

Experience Report on the Development of a Network Management Application in a Small Mexican IT Firm

Humberto Cervantes

*Universidad Autónoma Metropolitana-Iztapalapa (UAM-I),
San Rafael Atlixco N° 186, Col. Vicentina, C.P. 09340 Del. Iztapalapa. Distrito Federal, Mexico
hcm@xanum.uam.mx*

Abstract

This paper presents an experience report on the development of a network management application in the context of a small IT Mexican firm (PyME). In this firm, whose main focus is not on software development, a development team was created specifically for the project. Although the environment of the project was not particularly “software-engineering friendly”, the development team made an effort to introduce as many software engineering practices as possible with respect to the available and very limited resources to pursue quality. This report presents aspects related to the organization of the development team and a few experiences related to the development of the architecture of the system in this particular context. The experience gathered during the development of this project may be useful for development teams that have to work in similar environments. This report may also be useful for small IT owners who wish to venture into software development.

1. Introduction

Small IT firms (PyMES) are common in Mexico and many of them strive to survive in a difficult economic environment. Furthermore, many of these firms lack a clear focus on the type of projects they develop and as a consequence they venture on a variety of projects. On occasions, these small businesses initiate software development projects even if this is not their main area of expertise. Frequently, these projects are realized in an ad-hoc fashion without an established software development process. The end result is software which, if delivered at all, often lacks in quality.

This paper presents an experience report of the development of a network management application in the context of a small IT firm that had no previous particular focus on software development. The small firm hired the author, who specializes in software engineering, to participate mainly as a project manager and to a certain degree as a software architect. The author, who teaches at a public university in Mexico City, was responsible of

recruiting a development team to participate in the project. The team was composed of 3 developers which had been students of the author. Previous to this project, the development team and the author had only one experience working together in a development project inside the university. Although the previous project was not totally successful, it served to establish the basis of several of the practices used in the development project described in this paper. The main goal of the development team was to establish the minimum of required practices to create a quality product at the lowest cost.

The system that was developed as a result of this project is a network management application. It is today in its third release and currently implements around 40 relatively complex use cases and measures around 42 KLOCs. One of the main challenges of this project has been the development of the architecture of the application.

This article focuses on two main areas of this development experience and their relation to the system quality: the organization of the development team and the lessons learned in the development of the architecture of the system. The goal of this article is to give an example on how small teams can set up best practices to improve the quality of the software they produce even if they don't have many available financial resources. This article may also be useful for people who own small IT organizations and who desire to venture on software development projects so that they can better understand what to provide to their development teams to facilitate their tasks.

The structure of the article is the following: Section 2 briefly presents the context of the project. Section 3 describes the development organization with a focus on the development methodology and tools. Section 4 presents some experiences related to the development of the software architecture of the system. Section 5 evaluates the experience and describes the lessons learned through this project. Section 6 presents some related work and finally section 7 concludes the paper.

2. About the application

Although it is not possible to describe too many details about the application that was developed, it is worth noting that the application is a (heavy-) client-server multi-user network management application which provides FCAPS functionalities. FCAPS is a standard model which defines 5 areas of network management which include Faults, Configuration, Accounting, Performance and Security [1]. Fault management includes displaying a network representation and highlighting problematic network elements. Configuration management includes remote configuration of network elements. Accounting includes the gathering of information from the network elements for inventory purposes. Performance includes providing data to the user for analysis purposes (such as data graphs). Finally, security management includes authentication and authorization support for different types of users. One problematic (and risky) area particular to this application has been that the protocol used to interact with the network elements that are managed by the application is not thoroughly documented by the network element manufacturer.

Although the development team was part of a small IT firm, this firm collaborated with a secondary IT firm whose focus was on selling the hardware to be managed by the application discussed in this paper. The end customer is a big IT company with very bureaucratic procedures. As a result, this project involved stakeholders from at least three different organizations.

Furthermore, it must be mentioned that the development of the application started in June 2006. In the 24 month period of development (at the time of this writing) there have been 3 different releases. The first release had a very tight schedule as it had to be delivered by the end of August 2006. It is important to mention that the completion of the first release was a fundamental step to receive project approval and further funding. The second release was originally planned for the end of 2006 but was re-scheduled to the end of August 2007, after the stakeholders were convinced of the importance of developing a good software architecture (this is discussed in section 4). The development of the third release is currently terminated and is undergoing acceptance testing with the final customer.

