human proteins. [8] In addition, there are genomes and proteomes for mice and other model organisms that are also used extensively in cancer research. There are of course many investigators in this field, and so the state of knowledge in the domain is advancing rapidly. A great deal of this information is available in public databases. A GB of data is regularly pulled from these repositories to populate the lab's tools.

The effort is also characterized by dynamic requirements. Scientific research is, by its nature, an exploratory endeavor. As researchers gain new insight into their investigations, new avenues for inquiry come into focus. This dynamism means that new requirements for tool support are constantly emerging. Cooperation and competition can also change priorities. Supporting collaborations with other labs can cause priorities to change. Similarly, keeping pace with other scientists to be able to publish novel work can also drive changes in priorities.

Because many scientists are visual thinkers, many of the tools require visualizations to help users gain insight into the data being analyzed. Other investigators are more mathematically oriented, so these visualizations need to be consistent with statistical representation and analysis of the data.

The results of the tools, as well as the tools themselves, are included as part of scientific publications. This means that the results of the tools must be consistent and reproducible, so that they will pass the scrutiny of peer-reviewed journals and other scientists.

3 How We Started

The SRA team started work on a proof-of-concept for a new visualization tool, and there were several questions that the customer wanted to answer with this proof-of-concept. First, the customer wanted to validate the contractor model in general, and this team in particular. In addition, the customer wanted to investigate a pair of visualization technologies, Scalable Vector Graphics (SVG) and Spotfire. [9][10] For the proof-of-concept, we did not attempt to assert our own practices, but instead we worked with the practices that our customer already had in place. There was a general vision for the tool, but the details and requirements had not yet been worked out. The requirements were explored as the tool was built. This early effort was characterized by many demonstrations. Some of these demonstrations were for the customer's staff, but others were for the customer's sponsors. Preparing for these demonstrations was usually painful because of the lack of configuration management and integration practices. It often took a long time to integrate the work from different engineers on the team. The tool would work correctly on one machine, but not another.

4 The Turning Point

Despite some pitfalls along the way, the proof-of-concept was a success. The customer was satisfied with the tool itself and the performance of the team. The sponsors were satisfied, and another increment of funding was secured. The funding expanded the team size as well. One thing learned from proof-of-concept was that to realize the vision of the tool at an operational level was a more ambitious task than first realized. Substantial database and software development work would be required to bring the tool to production.

At this point we decided to move the development team toward an agile development approach. I had been studying the methods for some time, and the work in the lab appeared to be a good match for the sweet spot of these techniques. The team agreed that an agile approach would be a good one to take. The project also appeared to be at a good point to introduce a change in approach. The team had been presented with the ambitious objective of operationalizing the proof-of-concept, and the team had established trust and confidence with the customer. When then the first steps of adopting an agile approach were proposed to the customer, the suggested approach was accepted.

5 Our Incremental Approach

Our team wanted to adopt agile methods, but we also needed to continue to make visible progress towards our customer's goals. We also had varying degrees of knowledge about agile methods on the team, and we needed time to develop our skills. We decided to adopt an incremental approach to introducing agile methods. This incremental approach meant that the team members could learn the practices a few at a time. The incremental approach also enabled us to deliver value and build further trust as we deployed new practices. It also facilitated the development of a partnering relationship with our customer. However, because many agile practices are intended to work together, we tried to avoid introducing practices that would throw the team out of balance.

