Agility vs. Stability at a Successful Start-Up:
Steps to Progress Amidst Chaos and Change

Kurt Madsen
MetaTech, Inc.
18311 Sturbridge Court
Tampa, Florida, U.S.A.
+1 813 298 8180
madsen@metatech.us

ABSTRACT
It is not uncommon for good technical solutions to fail in the marketplace. Equally true, great business opportunities are not always met with appropriate technical solutions. While there can be many causes to such failures, one common problem is the gap between expectations and implementation. Extreme Programming is an excellent delivery methodology for bridging this gap. This paper presents lessons learned from applying Extreme Programming in a start-up environment. In particular, the challenges of meeting and adapting to evolving requirements are presented.

Categories and Subject Descriptors
D.2.9 [**Management**]: Software Process Models, Lifecycle

D.2.2 [**Design Tools and Techniques**]: Decision Tables, Evolutionary Prototyping, Object-Oriented Design Methods

D.2.11 [**Software Architectures**]: General

D.2.1 [**Requirements/Specifications**]: Elicitation Methods, Tools

K.6.5 [**Security and Protection**]: Authentication

General Terms
Design, Experimentation, Management, Security

Keywords
Agile Methods, Application Framework, Software Development Life Cycle, Extreme Programming

1. INTRODUCTION
Start-up companies present unique challenges to the software development process because the pressure to deliver is high, resources are thin, and requirements often change frequently.

To be successful, a start-up team must:

- Build domain expertise in the business.
- Be able to deliver, preferably reusable and adaptable artifacts with future value.
- Ensure that evolving deliverables are always aligned with evolving expectations. In essence, be agile.
- Provide investors with a sense of stability and progress.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

Extreme Programming is ideally suited for this environment. This paper presents our practical experiences applying the Extreme Programming lifecycle to such projects. In particular, it focuses on tools and techniques that facilitate progress amidst chaos and change in start-up environments.

2. SYSTEM OVERVIEW
Since much of our work is in the area of data communications, we have developed an application framework that can be extended to build a variety of networked applications [1]. The system architecture is shown in Figure 1.

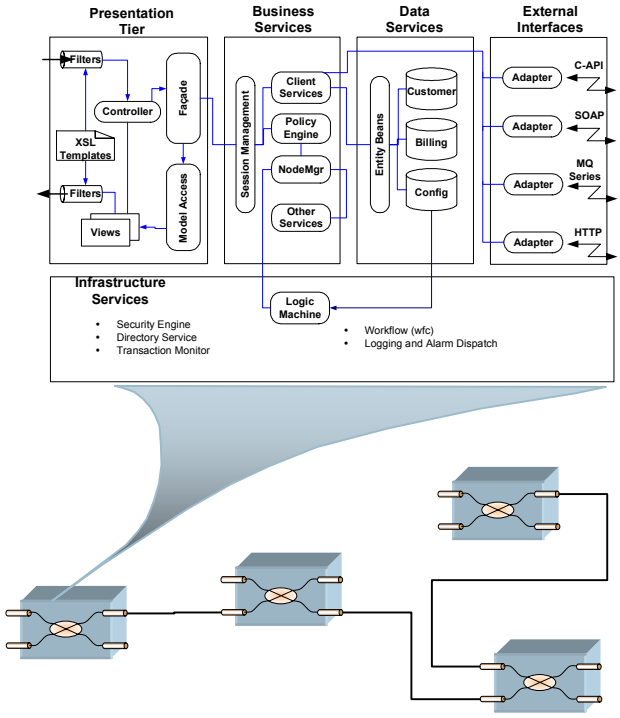

Figure 1. System Architecture

Components are classified in typical n-tier fashion and assembled into one or more nodes. Nodes can be networked together to build switching networks by feeding the output of one node into the input of another. Note that not all components or even tiers are used at every node. For example a switching node will have

network monitoring components but will not have user interface components.