3. Development team organization

This section presents the organization of the development team including the adaptation of the development process and the detail of the development practices associated to the process.

3.1. Process adaptation

The development of the project has been based on lightweight instance of the Rational Unified Process (RUP). The Rational Unified Process is a popular iterative software development process framework which divides a development project in 4 sequential phases called Inception, Elaboration, Construction and Transition [2]. Inside each of these phases, a series of iterations are carried out and, on each one of these iterations, activities related to specific disciplines such as requirements or testing are performed with a varying degree of effort (see figure 1). Since it is a process framework, the RUP must be adapted for a particular project. In the context of this project, each of the three releases of the system has been managed as a separate RUP instance. However, due to the tight schedule (particularly of the first release), it has not always been possible to carry out all the necessary activities corresponding to the different phases in the depth mandated by the unified process.

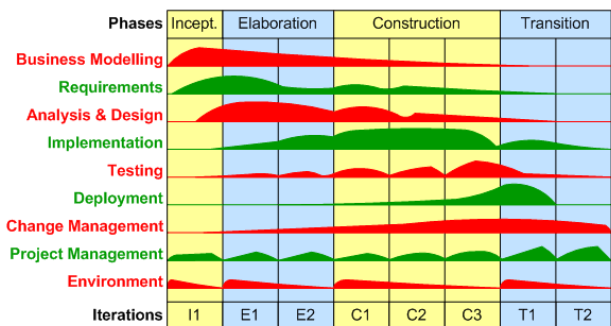


Figure 1: The 4 phases and 9 disciplines of the Rational Unified Process.

The main constraints that drove the adaptation of the unified process framework were the small size of the team (3 developers and one project manager), the low availability of financial resources, and the fact that the project manager was not available in full time as his primary work is in a university. As a result, the adaptation of the RUP framework had to be driven by these constraints.

The practices that are promoted by the RUP and that were chosen for the development process were mostly limited to the ones associated to the level 2 of the Capability Maturity Model (CMM) [3] and which include:

- Requirements management
- Project planning
- Project control and monitoring
- Quality assurance
- Configuration Management

The way these practices were covered and the tools that were used to support them are discussed in the following sections.

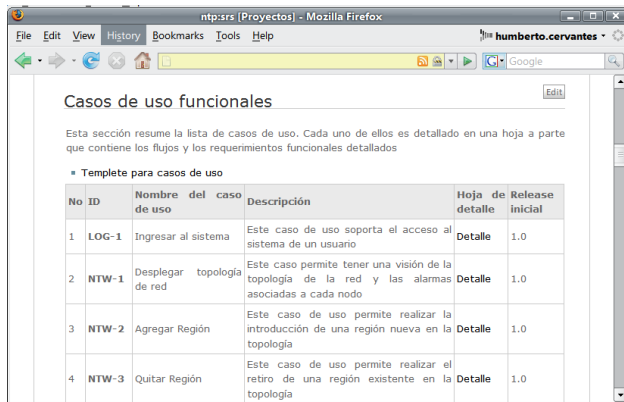


Figure 2: Screen capture of a web browser showing a portion of the SRS page

3.1.1. Requirements management. This practice includes requirements change control, version control, status tracking and tracing [4]. To facilitate these activities, the development team relied strongly on open-source tool support. The main tools that were used for this discipline include a wiki (Dokuwiki¹) and a simple web-based tool to support bug tracking (CodeTrack²). A Software Requirement Specification (SRS) page was created inside the wiki as a starting point to document requirements (see figure 2). This page references other pages that detail the specific use-cases. Although a wiki is not a dedicated requirements management tool, it has proved particularly helpful for version control activities as the wiki keeps track of document modifications. Furthermore the wiki is easily accessible to all stakeholders (not all have write access and all are on different locations) and it is simple to use and cheap to install and maintain. One aspect that must be taken into account early in the establishment of the wiki is the naming scheme related to the version identification. The development team concluded after a certain time that it is useful to create a new name for requirements pertaining to different software releases instead of simply relying on the wiki revision system. This is because it is often necessary to quickly access requirement documents from previous system releases and it is sometimes necessary to change them (this is not feasible on wiki revisions). Finally, requirement tracing between documents is facilitated by the creation of hyperlinks in the wiki. One limitation that has been identified of using a wiki for requirements management is that it is difficult to modify UML diagrams that are uploaded along with the text.