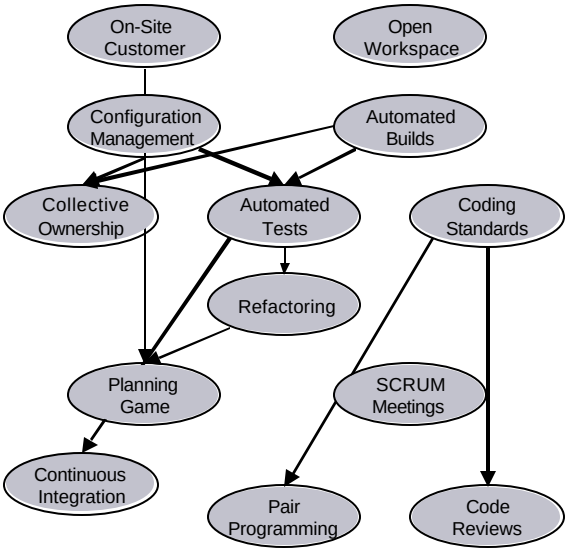

Figure 1: Agile Practices Adopted and the Dependencies We Considered

6 Core Practices

The team has adopted a number of agile development practices, as illustrated in Figure 1. The practices were adapted from a combination of Extreme Programming and Scrum. [11][12] The figure also illustrates our perspective on which practices provide pre-requisites for other practices. The practices at the top of the figure are those we had in place first, and the practices at the bottom of the image are those we have introduced most recently.

6.1 Practices for Free

Before we started any effort to introduce agile methods, there were already two practices already in place. Largely out of necessity, the team shares an *Open Workspace*. [13] All of the developers work in the room, as do the customer's bioinformaticians. The experimental biologists are in a lab space across the hall. Of course, this facilitates having an *On-Site Customer*. [11] At different stages of the work, the person who has been the primary point of contact for the team has either worked in the team's shared work space or in the lab across the hall.

6.2 Configuration Management (Started August 2001)

One of the first things we did when we started our agile development effort was to get our code under better *Configuration Management*. We used CVS as our primary tool for version control. [14] At first, we focused on just the code for the tools under active development. In addition to managing the code, we achieved a VERSION-CONTROLLED ENVIRONMENT, by keeping the development environment under configuration control. [15] The code is managed as a MAINLINE with PRIVATE WORKSPACES for each developer. [16] At first we checked code into the repository and integrated on a schedule of twice a week. Much later on we moved all of the legacy application and web site code under configuration control as well.

6.3 Automated Builds (Started August 2001)

We used the Ant tool from the Jakarta project to write scripts describing our *Automated Build* processes. [17] These scripts were written at about the same time as we were establishing *Configuration Management*. The Ant scripts were effective at tying together the code and development environment that we had checked into CVS. In addition to builds, we also used Ant to automate other processes for data acquisition and processing tasks. The combination of *Configuration Management* and *Automated Builds* made an immediate impact on the project. It made our software more consistent between the development and the demonstration machines. It also reduced the time needed to integrate our code to prepare for demonstrations.

6.4 Collective Ownership (Started September 2001)

Before we had *Automated Builds* and *Configuration Management* practices in place, we had a component ownership model that made it easier to integrate. Once we had these practices in place, we moved to a *Collective Ownership* model. [11]

6.5 Coding Standards (Started September 2001)

Although we had a corporate *Coding Standard* before we had started the agile development effort, when we started to get our code under configuration control, we made an effort to refocus on these standards. [11] We created a tailoring document that listed project-specific revisions to tune the corporate standard to our working styles.

6.6 Automated Tests (Started October 2001)

Our next major activity was to introduce *Automated Tests*. [11] Our core tool for writing the tests is JUnit. At first, we selected a set of

core classes from our existing code base for which to write tests. Some of these classes needed to be reorganized to have a testable interface. We did not attempt to write tests for all of the existing code, but we adopted a rule that as new classes were written, they would have test cases written as well. Similarly, if classes were modified that did not yet have test cases, new tests would be written as part of modifying the class. The developers run the tests as part of writing software, and during integration. We complemented the unit tests with system tests. We used JUnit in combination with MaxQ for these system tests. [18][19]

6.7 Refactoring (Started December 2001)

As our test case coverage grew, we started *Refactoring* the portions that had the tests. [20][21] There were a number of "smells" that had accumulated in the code up until this stage. In particular, much of the code we had written in the early proof-of-concept stages of the project was more tightly coupled than was desirable. We found that the stability and maintainability of the application improved. We also became less hesitant about making changes to existing portions of the code base. This past fall we adopted an integrated development environment with many refactoring tools built-in, IntelliJ. [22] The tool has been particularly helpful in automating simple, but tedious refactorings, e.g. renaming variables, methods and classes.