3. DEVELOPMENT LIFECYCLE

We built our lifecycle around Extreme Programming [2]. The lifecycle for this software development methodology is broken down into delivery iterations, each of which consists of three phases. They are exploration, commitment, and steering.

- Exploration Phase – This phase is for information gathering and includes collecting user stories, determining requirements, and performing spikes if needed. (A spike is typically throw-away code used to test an idea.) In practice, this phase was somewhat informal for us since our users are generally on-site.
- Commitment Phase – This is where developers take story cards, estimate scope and effort, and the technology team negotiates deliverables with the users. In practice, our team lead performed this activity with input from the team. Story cards are useful because they enforce succinct descriptions of units of work for development.

Steering Phase – This phase encompasses the rest of the iteration with an emphasis on providing feedback to the users. In practice, we found that this was one of the most valuable aspects of Extreme Programming because it led to requirements evolution and discovery. Additionally, while the users were not always happy, they always felt apprised of the current system status.

4. LESSONS FROM XP PRACTICES

Extreme Programming prescribes twelve key practices. We found that some of these were more useful than others. Our experiences with each of these practices is described below.

4.1 The Planning Game

In software development life cycles considerable time is often spent first capturing and then translating requirements into design specifications. Further, the end product does not always capture the intent of the end users.

In practice, two problems make requirements capture difficult: requirements are often ambiguous and users generally wait too long for implementation feedback in the form of a working system. The Extreme Programming planning game helps considerably by providing a fast iteration cycle in which users specify stories, developers negotiate deliverables, and communication between both groups in the form of early feedback is emphasized.

Still, we initially found ourselves reverting to maintaining requirements documents based on use cases. Such requirements documents are by their nature ambiguous and subject to change.

We wanted to minimize churning implementation to match changing requirements. So rather than move to story cards as prescribed by Extreme Programming, we gave our users a decision table template and asked that they use it to validate their most important use cases. These decision tables were versioned in our source control management tool, Subversion. And they were treated as first class development artifacts, just as important as source code. Collectively, they specified the system behavior. A sample decision table is illustrated in Figure 2.

Use Case: Security access control		RULES				
		R1	R2	R3	R4	R5
EVENT						
e1	Principal requests access to URI	1	1	1	1	1
CONDITIONS						
c1	Session exists	0	1	1	1	1
c2	Access authorized for this URI	*	0	1	1	1
c3	Access method A applies	*	*	1	0	0
c4	Access method B applies	*	*	0	1	0
c5	Access via SAML	*	*	0	0	1
ACTIONS						
a1	Retrieve session	1	1	1	1	1
a2	Encrypt assertion data	0	0	1	1	1
a3	Redirect via method A	0	0	1	0	0
a4	Get security token	0	0	0	1	0
a5	Redirect via method B	0	0	0	1	0
a6	Forward assertion via SAML	0	0	0	0	1
a7	Log session in audit trail DB	0	0	1	1	1
a8	Deny access	0	1	0	0	0
a9	Create session	1	0	0	0	0

Figure 2. Decision Table

The decision table is interpreted as follows: zeros are false, ones are true, and stars match either boolean value. One event, in this case e1, is the starting point into the table's logic. The set of conditions effectively define the current state of the system. Given the input event and the set of conditions, one or more actions are executed.

We worked with our users to get them to the point where they could maintain their own tables. After specifying decision tables, the users received immediate feedback in the form of modified system behavior. They used this process to experiment and discover new requirements, while we received a well-formed and unambiguous system specification. Consequently, our development team spent less time in analysis and design.

We built a logic machine, to load these tables as reference code and execute them using framework code – it's fast and accurate. The logic machine is essentially a configurable server that behaves as a finite state machine, but can handle many conditions and actions in one event dispatch cycle.

Figure 3 shows the behavior that results when the logic machine loads and executes the decision table from Figure 2. Note that event e1 initiates the execution cycle which invokes several actions, a1 through a9. (Action a9 is not shown.)