3.1.2. Project planning. Project planning has been a relatively difficult activity because of the problems associated to the estimation of the effort required to

perform the different project tasks. This was mainly originated by the lack of data from previous development experiences. To overcome this limitation, informal Wideband Delphi estimation sessions have been attempted among the development team to provide estimations for release delivery dates. Unfortunately, the results have not been very accurate because, on several occasions, the development team was not provided with all the necessary information to identify a majority of the tasks necessary to realize the project. Furthermore, the end customer for the application demands complex test procedures to be performed to accept the product. The realization of these procedures must be requested with a relatively long time in advance and it is always very difficult to have a precise idea when the end customer will finally provide a precise date for acceptance testing. As a consequence, the project calendar is based on certain dates when the test procedures are requested but between the procedure request and the actual test procedure execution, a delay of one or two months may occur.

During the development of the first release of the project, traditional project management techniques were applied to establish the project calendar (Gantt diagrams, etc...). These techniques were supported by another open-source tool dedicated to project management (OpenWorkbench³). In the first release, the traditional project management approach was appropriate because the delivery deadline was immutable, and the amount of functionalities to be delivered was well defined and not too high (the stakeholders understood clearly that they couldn't request additional features or the project could be put at risk). Furthermore, during this first release, the tight schedule limited very much the amount of work that could be performed on the development of the architecture (see section 4 for more details on this).

In the subsequent releases, the team decided to apply a more agile approach based on very short (one week) iterations. Instead of focusing mainly the global schedule (related to testing dates), the focus shifted towards the iterations. Iterations are performed in the following way: at the end of every week, the development team and the project leader gather together and evaluate several aspects, including the status of completion of the objectives that were set on the previous meeting. The objectives for an iteration are usually related to the current phase in the unified process. For example, in the construction phase (where the system is actually built) they are derived from the list of use cases referenced by the SRS page mentioned previously. During testing activities, the objectives for an iteration are typically obtained by revising the bug tracking tool in search of open issues which are prioritized according to perceived risk. Once the main iteration objectives are identified, they are decomposed into fine-grained tasks (this is the

1 <http://wiki.splitbrain.org/wiki:dokuwiki>

2 <http://kennwhite.sourceforge.net/codetrack/>

3 <http://www.openworkbench.org>

standard Work Breakdown Structure technique). Each of the tasks is then assigned to one of the team participants. Resource allocation is typically achieved through consensus (the same goes for the iteration scope). Once the tasks are defined, the members of the development team choose which tasks they wish to address. Iteration plans are documented inside the wiki using a simple template which contains a table that lists the main iteration objectives and a decomposition of these objectives into a list of tasks that are assigned to the individual developers (see table 1).

id	Description	Owner	Estimated time	Actual Time	Status

Table 1: Fields in the iteration plan

3.1.3 Project control and monitoring. These activities are also supported through the use of the previously mentioned wiki tool. During the execution of a particular iteration, the status of individual tasks inside the iteration plan is continuously updated as the developers complete the particular tasks. One additional activity that is supported by the wiki template developed for the project is the collection of metrics to improve future estimation efforts. The metric that is collected is the effort (in man hours) that is required to perform a particular task. At the beginning of the work on a task, the task owners must provide an estimation of the time they consider will be required to perform the particular task. Once they finish the task and when they update the status of the task in the template, they register the actual time required by the task. Although the collection of data has not been carried out in a very accurate fashion up to now, the approach presented here has been well accepted by the development team as data collection does not require a significant effort. At this point, data is still being collected and is awaiting analysis.

3.1.4 Quality assurance (QA). QA refers to the activities that are undertaken to ensure that the software satisfies the expectations of the customer in terms of the requirements. The Software Engineering Body of Knowledge (SWEBOK) specifies that the techniques related to QA activities can be categorized in two different areas: static techniques and dynamic techniques [5]. The former include the definition of standards and processes to assess the different artifacts produced during the life of the project and the latter include testing techniques (since testing requires code execution this is why these techniques are categorized as being dynamic).