6.8 The Planning Game (Started March 2002)

Up until this point of our effort, the agile development practices we had been introducing had dealt primarily with the activities within the software development team. However, the next practice required changing the dynamics between the development team and the customer; we introduced a flavor of *The Planning Game*. [11] Before *The Planning Game*, we managed our tasks and requirements informally, through email and conversations. We introduced to the customer the idea of using *The Planning Game* approach, and the customer agreed to try it out. One of the biologists thought the approach made a great deal of sense and commented that she was unsure why anyone would want to do anything else.

We described each task on an index card. We did not limit our tasks only to user visible features, but, we also included features for systems infrastructure. For features that were not well understood, we would create tasks for analysis to investigate the candidate features. This broad notion of tasks is more like the work items described in Scrum than the user stories in XP. For each task we estimated the effort required to implement the task in the form of story points. That is, we sought to estimate the relative effort of the tasks. A member of our customer's staff selected tasks for each two-week iteration. We measured the project velocity by counting how many points we completed during each iteration. This project velocity was given as the budget to our customer for selecting tasks for the next iteration.

6.9 Scrum Meetings (Started March 2002)

Throughout the effort to adopt agile efforts, we had made a conscious effort to establish a sense of rhythm for the project. [23] Our initial twice weekly integration and the two week iterations of *The Planning Game* were examples of maintaining a regular rhythm. Another practice that we adopted that also fit this pattern was daily *Scrum Meetings*. [12] The software team meets every morning to report what work had been done since the prior scrum meeting, what work was planned to be done before the next scrum meeting, and obstacles currently faced.

6.10 Continuous Integration (Started October 2002)

We had been manually integrating our code baseline at a pace of twice a week for more than a year when we decided to adopt *Continuous Integration*. [11] We used the Cruise Control tool to monitor our CVS repository for changes to the code base. When changes are detected, the Cruise Control tool invokes scripts to checkout and test the code and then email the results to the team.

6.11 Pair Programming (Started January 2003)

This past January the team brainstormed about what resolutions for new or modification practices that the team should adopt. *Pair Programming* was on everyone's list of practices to adopt. [11] However, for a number of reasons we were unwilling to adopt the practice as our only mode of operation, so we took an incremental approach. We selected one day a week to be pair programming day. We rotate partners each week. We also made an effort to recognize ad hoc situations when ad hoc pair programming would be useful, such as when a

developer is exploring a new technology or section of the code base.

6.12 Code Reviews (Started January 2003)

The other approach that surfaced from our New Year's resolutions discussion was our decision to incorporate *Code Reviews*. [24] However, we did not want to use code reviews as a gate in our development process. Instead we wanted a way to fit code reviews into our other practices, and to emphasize the ability of code reviews to share knowledge among the participants. For each iteration, one developer takes the role of "author" for the review. That author gets to pick tasks for that iteration first. The author then selects 2 to 6 classes to be reviewed from those that author would expect to touch to complete the tasks. The others on the team are informed of the selection, and a review takes place on the fourth work day of the iteration. A typical code review meeting is held, and the author is then responsible for addressing the issues raised in the review.

We introduced limited *Pair Programming* at the same time as we were introduced *Code Reviews*, and we found the techniques complemented each other. While both approaches provide feedback from other developers, we found the perspective of reviewing and discussing code offline useful for two reasons. Issues raised during the review meetings were shared immediately shared with the team. It was also easier to recognize implementation issues that occurred across larger portions of code.

One reason we adopted code reviews when we did was that the team was very pleased with the performance of IntelliJ. [22] In the past, applying all of the good suggestions from a code review was very tedious. However the refactoring power of IntelliJ in combination with the test cases have made our code reviews more effective and useful. With these tools, incorporating the changes suggested by the reviews takes much less time.

7 Process Interruptions

Although we believe we have been successful with our incremental approach to introducing agile methods, we have encountered a number of obstacles that have required us to take a step back to re-evaluate our approaches. These issues cut across a number of the practices areas described here.