In this example, the Principle is a user with a browser client and the logic machine is running on an authentication and authorization server. The Principle seeks access to protected URLs, which are only accessible via one of three security protocols that are specified by the decision table. The first protocol, called method A, applies rule 3 (which executes actions a2 and a3) to encrypt user credentials and pass this data as hidden text fields in an method A redirect. The Principle is then redirected to the resource owner who decrypts the authorization credentials. The second protocol, method B, applies rule 4 (which executes actions a2, a4, and a5) to encrypt the authorization credentials, get a security token from the URL owner (generally a 3rd party), and finally send this data to the principal via a method

B redirect. A security token is generally valid for a pre-defined time period during which access is valid. The third protocol is based on the security assertion markup language (SAML) [3]. It applies rule 5 (invoking actions a2 and a6) to encrypt the authorization credentials and assert the Principle's identity to the URL owner via a SAML assertion message.

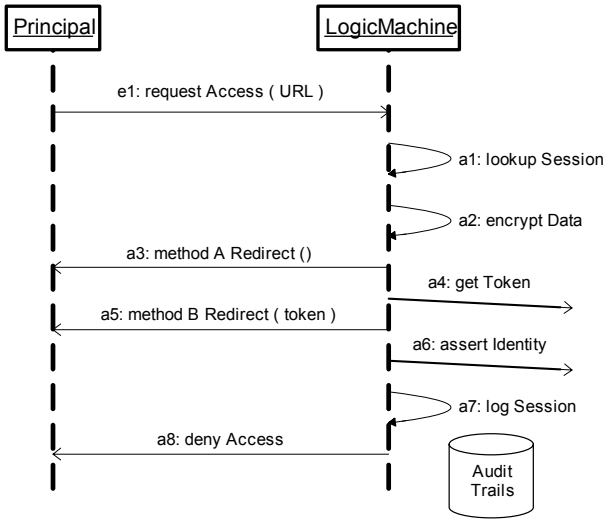

Figure 3. Desired System Behavior

Note that Figure 3 highlights portions of all three protocols for the purpose of illustrating the flexibility of the decision table approach. From this experience, we learned the value of having a simple tool to quickly implement user stories and new requirements.

Several lessons came out of our planning game experience.

> LESSON: Put your customers to work by letting them maintain specifications under supervision.
>
> LESSON: Unambiguous requirements align delivery with expectations and reduce developer workload.
>
> LESSON: Rapid feedback leads to better exploration of the problem space and requirements discovery.

4.2 Simple Design

Another XP practice is simple design. As Albert Einstein once said, "make everything as simple as possible, but not simpler." In our case, this meant using only one decision table template and resisting the temptation to overload the functionality of the logic machine.

Regarding implementation, we first considered using the State pattern [4], which can be used to build a finite state machine that is easily extended. But, the decision table approach turned out to be a simpler design because of the number of states (combinations of conditions) and the fact that each dispatch cycle generally invoked several actions.

Figure 4 shows the static structure of the logic machine used to load and execute decision tables.

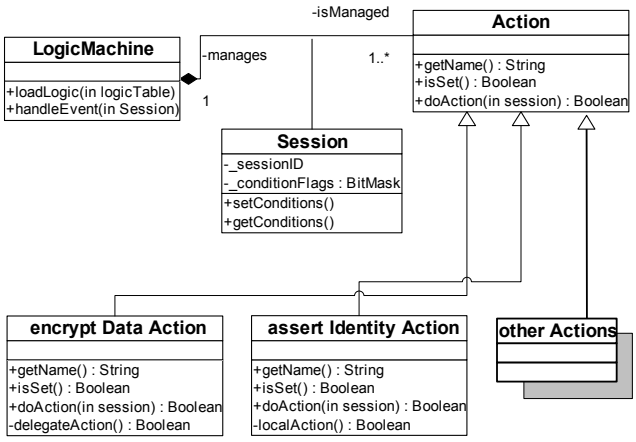

Figure 4. Logic Machine Implementation

The logic machine has two methods, loadLogic() and handleEvent(). At system startup, loadLogic() loads each decision table into a HashMap of rules. The hash key for each rule is the bit mask of conditions and the lookup value is a set of references to actions. Note that for each rule, the set of actions are organized along the lines of the Pipes and Filters pattern [5].