In this particular project, the actions that have been undertaken to support quality assurance include: setting of standards and reviews and setting of different testing techniques. As with the previous practices, quality assurance is also supported by different open-source tools.

Regarding static techniques, the wiki has proved to be once more a useful tool in their support. The project wiki contains a section where standards are documented. As the intention of the team is to maintain a relatively agile spirit in the project, only essential standards are documented. These include: coding standards (including exception management and comments) and user interface standards. With respect to reviews, informal peer technical reviews are carried out by the developers when completing difficult changes, and periodical technical reviews are carried out during the weekly meetings with the project leader.

In the area of dynamic techniques (i.e. testing), the team has focused mainly on unit and system level tests. The tools associated with the testing effort include Junit⁴ to automate unit tests and Profiler4J⁵ to evaluate memory consumption. In addition to these tools, many specific tools have been developed for different system-level tests (including stress testing). Finally, it must be noted that test documentation is stored inside the wiki pages.

3.1.5 Configuration Management (CM). CM focuses on the identification of the configuration of a system for the purpose of controlling changes in a systematic fashion. In the context of this project, configuration management is supported extensively through different tools which support the change control process. The project relies essentially on two different repositories: a document repository and a code repository. The document repository is the wiki and since most document artifacts are hosted inside its pages, their changes are registered in the wiki's revision history (although, as mentioned in the Requirements Management section, some naming conventions were adopted to facilitate access to older document versions). The code repository is supported by a version control system called Subversion⁶. The code repository hosts the source code, libraries, scripts, and other resources necessary to the project including documents such as manuals and UML models that cannot be (for practical purposes) stored on the wiki because they are stored in binary formats. Besides the wiki, the previously mentioned web-based bug tracking tool (CodeTrack) has proved to be extremely useful to support the change control process. Proposed requirement changes are submitted as new issues and the status of the requests is managed by the tool which also provides mechanisms that allow status accounting to be performed (through the creation of simple reports, see figure 3).

With respect to the change request process, aside from the registration of the change requests in the tool, these change requests are evaluated by the team as part of the

4 <http://www.junit.org>

5 <http://profiler4j.sourceforge.net/>

6 <http://subversion.tigris.org>

weekly meeting. In this meeting (and depending on the global project schedule and the project phase) every open issue is discussed and some comments on its solution are added in the tracking tool. If accepted, the change is scheduled either for the next iteration or for a subsequent one.

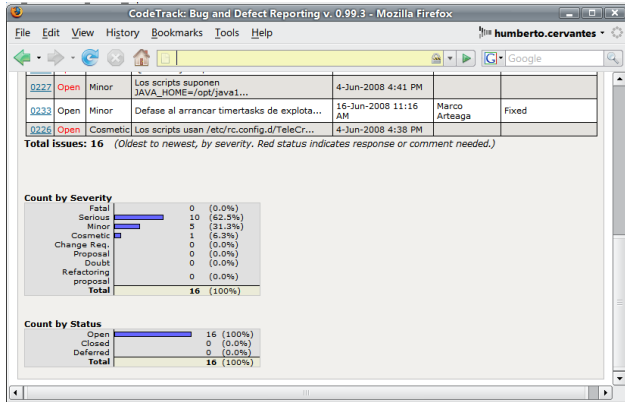


Figure 3: Bug tracking tool showing simple accounting reports

4. Development of the Architecture

Although the quality of the software development process is essential to improve the quality of the end product, another fundamental driver of quality is software architecture. This section provides background information with respect to software architecture and then describes the experiences encountered in the development of the architecture in the context of the project.

4.1. Software architecture

The Software Engineering Institute defines software architecture as:

“The structure or structures of the system, which comprise software elements, the externally visible properties of these elements and the relationships among them” [6].

The importance of software architecture and its relationship with quality lies in the fact that the architecture of a system is a key to supporting what is known as “non-functional requirements”. Non-functional requirements (NFRs) are, as their name suggests, requirements which do not specify the system's functionality but rather characteristics which support this functionality [4]. Among non-functional requirements, quality attributes are properties of the system by which quality will be judged by stakeholders. Quality attributes allow different aspects such as performance, security and modifiability to be defined quantitatively. Other non-functional requirements include constraints and external interfaces.