7.1 Customer Relocation

For nearly a year we had been using *The Planning Game* for selecting our work for each two-week iteration. The person on our customer's team who had been the "shopper" responsible for prioritizing the work and explaining the domain to the team moved to the west coast. She was going to remain a part of the customer's staff and was very interested in continuing her role guiding the software development direction. We realized that while our index card approach worked well when we were co-located, it would not be a feasible model for this new distributed environment.

We built a tool we call CardMiner to address this problem. We had examined other tools that were available at the time, but we were not satisfied with them. We wanted to maintain the index card metaphor, and we also wanted to maintain the experience of looking at the cards on a table. CardMiner supports both ideas. It even has a tabletop UI that simulates how we would lay out cards during estimation and selection. Figure 2 illustrates the table-top view of CardMiner. We have a projector and display screen in the common workspace so we can easily gather around this "virtual" table. We are still in our preliminary stages of using this tool. We have gone through about three iterations. So far the tool has satisfied our goals of working with our liaison on the west coast, however, we have not yet achieved the same sense of fluidity with the tool that we had found when working with the index cards.

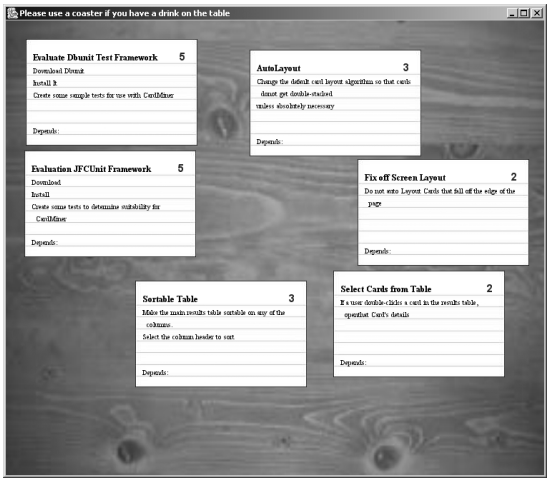

Figure 2: CardMiner Table-Top View

7.2 Inheritance

Our customer's lab collaborated with a university one summer. In a feverish pitch of

programming, the students who worked as part of the collaboration built a prototype tool with a Java Swing user interface. The application looked great and had significant merit as an analytical tool, but there were several bugs that needed to be addressed as well as desired enhancements after the students had returned to their school. Our team was asked to pick up maintenance responsibilities for the tool. We discovered that the tool did not have any tests written for it, nor were the component boundaries inside the tool particularly well suited to testing. We needed a way to adapt our agile development processes to make the desired corrections and enhancements to the tool.

One of the desired features was the addition of a command-line interface that mirrored the features of the GUI. We extended the tool to provide such an interface, while making a minimum of changes to the core code. We then used the command-line interface to write a comprehensive set of systems tests. In designing the test cases, we included unit-test-like conditions by ensuring we exercise important paths in the core units. Once we had these tests written, we started implementing additional changes to the application. We started refactoring the classes into ones with more testable interfaces, and writing the test cases for them.

7.3 Re-estimation

As described in the section above on *The Planning Game*, we assigned story points when estimating the effort required to complete a task. We got better at estimating as we estimated more tasks. For example, we got better at estimating not just the work required to complete a feature, but also the work required to write suitable tests. However, we began to notice that cards that had been in our collection for a very long time often had bad estimates. That is nine months earlier we might have thought a task would require four story points, but today if we were to estimate the same card, we might estimate it at seven points. For a while, we tended to discount the older cards as good baselines for new estimates. At the time, we were moving from our index cards to our CardMiner tool. We decided that it would be silly to go through the trouble typing in all of the cards from our backlog with bad estimates. We re-estimated all of the cards in our backlog. We found going through this exercise both useful and tedious. We asked our customer for a break from development to make the transition. We found duplicate cards. Some estimates went up as we factored in additional work implied by the features of the task. Other estimates went down, especially in areas where we were on new ground when we made the first estimates. We also found duplicate cards and tasks that had become overcome by events. For an ongoing engagement such as this one, we found it helpful to revisit our backlog. We may find it useful to do again in another 12-18 months.