For example, the HashMap entry for rule r4 is:

Key = {1, 1, 1, 0, 1, 0}

Value = {a1, a2, a4, a5, a7}

This process is repeated with one new HashMap created for each decision table. (In general, we only used one decision table and one HashMap.)

At runtime, handleEvent() is passed a session from which it obtains the set of current conditions (i.e., the bitmask). Using this condition set as a key, it finds the set of actions that apply for that rule. Iterating over the set invokes all actions for that rule. For each action, if isSet() == true, then doAction() is invoked. We later changed this to only load those actions where isSet() == true for a given rule. In other words, it is not necessary to load actions when isSet() == false because they will never be invoked.

Note that some action classes, such as A2, implement local actions while others, such as A1, delegate an event and session context block to other handlers or even applications. It is important to ensure that the processing time consumed by such handlers does not exceed (or even block) the logic machine's processing cycle. The Active Object design Pattern is an excellent approach to dealing with this problem [6].

In addition to the logic machine, we needed a concurrent server for asynchronous event de-multiplexing and dispatching. We considered the Reactor pattern and more generally the ACE communications framework [7] as well as various options using J2EE and EJB. The team had good J2EE experience, but we needed a simple solution with good performance. We opted to use a concurrent server based on work by Comer and Stevens [8].

The main loop of the server is essentially two lines of code:

```
SCB scb = sessionPool.retrieve( SessionID);
logicMachine.handleEvent (e, scb);
```

This implementation is simple to maintain and it is fast. From an external point of view, the behavior is what matters; the implementation details are irrelevant as far as the users are concerned. Keeping the implementation simple allowed us to focus meeting user expectations.

LESSON: Keep it Simple.

4.3 Metaphor

The Metaphor practice within Extreme Programming refers to maintaining a common system description that guides communication and development. This practice was not clearly understood by the team (other than the obvious need for common understanding among team members). Thus, it was not intentionally implemented.

4.4 Test First

Writing unit tests before developing code forces developers to specify the acceptance criteria before coding features. This practice is immensely valuable, but takes time to set up initially which can make it difficult to justify in a start up environment. It is particularly valuable for regression testing when implemented together with continuous integration. This permits features to be added incrementally without breaking previous work.

Aside from using Junit, we did not spend enough time up front to get set up and establish a regression test suite. However, the effort that we did do in this area was time well spent. As of this writing, we are just getting started with Clover.

4.5 Re-factoring

Re-factoring in Extreme Programming provides agility. It also gives developers the courage to make significant changes under pressure (see continuous integration.) In practice, it is often difficult for developers – and investors – to accept the fact that it is sometimes necessary to delete code. Discarding or changing code that works is often seen as a step backwards, particularly when the new code that replaces it does not work.

It is important to understand that the activity of re-factoring code is itself a deliverable because it hones the teams skills and creates an agile delivery pipeline.

We found it useful to pursue axiomatic development by using formal transformations such as boolean algebra [9], in particular DeMorgan's Laws. These transformations facilitate rapid re-factoring while preserving program correctness. For example, the law:

$$\overline{(A \vee B)} \leftrightarrow \overline{A} \wedge \overline{B}$$

can be used to verify that the following two statements are equivalent:

```
if (! (user.role() == "manager" || access == "allow"))
if (user.role() != "manager" && access !=  "allow")
```

Real-world business logic can be substantially more complicated. Such formal transformations reduce the risk of re-factoring and build confidence among team members. In this sense, the act of re-factoring is itself a deliverable. This practice combined with tight SCM control ensured axiomatic development – we always had a working integration code base with minimal branching.

