Going On-line on a Shoestring: An Experiment in Concurrent Development of Requirements and Architecture

Charles B. Haley, Bashar Nuseibeh

***Abstract*—A number of on-line applications were built for a small university using a micro-sized development team. Four ideas were tested during the project: the Twin Peaks development model, using fully functional prototypes in the requirements elicitation process, some core practices of Extreme Programming, and the use of open-source software in a production environment. Certain project management techniques and their application to a micro-sized development effort were also explored. These ideas and techniques proved effective in developing many significant Internet and networked applications in a short time and at very low cost.**

***Index Terms*—product champions, project management, software requirements, software design and development.**

I. Introduction

THE American University of Paris (AUP) is a small American-style liberal arts institution located in Paris France, with approximately 900 students and 115 full- and part-time faculty. Before the year 2000, AUP had not 'gone on-line' other than with a static web site. There was no ubiquitous e-mail system. Faculty and students could not store electronic information, create instructional and personal web sites, or experiment with networked applications. There was no on-line access to e-mail directories or course catalog and schedule data. In 2000, AUP concluded that its 'low-tech' state complicated its ability to attract students and faculty.

C. B. Haley joined AUP in 1999 as a part-time professor in the Computer Science department. In addition to wanting to improve the technical infrastructure to aid teaching his courses, he was interested in extending the requirements elicitation techniques described in [7] by incorporating fully operational prototypes, investigating using the tenets of Extreme Programming (XP) [1] to build the prototypes, and exploring the effectiveness of micro-sized teams. Combining these research interests with the need for a better technical infrastructure for teaching, he proposed constructing several Internet-enabled systems for AUP. The proposed systems were to be based on free and open-source software and connected with existing AUP administrative systems to the maximum extent practical, with the twin goals of a) minimizing cost and duplication of data, and b) maximizing value to the university community. The institution accepted the proposal and funded the project at a level of approximately 15,000 € per year. The project started in the summer of 2000 and ran for 2 years.

The remainder of this paper is laid out as follows. Section II provides some background material and the project goals. Section III sets the stage by briefly presenting what was built during the project. Section IV describes how construction of each system progressed, with particular attention to how the principles of Twin Peaks were used. Section V revisits the research questions, and section VI presents conclusions.

C. B. Haley is a part-time 1) associate professor in the Department of Computer Science, The American University of Paris, 75020 Paris France; 2) professor of MIS, European School of Economics, 75116 Paris France 3) PhD student, Computing Department, The Open University, Milton Keynes MK7 6AA, UK; and 4) consultant. (e-mail: charles.haley [at] the-haleys.com).

B. Nuseibeh is a professor of computing at The Open University. (e-mail: B.A.Nuseibeh [at] open.ac.uk).

II. Background and Goals

A. Twin Peaks

Although the description of the Twin Peaks model [5] [6] was published after the project began, the model so completely captured the project's spirit that there is significant advantage in using the model and its terminology to describe the project. The model proposes a partial development methodology wherein *requirements* and *architecture* (where architecture includes *implementation*) are simultaneously elaborated and verified against each other, bound together by the *specification process*. The model extends the spiral method [2] by making elaboration of requirements an explicit part of the spiral. The benefits derived from the model include earlier understanding of the problem(s) being solved, rapid turn-around, and inherent recognition and incorporation of project management concerns such as IKIWISI (I'll Know It When I See It), easier incorporation of reusable components such as COTS (Commercial Off-The-Shelf) products, and rapid change in requirements and technology [3].