8 Open Issues

Although we are pleased with the progress we have made using agile methods, there are a number of issues that we hope to address as the project progresses.

8.1 Product-Line Management

We now have three distinct tools under our active development baseline and several other legacy applications as well. It is a challenge to balance the priorities of these tools and support them with a relatively small team. Our current approach is to manage the tools are that are actively under development from one shared pool of story points. This pool directs the activities of most of the team. One software engineer is dedicated to the operation and maintenance of the legacy applications. The legacy applications do not have automated test suites, and so we are not comfortable with including those tools in *The Planning Game* process.

The challenge with the arrangement is that there is no explicit structure to help the customer manage priorities across the tools. We have taken some simple steps, e.g. we categorize the tasks to be completed by tool. It would be helpful if we had an approach to better help our customer choose between moving forward a little on all of the tools or to focus on one particular tool.

8.2 Database Testing

While our goal is to pass our tests always during integration, we do fail tests from time to time. The most frequent case of a test case failure is a database synchronization problem. Our tools are typically query-oriented, and we have a set of test data we use to verify our data loading and data querying components. However, even though our test data are a fraction of the size of our production data, we do not reload the data into the database during each test. When our test data are stable, the arrangement works well. When we have changes to the test data, we manually rerun the dataload on each of the

developer and test machines, and this is where things sometimes get out of sync. We are considering adopting Dbunit to see if it provides a better way to manage our test data and validate our database. [25]

8.3 UI Testing

Our user interface layer has proven to be the most difficult to test. There are a number of different technologies used in our UI layer including Java Swing, Javascript executed in a web browser, and Javascript executed in an SVG Viewer. We also have a text-based command-line interface for some applications. There are a number of issues with these various technologies. When we started this effort test frameworks for testing Swing components were in their infancy. We have focused on testing code underneath the UI later. Now that tools such as JFCUnit have been further developed, we are going to consider incorporating it as well. [26]

Simulating web browser behavior is another challenge. While it is fairly straightforward to test the server components of web applications by generating appropriate HTTP requests, simulating behavior that is executed in a web browser is more difficult. While there have been some attempts to build Javascript-aware testing frameworks, our assessment of the tools has been that while they do execute Javascript correctly, they do not yet provide sufficient simulation of the components in a browser environment for us. These tools do not begin to approach the issue of web-browser-specific behavior. We will continue to monitor the available tools, and we expect to be able to test basic web browser behavior in the not too distant future. We do not expect to see low-cost or open-source tools to address web-browser specific behavior, or test tools that can interact with plug-ins such as the Adobe SVG Viewer.

We have a number of command-line tools. Launching our command-line tools within the test environment is not a problem. However we have not found a good way to simulate user command-line interaction within the test environment. What we would like to have is a cross-platform tool similar to an UNIX answer file that can be integrated into JUnit.

9 General Observations

There are several observations that we have drawn from our experience with agile methods. Most significant is that we believe that it is possible to introduce agile methods incrementally. Such an approach certainly requires care because many of the practices reinforce and complement each other, but it has worked for us. Our customers became partners in the agile development approach. The incremental approach built this partnership by enabling the team to demonstrate results and build trust. This created an environment conducive to the more controversial or disruptive practices.

We were also fortunate that we had a contract in place that facilitated the use of agile methods. Since the work was completed as part of a staff augmentation project with no explicit software deliverables, the contract was not an obstacle to making the content of the developed software flexible.

We also believe that agile methods are a very good match for science-driven software development. The rigor of the automated testing approaches satisfies the need of science to have reproducible, correct results. The flexible approach to requirements is a good match for the exploratory nature of science.

10 Acknowledgements

This project has been successful because of the contributions of many people. I would like to thank our customers past and present: John Weinstein, Ajay, Kim Bussey and Barry Zeeberg. I would also like to acknowledge the software engineers who have been on the project Hong Cao, Steven Day, Sudar Narasimhan, Margot Sunshine and Jon Whitmore. I would also like to thank Tim Ruppert, John Weinstein, Kim Bussey, and this paper's shepherd, Rebecca Wirfs-Brock, for their feedback on this paper.