As mentioned earlier, The Rational Unified Process divides the development project into four sequential phases called Inception, Elaboration, Construction and Transition [2]. With respect to software Architecture, the Elaboration phase is the most relevant. During the Elaboration phase, an executable architecture is created. This is a build that realizes a subset of the architecture and is used to validate that the architecture correctly implements the architecturally significant requirements [7]. The development of the architecture precedes the construction of the system itself, and for this reason, the architecture can be seen as the foundation upon which the system is built. The importance of creating the architecture of the system before building the whole system resides in the fact that architectural decisions are not easily changed and also, it can be very difficult to support some particular non-functional requirement on a finished system if the system was not designed with this NFR in mind from the beginning.

4.2. Architecture in the context of the project

The experience on the development of the software architecture in the context of this project can be divided in two periods. The first one corresponds to the initial release (where there was not much architectural development) and the second one to the subsequent releases. The following sections briefly describe how these experiences took place. Furthermore some details on the complexity of the architecture and the tools that were used to support its development are presented.

4.2.1. Initial release. In the context of this project the development of the architecture was not initially performed very thoroughly. As mentioned in the project background section, the first release of the system had to be developed against a very tight schedule to receive project approval and funding. The project team decided that the architectural driver of the first release would be to focus on implementing functionalities, as the evaluation of this first release depended only on a relatively simple acceptance test procedure which did not focus on NFRs. As a consequence of this, the architecture of the system for the initial release was developed in a very rapid fashion and only provided the basic elements to support the communication between a heavy client and a server. The evaluation of the first release turned out to be positive and the project was approved and received funding. It can be concluded that the decision to not develop the architecture extensively in the first release was the right one, because otherwise the project would probably not have been ready on time and as a consequence it would have been canceled.

The initial problem faced by the development team after the completion of the first release was, however, that the stakeholders demanded more functionality to be added

to the system. This can be understood from the fact that based on the acceptance test results, the stakeholders' impression was that the system was already working and that now it could be extended to accommodate additional use cases (the stakeholders were even already envisioning a 'release' date for the new version).

To overcome this situation, the project team decided to explain to the stakeholders the importance of software architecture and the risks associated with the development of a system where the architecture is not fully taken into account. After several weeks of deliberation, the stakeholders finally realized the risks were real and decided to postpone the release of the second version of the system (this would actually be the first 'real' version of the system).

4.2.2. Subsequent releases. Once the stakeholders agreed on the importance of creating a robust architecture for the system, the development team focused its effort on the creation of such an architecture. The development of the architecture proceeded at a slow pace for several reasons, but chief among them is that the development team wasn't familiar, at the time, with any software architecture development methodology. There were also some particular issues that occurred during the development of the architecture. Firstly, non-functional requirements were not specified with much detail due to the lack of experience by the members of the team in gathering and documenting them. Secondly, the stakeholders didn't have a clear picture of many NFRs and constraints imposed by the end customer on its software purchases. Some of these constraints (such as heavy limitations on communication bandwidth) which imposed complicated design decisions were discovered late and resulted in the need to do several refactorings. At a certain point, this resulted in that the stakeholders became somewhat impatient because the development team did not show any new functionality after several months of programming. In the end, the development of the architecture took nearly one year and, after this period, the system had no new functionalities but it now supported the different constraints and non-functional requirements. At the moment of this writing, the development of the third release of the system is still taking place, fortunately now the stakeholders have expressed satisfaction with the current results (which include new functionalities).

4.2.3. Architectural complexity. Measuring architectural complexity is not a straightforward task. Table 2 presents several measurements which can help appraise the complexity of the architecture that was developed. Some of these measurements (including source lines of code, number of classes and number of components)⁷ are

suggested by G. Booch as being useful 'rules of thumb' for architectural complexity measurement [12]. It must be pointed out that the server was developed using a dependency injection framework called Spring and an object-relational mapping framework called Hibernate [13]. The use of a dependency injection framework provides a way to estimate the number of components that

Description	Value
Number of documented use cases	40
Identified architectural scenarios	15
Number of sub-systems	3 (client, server, license generator)
Number of packages (total)	112 (11 for tests)
Number of Java classes (total)	533 (includes 66 unit tests)
Number of Jars from open source libraries (total)	35
Logical lines of source code (total)	42 KLOC (20K for server, 22K for client)
Number of components in the server (defined as 'beans' declared in spring's application context)	73 (corresponding to 164 classes)
Number of interfaces (generally one per component)	57 (server) + 42 (client)
Number of hibernate object-relational mappings	7

Table 2: Architectural complexity measurements

conform the system (one component is considered for every entry in the dependency injection descriptor, which in the case of Spring is called the "application context"). Another way to estimate the number of components is to count the number of interfaces since one of the project standards imposed the creation of one interface per component. Finally, it must be noted the number of identified architectural scenarios was obtained after the architecture was developed.