LESSON: Use Formal Transformations.

4.6 Continuous Integration

Continuous integration is critical to stability and quality. It allows teams to have courage – an Extreme Programming value – when making changes under pressure. We made the mistake of asking management for time to implement and test our build/integrate/release cycle early in the project lifecycle. We did this by pitching continuous integration as a first class feature worthy of being placed on the project schedule. So, when it came time to negotiate user stories, continuous integration was not approved. This decision was due the pressures of a start-up as well as management's failure to recognize the value of continuous integration (or our failure to communicate this value).

For future projects, we recommend that the development team set up infrastructure elements such as continuous integration quickly at the beginning of the project and without rationalizing it with users. If the first few user stories are late, yet a continuous integration process is in place, the project will be much better off in the long-run.

LESSON: It is easier to ask for forgiveness than permission. Don't seek approval for CI; just set it up.

4.7 Pair Programming

Unlike continuous integration, which can be set up at the beginning of the project under the radar, pair programming is an XP practice that must be sold to management because staffing, budget, and expectations are more visible. Unfortunately, the concept of pair programming is still difficult to sell.

Too often, companies put several programmers together in one conference room, each working on different tasks, and call it XP. Assigning two developers to work on the same task is anathema to business leaders.

Our primary observation is that both developers must be assigned the same task in order to fully realize the benefits of pair programming. This is partly due the psychology of a software development team: egos are strong, and people are often reluctant to ask dumb questions. Without true pair programming, these questions – and bugs – linger in the project.

LESSON: Both developers must be assigned the same task.

LESSON: The team culture must encourage asking dumb questions.

4.8 Collective Ownership

In any complex system, developers inevitably develop specialized knowledge, which can put the project at risk if someone leaves the project. To build common understanding of the system and a sense of collective ownership, our management periodically asked a random developer to give a brief presentation, similar to a lightning talk, on one aspect of the system. The developer was

generally asked to present a portion of the system that they he or she not develop. Advance notice was short, sometimes only fifteen minutes, which kept preparation time to a minimum. The intent was that this practice should not interfere with on-going development.

LESSON: Collective ownership is built on common technical understanding and random lightning talks.

4.9 Maintain a 40-Hour Week

We found it difficult to adhere to this XP practice. After months of overtime, people burned out and some left the team.

Unfortunately, significant overtime in the form of a four to six month crunch period is somewhat inevitable at a start-up company because market windows are small and serious effort is required by developers to bring a product to market. Promises of stock options keep staff on-board during this critical build phase. But once the product has been released into the market, sales and support become top priorities and staffing levels are generally cut back.

The best way to address work environment quality issues is to communicate with investors and management regarding the effect of stress and overtime on peak productivity rates. Proper resource leveling and upfront communication with the team builds loyalty.

LESSON: Burned-out Developers are not Productive.

4.10 Coding Standard

We failed to take the time up front to agree upon one set of coding standards. Coding standards become important to ensure staff substitution and enterprise integration, particularly in larger organizations.

4.11 Small Releases

Our experience has been that when properly controlled, frequent, small releases lead to more stable production releases. This requires tight control over source code control and release management.

One best practice is to only have one set of project source files at all times: the production set. Maintaining multiple sets of files for various purposes (e.g., development, QA, demos, production) is bad practice. Also, hard-coding development or test or demo data into production source files is bad practice.

Some examples of best practices in release management include the following [10]:

1. The build process and source code artifacts should be kept self-contained: they should not depend on specific target environments such as one developer's PC. Also, the build manager role should rotate among the team.
2. Release-specific properties should be isolated to one place such as the project properties file. To change properties that are specific for a particular target environment, the build script should first source the production properties file followed by a second properties file to override settings specific to a non-production environment (e.g., DEV, STAGE, CTGY). If the second properties file does not exist, then a production release will be built.