11 References

[1] National Cancer Institute's Laboratory of Molecular Pharmacology's Genomics and Bioinformatics Group (Web site: http://discover.nci.nih.gov).

[2] JN Weinstein, KW Kohn, MR Grever, VN Viswanadhan, LV Rubinstein, AP Monks, DA Scudiero, L Welch, AD Koutsoukos, AJ Chiausa, KD Paull, K. D. Neural computing in cancer drug development: Predicting mechanism of action. *Science* 1992; 258: 447-451.

[3] JN Weinstein, TG Myers, PM O'Connor, SH Friend, AJ Fornace, KW Kohn, T Fojo, SE Bates, LV Rubinstein, NL Anderson, JK Buolamwini, WW van Osdol, AP Monks, DA Scudiero, EA Sausville, DW Zaharevitz,

B Bunow, VN Viswanadhan, GS Johnson, RE Wittes, and KD Paull, An information-intensive approach to the molecular pharmacology of cancer. *Science* 1997; 275:343-349.

[4] L Tanabe, U Scherf, LH Smith, JK Lee, L Hunter and JN Weinstein, MedMiner: an Internet Text-Mining Tool for Biomedical Information, with Application to Gene Expression Profiling, *BioTechniques* December 1999 27:1210-1217.

[5] KJ Bussey, DW Kane, M Sunshine, S Narasimhan, S Nishizuka, WC Reinhold, BR Zeeberg, Ajay and JN Weinstein, MatchMiner: a tool for batch navigation among gene and gene product identifiers, *Genome Biology*, April 2003 4(4):R27.

[6] BR Zeeberg, W Feng, G Wang, MD Wang, AT Fojo, M Sunshine, S Narasimhan, DW Kane, WC Reinhold, S Lababidi, KJ Bussey, J Riss, JC Barrett, and JN Weinstein. GoMiner: A Resource for Biological Interpretation of Genomic and Proteomic Data. Genome Biology, April 2003 4(4):R28.

[7] SRA International (Web site: http://www.sra.com).

[8] PM Harrison, A Kumar, N Lang, M Snyder and M Gerstein, A question of size: the eukaryotic proteome and the problems in defining it, *Nucleic Acids Research*, 2002, 30(5):1083-1090.

[9] Spotfire Decisionsite (Web site: http://www.spotfire.com/products/decision.asp).

[10] Scalable Vector Graphics (SVG) 1.0 Specification, W3C Recommendation 04 September 2001, http://www.w3.org/TR/SVG.

[11] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[12] K. Schwaber, M. Beedle, *Agile Software Development with Scrum.* Prentice-Hall, 2002.

[13] RC Martin, *Agile Software Development, Principles, Patterns and Practices*. Prentice-Hall, 2002.

[14] Concurrent Version System (Web site: http://www.cvshome.org).

[15] R Cabrera, B Appleton, SP Berczuk, Software Reconstruction: Patterns for Reproducing Software Builds, *Pattern Languages of Programming'99*, (Web site: http://jerry.cs.uiuc.edu/~plop/plop99/proceedings)

[16] SC Berczuk, B Appleton, *Software Configuration Management Patterns*, Addison-Wesley, 2002.

[17] Ant (Web site: http://ant.apache.org).

[18] Junit (Web site: http://www.junit.org).

[19] MaxQ (Web site: http://maxq.tigris.org).

[20] WF Opdyke, Refactoring Object-Oriented Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992. Available as Technical Report No. UIUCDCS-R-92-1759.

[21] M Fowler, *Refactoring*, Addison-Wesley, 1999.

[22] Intellij IDEA (Web site: http://www.intellij.com/idea).

[23] D Dikel, D Kane, J Wilson, *Software Architecture: Organizational Principles and Patterns*, Prentice-Hall, 2000.

[24] T. Gilb, D. Graham, Software Inspection. Reading, Massachusetts: Addison-Wesley, 1993.

[25] Dbunit (Web site: http://dbunit.sourceforge.net).

[26] JFCUnit (Web site: http://jfcunit.sourceforge.net).