Figure 1 illustrates how a project might move from idea to implementation while using Twin Peaks. The peaks represent the requirements and architecture artifacts. The further one moves down a peak, the more detail is present and the more complete the artifact is. The spiral line represents the specification process, which is itself not an artifact but the

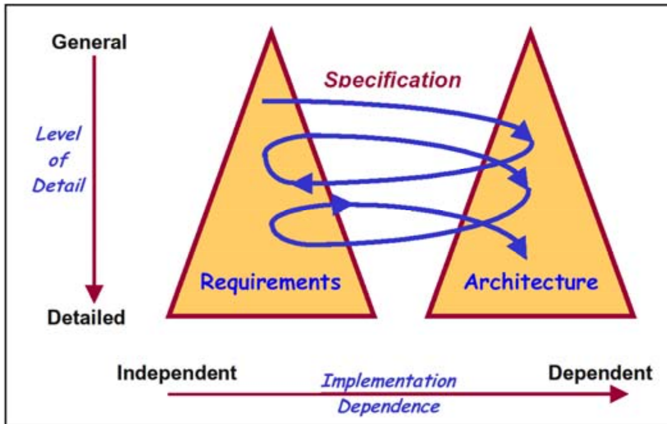

Figure 1: Twin Peaks – A model of concurrent development of requirements and architecture

simultaneous application of various and distinct methods to elaborate requirements and implementation.

Twin Peaks does not impose a requirements engineering method, a software architecture style, or a design method on a project. Members of the project choose appropriate methods based on their experience and knowledge, and on project goals, size, and complexity.

B. *Extreme Programming (XP)*

XP [1] argues that using 12 *core practices* together facilitates releasing software that works, is on time, and meets the customer's needs. Although applying all 12 XP practices requires a programming team larger than one person, it did seem reasonable to test six of them in the context of this research. The six practices chosen are *Planning Game* (release and iteration planning), *Onsite Customer* (a real end-user is on the team), *Continuous Integration* (changes are integrated on at least a daily basis), *Small Releases* (release small increments early and often), *Simple Design* (engineer for today, not for tomorrow), and *Refactoring* (improve the design of existing code).

One purpose of a Twin Peaks spiral is to verify the architecture and to obtain feedback on the requirements. Four of the six XP practices should support this verification: iteration planning, small releases, continuous integration, and simple design should help to complete a cycle quickly. The size of the development team necessitated having end-user product champions in the project (the Onsite Customer). Design improvement should be equally natural when revisiting a prototype.

C. *The Goal of the Project*

The project's goal was to make progress on the following research questions, posed both initially and during the effort:

1. Twin Peaks requires that requirements elaboration and implementation proceed in parallel. Is this practical in the context of severely constrained resources? Which techniques work best in this situation?
2. Both XP and Twin Peaks admit that an implementation might not be acceptable. In addition, a fully functional prototype will develop its own user community. Would users tolerate the levels of change that correction cycles entail? How does one minimize the problem?
3. One could argue that this project is an example of "heroic efforts" as described in [4]. Alternatively, to survive over time the project should be an example of the "sustainable pace" principle of XP. What would happen?
4. To be successful, existing staff must be able to administer the systems. How would this constraint affect the development and rollout of the systems?
5. A significant fraction of AUP's staff and faculty mistrust "new" technology. Insertion of new technology is often seen as 'fixing something that isn't broken'. Users feel that they do not have time to learn about or participate in building a new system. Does aggressive prototyping help in this case, or make the problem worse?

III. The Systems

It is convenient to start at the end of the story to avoid too many forward references, to present what was built, and to list the COTS[1] products used during construction of the systems. Descriptions are very brief; no attempt is made to provide details about what the systems do or how they do it.

The overarching requirement was that the applications be written for the Internet (web-enabled) using a 'pure browser' model. The requirement applied to both the query and maintenance applications. One reason was to permit people outside of AUP to see the information (marketing, if you will) while avoiding installing software of any kind on any client. A second reason was to permit system maintenance from anywhere in the world. Finally, it would be nice to have clean examples of web applications for use in courses.

There were 8 systems implemented during this project:

A. *Basic Support*

Every member of the community (student, faculty, staff, trustee, alumnus, etc.) has network storage and a personal website. Read/write access to the storage and write access to the user's website is secured by userID/password. COTS used: Linux, apache, samba, MySQL, and netatalk, perl, PHP, squid.

B. *Student E-Mail*

Every member of the community has an e-mail account. The person can choose to have e-mail forwarded or to read it locally. Local reading is done using a web and/or standard POP3/IMAP clients. COTS used: sendmail and delivery daemons, MailMan by Endymion, and RAV e-mail antivirus[2].