Regarding the second point, each developer may have his or her own properties file, my.properties, which is not checked in under source control. In this way, each developer can alter properties for testing without introducing developer-specific dependencies to the project source tree. Thus, a demo on demand can be build quickly by substituting the current build file.

LESSON: More frequent, smaller releases are less risky.

4.12 On-Site Customer

Users have the annoying habit of requesting demos at the worst times, typically interrupting the development process. We found that to maintain good communication between users and technical staff, users must feel welcome to visit the development environment anytime with minimal advance notice. This open house policy creates an atmosphere of mutual respect between business users and developers. Figure 5 shows how we involved our business domain experts in the requirements discovery and story creation process.

The key to making this feedback cycle work was our ability to reconfigure the system by editing decision tables, configuring components, and rebuilding the application. Here is where all of the practices in Extreme Programming come together.

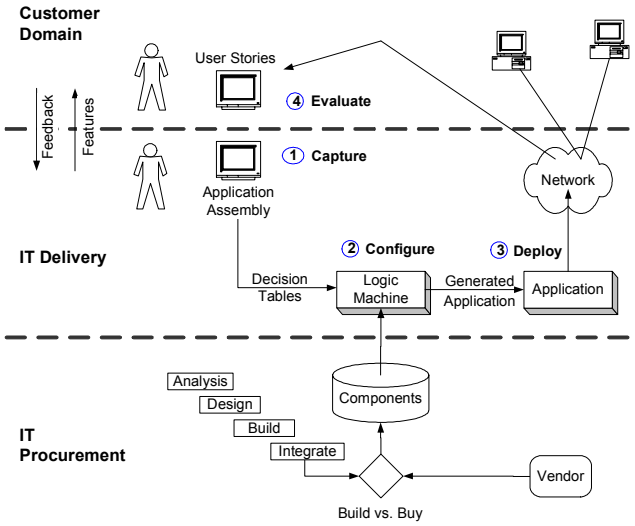

Figure 5. Experimentation Drives Requirements

The result was that requirements evolved in sometimes unexpected ways. This models the way people naturally solve problems. One cannot learn to swim by writing requirements documents. Rather, one must jump in the water and experiment.

LESSON: People Learn by Doing.

LESSON: Always be Ready for Demo on Demand.

5. FURTHER RESEARCH

Future plans include applications in new domains as well as parameterized conditions. Figure 6 provides an example of a decision table in the aviation domain in which condition c3 has a parameter, $18. This parameter represents the cross-wind component of the current wind speed and direction at the runway: this crosswind component must be below 18 knots to land safely.

Parameters are implemented by establishing buckets of range values, with one bucket for each rule. In this example, rule 1 applies if the crosswind condition, c3, is less than 3 knots, rule 2 applies if it is less than 6 knots, and so on by increments of 3 knots per bucket. Since the actual parameter for condition c3 is 17 knots, only rules R1 through R6 apply.

Use Case: Aircraft landing		RULES							
		R1	R2	R3	R4	R5	R6	R7	R8
EVENT									
e1	Landing sequence begins	1	1	1	1	1	1	1	1
CONDITIONS									
c1	Runway length is sufficient	0	0	0	0	1	1	1	1
c2	Landing gear is down	*	*	*	*	0	1	0	1
c3	Crosswind < $18 knots	1	1	1	1	1	1	0	0
ACTIONS									
a1	Emergency gear extension	0	0	0	0	1	0	0	0
a2	Calculate final approach	0	0	0	0	1	1	0	0
POST ACTION									
a99	Decide to land	0	0	0	0	1	1	0	0

Figure 6. Decision Table with Parameters

Rules R1 through R8 in Figure 6 are interpreted as follows:

- R1 through R4: $\overline{c1} \rightarrow \overline{a99}$. The runway is too short.
- R7 and R8: $\overline{c3} \rightarrow \overline{a99}$. It is too windy to land.
- R6: $c1 \wedge c2 \wedge c3 \rightarrow [a2] \wedge a99$. The aircraft can land.
- R5: $c1 \wedge \overline{c2} \wedge c3 \rightarrow [a1 \wedge a2] \wedge a99$. The aircraft can land after the emergency landing gear procedure is complete.