4.2.4. Tools used in architectural development.

Although architectural development was not really made following a particular methodology, it was once more supported by a set of open source tools. Architectural documentation was stored in the wiki pages using the traditional 4+1 model that is suggested by RUP [8]. Different tools were used to test the system to evaluate its behavior and how well it satisfied non-functional requirements (see the QA section). On the code side, the development team incorporated several open source java libraries to support aspects such as remote communication, object-relational mapping, transaction management, component composition, etc... It is interesting to notice that at a certain point in the development of a system, the management of reused

⁷ Some of these measurements were obtained using the metrics plugin for Eclipse (<http://metrics.sourceforge.net>)

libraries can become a complex task, in particular it becomes difficult to update the libraries and to manage dependencies between the libraries as they are not explicit (this is a limitation of java).

5. Lessons learned

There are a number of lessons learned by the experience of this project. These lessons can be categorized either as process-related lessons or architecture-related lessons.

5.1. Process-related lessons

One important lesson that was learned from this project is that it is possible to set up a development process to produce quality software in a small team with limited resources. One particularity of the project presented in this article that must be underlined is that the participants had some level of experience in the practices that were established for the project previous to its beginning. This eliminated the need for training. Furthermore, all of the participants “believed” in the benefits of following a disciplined development process. As a consequence, it may be challenging to set up a similar type of development organization with people who do not possess any background on software development methodologies. One particular area that the author considers somewhat difficult to adopt initially is the collection of process metrics, however, this may be due to lack of experience in this field.

Another lesson learned was that a low financial resource constraint can be overcome to a certain degree by the use of open source software at every level of the project. In general, the development team was very satisfied by the use of open source software. The team only faced certain difficulties with some of the libraries that were not totally mature in their development. Overall, the tool that proved to be fundamental for the project was the wiki. It is worth mentioning that inside the wiki a project log page was maintained. This log was very useful when negotiating with the stakeholders to help them “refresh” their memory.

Finally, another lesson that was learned from the project is that it is difficult for a development project to succeed completely if only a subset of the participants follow (and believe in) well established methodologies. Examples of this can be cited with respect to planning and risk management as the stakeholders of the project didn't always provide information necessary for planning purposes upon request and also considered risk management more as an inconvenience than something that could be beneficial. Fortunately, the team never gave up on the consideration of risks, and many problems were avoided thanks to this attitude (although the stakeholders

had the impression that the team's attitude was somewhat “pessimistic”).

5.2. Architecture lessons

Several lessons were learned from the development of the architecture. The first lesson is that this is not an easy task even though it is fundamental. Previous experience and also good design knowledge are required. Long after the architecture was developed, the author became acquainted with the SEI architectural development methods (see Related Work section) but this knowledge was unfortunately not available to the team at the time the architecture was developed. A second lesson that was learned is that it can be difficult to convince the stakeholders about the importance of devoting time to the development of the architecture. This is exacerbated by the fact that the development of the architecture is often not accompanied by an increase in the number of functionalities (at that particular point in time). In the context of this project, the stakeholders had some technical knowledge, which facilitated the explanations greatly but this is certainly not always the case.

6. Related work

As this article relates to a complete development project, there is a huge amount of work that is related to software development (a good deal of the information contained on the SWEBOK [5]). This section only summarizes a few references which may be helpful when undertaking similar types of projects.

Establishing a customized RUP instance for the project has been one of the most difficult tasks of the project. Many of the practices that were presented here had already started being developed previous to the beginning of this project inside several university projects and one software engineering course. Later in the development, the author became aware of a similar public effort to create a 'minimal' unified process instance. The name of this effort is OpenUP/Basic [7]. The benefit of OpenUP/Basic is that it is an open source unified process instance that is oriented towards small development teams. It incorporates many practices that are derived from agile software development methodologies such as Scrum [9]. It is interesting to notice that there are many coincidences between the practices identified as being required at a minimum by OpenUP/Basic and the practices adopted by the development team.