C. *Course Catalog*

The AUP course catalog, which is the list of all courses AUP offers regardless of the semester, is available on the web and in print. The course listings in the both versions are generated from a unified course database.

[1] Even though the products would be better described as OSOTS (Open Source Off-The-Shelf software), the term COTS will be used in this paper.

[2] This is the only commercial (for fee) software product the project used.

D. Course Schedules

Semester course schedules (course, time, place, professor, etc.) are available on the web, updated daily. The schedule is generated by combining schedule, professor, and enrollment data from the Registrar's database with the information in the course catalog database. The schedule system, implemented within the course catalog application described above, is a reasonably large web application consisting of approximately 50 web pages and 20,000 lines of code in 60 objects.

E. Course Registration and Advising Support

Academic advisors have on-line access to their advisees' course and transfer credit records. The student and the advisor can together create a schedule for a semester; the courses on the schedule are checked to see if prerequisites are satisfied and that there are no schedule conflicts. The result is printed and signed, and then carried by the student to registration.

F. Systems Administration

Web-based tools were built to create, delete, modify, and suspend accounts. The system automatically verifies that an account holder is receiving e-mail by periodically sending an expiration notice. The account is suspended if the account holder does not reply to the notice.

G. University-wide E-mail directory

Some faculty and staff use Lotus notes as their e-mail system. A web-accessible directory was built that could query both the Notes and the system built for this project.

H. Wireless and Virtual Private Network Access

Secured wireless and VPN access to the university's network has been implemented and tested, but is not yet generally available for use.

IV. The Experience

The Twin Peaks model accurately describes the interplay of methods used to implement the systems built during this project. Requirements were elaborated based on user feedback and the constraints imposed by the COTS software and then tested using fully functional prototypes. Depending on the system there were from 2 to 5 iterations of the spiral. Varying requirements elicitation and project management approaches were used for the following reasons: failure of the initial attempt, a desire to test an alternate method, a lack of resources, or a resource became available. The prototypes and 'finished' applications were built using web-application design and implementation techniques such as embedded scripting, page generation, and objects with templates.

Some details for each system are presented below.

A. Basic Support

The first version of the requirements came from the experience of the CS faculty, the supervisor of the student-accessible computer lab, and a student charged with building a website for one of the academic departments. It was a completely functional prototype lacking only security, and was put into 'production' for use by CS and other technologically well informed students.

Version two of the system completely replaced version one. It was built in stages, using an order determined from experience with the first version. Each component of the first version was discussed with representative stakeholders, reconstructed, and then verified with the stakeholders. The number and diversity of stakeholders grew with each stage. Several components, such as systems administration and user account expiration went through several revision cycles.

The project was confronted with IKIWISI from the start. Stakeholders were not confident that their descriptions of their needs accurately reflected reality. It was only after using the system that they were able to be precise. This was especially true for any part of the system that incorporated a user interface, such as system and user file management.

B. Student E-Mail

This project started by using interviews with students as the requirements elicitation technique. The technique did not work, primarily due to the nature of the users. Students are by nature transient. Long-term means next weekend. Their coursework presents sufficient difficulties and they do not wish to be further challenged outside of class. Ideas and prototypes were ignored during the interviews. The students said "do the same as hotmail" or yahoo or caramail or one of around 25 alternate public web e-mail systems. The interviews gave no useful input beyond 'make it simple', so a benevolent dictator approach was imposed to choose a free (but not open source) COTS web-based e-mail product that is simple, works reliably, and is maintained. There have been few complaints.

When first asked, AUP's administration was not interested in the student e-mail system, with the exception of people in the Office of Student Affairs who wanted to be able to send e-mail to all current students. This disinterest lasted until the first version was deployed, at which point there was a deluge of requests to build and maintain e-mail aliases for purposes ranging from students visiting from a particular university, all graduating seniors, and members of the drama club. Continuing administrative costs vetoed maintaining these aliases by hand, so a way was found to use the Registrar's database to identify the members of most of the requested groups. An end-user-administered mailing group system was built so the clubs and organizations could maintain their own mailing lists.