Rules 5 and 6 highlight another area of current research, the ability to specify post-actions. This is not the same as a post condition in a use case. In our implementation, the order of execution of actions is not guaranteed. But, it is often desirable to have one action that executes last, which typically depends on the outcome of the previous actions. In this example, the final action, a99, is the decision to commit to a landing, which must follow actions a1 and a2.

The brackets above act as state guards [11] to ensure that the post action a99 executes last. For example in rule R6, action a2 must execute before post action a99. Likewise, in rule R5, actions a1 and a2 must execute before post action a99.

Note, that this table assumes that action a1 succeeds. This action initiates the emergency gear extension process. If this action were to fail, then a landing would not be possible. Thus, rule R5 changes condition c2. We are currently experimenting with multiple passes through the table to process one event. But, iteration cycles must be marked to avoid infinite loops as would result if the emergency gear extension process fails.

6. CONCLUSSION

There are many factors that determine the success or failure of a software project. Our experience in applying the Extreme Programming methodology at a start-up company was, on the whole, very positive.

One general observation is that start-up companies are at risk of being "penny-wise, pound-foolish." Examples of areas that did not receive enough attention early in the project were setting up continuous integration and release management, establishing coding standards, and properly emphasizing test-first development. Also, in an effort to conserve scarce funding, we used open source for everything even though, in some situations, expensive products were appropriate. Our labor costs eventually exceeded the initial savings. Still, these challenges are part of what make start-ups exciting.

Our advice for future projects is that the value of unambiguous requirements capture as well as the requirements discovery that results from quick iterations and feedback cannot be overemphasized. In summary, Extreme Programming helps bring agility and stability to start-up environments.

7. REFERENCES

[1] Madsen, K. Five Years of Framework Building: Lessons Learned. In *18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, conference companion*, pages 345–352, Anaheim, CA, Oct. 29–30, 2003. ACM Press, New York, NY.

[2] Beck, K. *Extreme Programming Explained – Embrace Change*, second edition. Addison-Wesley, 2004.

[3] Lockhart, H., Hughes, J., Maler, E. *Security Assertion Markup Language 2.0 Technical Overview*, working draft 03. OASIS Open. Feb. 20, 2005. www.oasis-open.org.

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 331–344. Addison-Wesley, 1995.

[5] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*, pages 99–122 and 53–70. John Wiley & Sons, 1996.

[6] Lavender, G., Schmidt, D., Active Object – An Object Behavioral Pattern for concurrent Programming. *Proceedings of the Second Pattern Languages of Programs conference* in Monticello, IL. September 6–8, 1995.

[7] Schmidt, D. Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing. *C++ Report, SIGS*, Vol. 5, No. 2, pages 1–12, February, 1993.

[8] Comer, D., Stevens, D. *Internetworking with TCP/IP Volume III: Client-Server Programming and Applications,* Linux/POSIX Socket Version, pages 143–150. Prentice Hall, 2000.

[9] Roth, C. *Fundamentals of Logic Design*, third edition, page 37. West Publishing Company, 1985.

[10] Burke, E. *Top 15 Ant Best Practices*, O'Reilly OnJava.com., Dec. 17, 2003. http://www.onjava.com/pub/a/onjava/2003/12/17/ant_bestpractices.html

[11] Fowler, M., Scott, K. *UML Distilled*, second edition, page 119. Addison-Wesley, 2000.