Another important work that must be mentioned is the architecture development effort promoted by the Software Engineering Institute [10]. The SEI provides many tools that help development teams capture non functional requirements (as scenarios), develop the architecture, document the architecture and finally evaluate it.

Finally, it is worth mentioning that there is a quality model called MoProSoft [11] which provides guidelines and best practices for structuring small development organizations. As mentioned in the lessons learned section, software quality cannot be pursued effectively if only a subset of the participants in a development project follow well established methodologies. The work presented here is a good complement to MoProSoft which focuses more on structuring the rest of the organization.

7. Conclusion

This paper has presented an experience report on the development of a network management application in the context of a small Mexican IT firm. The paper presented the different practices adopted by the development team to promote discipline and pursue quality. The experience presented in this paper has lead the author to confirm that it is not impossible for small IT companies to adopt lightweight development processes to improve the quality of their software products. Although the adoption of such processes can be made at a relatively low cost with respect to tools, it is important to consider that adopting them requires training the participants (if they have no previous background) and this can incur in additional costs (which are well worth the benefits).

Although at a first glance it may seem very difficult to set up such the complete set of practices presented in this paper, it seems difficult to try to reduce their number. Furthermore, the choices are corroborated by the similarity of the process adopted for this project and the practices promoted by OpenUP/Basic.

It is also important to mention that it is necessary to change the software development culture among small Mexican IT firms. From informal discussions, it seems that today many small IT firms still develop software in an ad-hoc fashion, however, in this age of competitiveness the companies that do not adopt well established practices are in peril of being left behind. More software engineering education is also required in Mexican universities as the adoption of practices such as the ones presented in this paper require certain theoretical background which is difficult to learn only through practice and which is not covered in many undergraduate programs.

Finally, one last message of this paper is that collaboration between university and enterprises can bear very good results which are beneficial to both parties. It is necessary to promote more collaboration between these two parties.

8. Acknowledgments

The author wishes to thank the development team: Marco Arteaga, Jose Luis Ortiz, and Ismael Nuñez for

their outstanding work and dedication. The team participants were always open to experiment on the author's proposals and were key contributors to many of the practices described in this paper. The author also thanks Alonso Leal from the IT firm that sponsored the project as he let him put his theoretical knowledge into practice at the risk of hiring an unexperienced project manager.

9. References

- [1] A. Clemm, "*Network Management Fundamentals*", Cisco Press, 2006
- [2] P. Kroll, P. Kruchten, "*The Rational Unified Process made Easy*", Addison Wesley Professional, 2003
- [3] Software Engineering Institute, "*The Capability Maturity Model: Guidelines for Improving the Software Process*", Addison-Wesley Professional, 1995
- [4] K. Wiegers, "*Software Requirements, 2d Ed.*", Microsoft Press, 2003)
- [5] IEEE, "*Software Engineering Body of Knowledge (SWEBOK)*", 2004, Available online at www.swebok.org
- [6] L. Bass et al. "*Software architecture in practice, 2d ed*", Addison-Wesley Professional, 2003
- [7] P. Kroll, B. MacIsaac "*Agility and Discipline Made Easy: Practices from OpenUP and RUP*", Addison-Wesley Professional, 2006 (see also <http://www.eclipse.org/epf/>)
- [8] Kruchten, P. "*The View Model of Software Architecture.*" IEEE Software 12 (6), pp. 42-50, 1995
- [9] K. Shwaber, "*Agile Project Management with Scrum*", Microsoft Press , 2004
- [10] Software Architecture Portal at the Software Engineering Institute: <http://www.sei.cmu.edu/architecture/> (retrieved 30/06/08)
- [11] H. Oktaba, "*MoProSoft®: A Software Process Model for Small Enterprises*", SEI Technical report CMU/SEI-2006-SR-001 93 , 2006
- [12] G. Booch, "*Measuring Architectural Complexity*", IEEE Software, July/August 2008
- [13] R. Johnson et al., "*Professional Java Development with the Spring Framework*", Wrox, 2005