Experience building this system leads one to postulate a corollary to IKIWISI, specifically IKIWIDSI (I'll Know It When I Don't See It). Even though the administrative users were not initially able or willing to describe what they wanted, they had no problem once they had a system that did not satisfy their needs. In other words, in one step they went beyond the notion of "that isn't it, try again" to "what I really need is X, which isn't what you did."

C. *Course Catalog & Course Schedule*

These two systems are clearly the most problematic on several dimensions: requirements shift, rapidity of change, and complexity of development. Four distinct versions were built; three of them were tried in production.

When the project started, AUP had at least three groups maintaining versions of course catalog information in 'production.' The first group maintained the paper catalog, the second AUP's website, and the third the Registrar's data. One goal was to combine these versions into a single definitive database. Centralizing the data and the processing would dramatically impact how the three groups operate, making finding a "product champion" from one or more of the three groups imperative. Unfortunately, in the beginning none of the stakeholders saw enough benefit in combining the information to participate in the project, so the first system was built using a "best guess" method. IKIWISI (and IKIWIDSI) worked again, and the university Webmaster became the first product champion. She took it upon herself to interview students and administrative staff, and to sell the idea to the Registrar by showing them something that they could use. The second version was designed using her results.

The second version was very successful from the point of view of the users, but much less so from the point of view of the people who were responsible for maintaining the data. Maintenance consisted of changing several small files and running an "import" process; a process that was error prone and complex. Fortunately, at this point the project picked up a product champion in the Registrar's office. She worked out how to obtain the needed information from the Registrar's course schedule database, eliminating 95% of the complexity of data entry. Version 3 was born.

The last version came into existence when the supervisor in charge of the printed catalog was convinced to bring the production of the document in-house and to generate the course listings in the printed catalog from the database. She agreed to participate under the condition that she be supported by the developer[3] on an on-going operational basis throughout the initial project. This was the only time the technique of reporting directly to the user of the technology was used, and the technique worked very well in this case. The supervisor became the third champion.

This system would not have been successful without the Twin Peaks spirals and without aggressive prototyping. The project initially faced indifference and occasional hostility, both of which had to be overcome to succeed. The product champions were nurtured by giving them something to work and play with, and by encouraging their inventiveness by implementing their ideas. Reticence was overcome by working directly with users, isolating them from 'blame' if the project did not go well. Hostility disappeared as the champions became more enthusiastic.

[3] To help identify who is doing what, the role of 'developer' is introduced. C. B. Haley was the only developer.

Shifting requirements remains a difficult problem. The institution is continuously modifying its curriculum and changing how courses are related. For example, in Fall 2002 the notion of 'paired courses' was developed, where a student must take both courses simultaneously or take neither. As of this writing, information systems maintenance concerns are not taken into account when deciding to make changes, usually leading to 'after the fact' scrambling.

D. *Course Registration and Advising Support*

A more classical approach was used to build this subsystem. Two CS students asked if they could build a registration and advising support system as their senior project. An agreement was struck stipulating that the students start with the existing catalog and schedule system. In effect, it became a COTS product for their project. Starting there, the students identified the stakeholders (students, advisors, the Registrar, and the Academic Dean), conducted interviews, prototyped the results, and repeated the process. In the end, there were two spirals through the requirements and architecture peaks.

The classical approach worked well here because what was being built was an add-on to the existing system. In effect, the prototypes were already built and in production. Imagining the extensions was not an overly large step.

E. *Systems Administration*

The stakeholders for systems administration were easy to identify and easy to work with. In particular, the supervisor of the student computer lab was and is an enthusiastic champion of the project. However, he and his staff were unable to describe what they needed.

We solved the problem by using two different methods during the spirals. The first was the approach described above: aggressive prototyping and IKIWISI. The second approach corresponds roughly to the XP "simple design" practice; anticipate very little but respond rapidly when the need arises. When a problem was discovered it was first resolved by brute force. Immediately thereafter, the developer worked with the stakeholders to implement a means for them to resolve the problem themselves. This technique worked well, but was difficult to live with because for a time the developer was on call seven days/week. Fortunately, the problems were minor and discovered within weeks of deployment.

F. *University-wide E-mail directory, and Wireless and Virtual Private Network Access*

It was clear from the beginning what these two systems were to do. Both efforts were dominated by technical and security concerns, and almost no end-user input was required.

A major benefit of this subproject was to bring the IT staff into the overall effort. As they control both the network and the Notes e-mail system, their participation was desirable and necessary. They are now active project supporters.

The e-mail directory is in production, but the VPN and wireless system is not. VPN and wireless access rollout awaits sufficient budget to cover maintenance and user support.

V. The Lessons

One surprising lesson learned was that the severe limit on development resources was not a deciding factor in any of the projects. The effort was sustainable, and in fact enjoyable. There was never a situation where the time required to accomplish something destroyed the utility of doing it. Until the project finished in spring 2002, all of the expressed needs of the stakeholders were satisfied. The nature of the systems (many small pieces), the extensive use of COTS components, the willingness of the champions to find expedient solutions, and the natural flexibility of a one-person programming group all played a part in this happy state of affairs.

Although 'heroic efforts' was not a direct problem, the idea reappeared and contributed indirectly to a major project failure. Ending the research project has proved problematic for AUP for two reasons. The first is that the project produced useful systems with no apparent effort, giving the impression that continued development would be equally as easy and inexpensive. The second is that because of the apparent ease, the administration did not realize that the maintenance costs would exceed the construction costs. The projected cost of maintenance was a shock for AUP's administration, resulting in the indefinite postponement of rollout of the VPN and wireless services. It is a clear failure of the project to have built systems that AUP has come to depend on, while inadequately assisting the University to prepare for the systems' continued maintenance and development.

Initially, the user community was not overly disturbed by the changes in the systems that came with reimplementation of the initial versions. There were a few cases where some champion was so attached to a prototype ("her" prototype!) that she resisted a change, but these events were rare and easily overcome. However, as time went on the resistance grew rapidly. The hypothesis, supported by interviews, is that the early adopters were interested both in the improvements offered by the systems and in being a part of the process, whereas the late adopters were interested in the results but had no interest in the process. In addition, the apparent stability of the system had begun to convert the 'anti-tech' members of the community into users. Interviews with people who put themselves into the anti-tech group have made it clear that they began to use the systems only because they saw colleagues deriving benefits and because they felt confident that what they learned to use would remain stable.

The project would have failed if the systems administration problems had not been addressed as they were. On one hand, an attempt could have been made to anticipate all of the tools that the computer lab staff would need and build them; the project would never have finished. On the other hand, the staff members could have been given the system and then left to their own devices; the system would have been shut off a week after deployment. The compromise of first solving problems by hand and then building the needed tools worked well, ensuring that tools were known to be useful before they were built and spreading out the workload over many weeks.

The micro-development effort helped in three ways. The first and most obvious is the lack of bureaucracy; there were no reasons for members of the project to put procedural barriers in front of themselves. The second was the ability to form and disband small user-based teams as needed while maintaining project consistency. The third was the ease in changing project management styles. The major negatives were having only one pair of eyes on the problem and personality clash. The first was worked around by holding 'design reviews' with students taking web-related courses. The second was more problematic; there was no fallback if the developer did not work well with some stakeholder. AUP's Webmaster becoming a product champion obviated the problem, because she became a second channel between the project and the users.

Five of the six principles of XP that were tested with the project worked well. The Twin Peaks spirals were more effective and completed more quickly because of the focus on simplicity and small releases. Continuous integration and iteration planning helped with verifying and testing the systems because "new" features were always available to the champions. The product champions (the "Onsite Customers") were essential to the project. However, design improvement in the XP sense did not yield the expected benefits, primarily because the prototypes were rewritten from the ground up. In XP, design improvement is defined as refactoring, or improving design without modifying behavior, with the result that rewrites for new functionality do not generally qualify as design improvement even if basic improvements were made during the process.

VI. Conclusions

Twin Peaks works. It works even better when used with completely functional prototypes. Prototypes have a risk, though. One must really be willing to abandon completely a given prototype, or many of the benefits will be lost because keeping a prototype can pollute the process. The catalog and schedule system required three complete and distinct implementations before the requirements were well understood, then a fourth to build the version running today.

The use of fully functional prototypes as part of the spiral works. Use of prototypes can suffer from the criticism found in [7], that users sometimes accept prototypes without criticism. Users can also become attached to what they are offered. These problems were manageable in the context of this project. A major positive argument is that users can freely experiment with working systems whenever they want to as opposed to during a meeting. The two people who became the most influential product champions began this way, playing with the system as they confronted particular problems in the course of their work. They wanted to see if the new system could help them.

As noted in section V, the largest failure of the project was the failure to adequately involve the upper levels of the University's administration in the process. One could

conclude that the University was not well served by producing these systems at a bargain-basement price as opposed to a more 'normal' cost[4], and by failing to make explicit the continuing development and maintenance costs.

Although building multiple implementations could be considered a waste of resources, the effort was in fact efficient.[5] The cost is clearly reasonable. Interviews have indicated that satisfaction with the result is high. The quality of the feedback from the early prototypes was superb, both at the conceptual and at the detail level. Vocabulary problems were avoided by 'pointing' at examples. It was easier to discuss cost/benefit because we had a very accurate estimate of the cost. People were converted from being actively hostile to being active supporters because they could directly participate and rapidly see the results of 'their' suggestions.

Applying the five principles of XP helped in two significant ways: rapid turnaround of the prototypes and empowering the product champions. Focusing on results and releases helped ensure that the needed functionality was built without delaying the project to add extraneous features. All three of the champions felt that they had real control over the project's direction. They could see the effects of their suggestions, sometimes within hours. The priorities the champions set were respected. Two champions began to think of the system as 'theirs'. XP as a whole was not tested, but using these five as an ensemble can be recommended.

As noted in section V, the project did not test the sixth XP practice, design improvement, that was originally on the list. Not testing the practice should not be taken as a statement that the practice does not work.

The micro-team approach worked well on this project. Having the flexibility to change approaches, involve others in the project, and negotiate pathways through the obstacles made a large difference both in the degree that the results met the users' needs and in how quickly the project could progress.

[4] The director of AUP's IT department estimated that replacing the applications built during this project would require several experienced engineers, costing approximately 10 times more than spent on this project.

[5] Unfortunately, accurate records of time spent developing, in interviews, and in 'support' of the individual systems were not kept. As such, these arguments are somewhat anecdotal.

References

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

[2] B. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, vol. 21, no. 5, May, pp. 61-72, 1988.

[3] B. Boehm, "Requirements That Handle IKIWISI, COTS, and Rapid Change," *IEEE Computer*, vol. 33, no. 7, Jul, pp. 99-102, 2000.

[4] D. Jung-Gribble, "Heroic Effort is not a Sustainable Model," in *Proceedings of the 25th SIGUCCS conference on User services*. Monterey, CA USA: ACM Press, 1997, pp. 67-72.

[5] B. Nuseibeh, "Weaving the Software Development Process Between Requirements and Architecture," in *From Software Requirements to Architectures (STRAW '01)*. 23rd International Conference on Software Engineering, ICSE 2001. Toronto, Ontario, Canada, 12-19 May, 2001.

[6] B. Nuseibeh, "Weaving Together Requirements and Architectures," *IEEE Computer*, vol. 24, no. 3, March, pp. 115-119, 2001.

[7] A. Sutcliffe, "A Technique Combination Approach to Requirements Engineering," in *Proceedings of the Third IEEE International Symposium on Requirements Engineering (RE '97)*. Annapolis, MD, USA, 6-10 Jan 1997, pp. 65-74.