Outlining Software Productivity Aspects in Feature Driven, Lean Software Development and Pragmatic Programming.

Kleber Silva, Augusto Sampaio, Alexandre Vasconcelos.

Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 785 – 50.732-970 – Recife – PE – Brasil

{khfts, acas, amlv}@cin.ufpe.br

Abstract. *Software companies eagerly struggles to identify points where software productivity can raise up in order to be increasingly competitive in the market. Academic studies and researches could not still give a final word or a silver bullet to solve the issue of understanding software productivity and what really affects it, i.e. the software productivity drivers, due to the great amount of variables related to the different nature of project contexts subject to continuous and unexpected amount of change. Agile methodologies such as Feature Driven Development and Lean Software Development have tackled this issue by avoiding software productivity bottlenecks related to unsatisfactory response to changes. Also, pragmatic programming which embodies several tips to help pursue high software productivity is slightly examined. This work intends to gather, analyze and outline software productivity aspects found in those two agile methods and pragmatic programming approach and then suggests a simple framework to bind them all together.*

1. Introduction

Software productivity gains have been seen by software companies throughout the world as a major issue that needs to be addressed seriously and effectively. Fast responsiveness to changes appear as a dramatically challenge due to the great variety of project profiles with highly dynamic requirements. Traditional processes tend to be thought as of heavy ones. This issue has been tackled by agile methodologies such as Feature Driven Development (FDD) along with Lean Software Development (LSD) which can be seen as an alternative to develop projects improving software productivity. Also, experience has shown that there are many tips for the programmers to follow that can help augment their own productivity level. This is where pragmatic programming (PP) comes in. This work gathers best practices found in FDD, LSD and tips from PP under software productivity lens and then suggests a simple framework to bind them all together. This paper is organized as follows. Section 2 gives an overview on FDD, LSD and PP. Section 3 briefly discusses some major aspects on traditional processes and compares them with others found in agile methodologies according to the lens of productivity issues. Section 4 discusses aspects on FDD, LSD and PP along with the identification of software productivity aspects. Section 5 proposes a simple framework to combine the best practices found in FDD, LSD and PP. Section 6 concludes our work and suggests

future experiments that can be carried out to validate the proposed framework effectiveness in improving software productivity.

2. FDD, LSD and Pragmatic Programming

2.1 FDD Overview

Feature Driven Development (FDD) is an agile and adaptive approach for developing systems. The FDD approach does not cover the entire development process, but rather focuses on the design and building phases (Palmer and Felsing 2002) [1] . FDD is built around the notion of features which means a small, client-valued function in the form of <action><result><object>, i.e. “calculate the total of a sale”. Usually, a feature implementation must be fit in a 15-day timeframe. FDD emphasizes quality aspects throughout the process and includes frequent and tangible deliveries, along with accurate monitoring of the progress of the project. It comprises only five sequential processes depicted in Figure 1 [2]. Below follows a brief explanation on its processes [3]:

1. **Develop an overall model –** An initial project-wide activity that implies a thorough understanding of the domain is carried out;
2. **Build a features list –** An initial project-wide activity which comprehends the identification of the list of features to support the requirements.
3. **Plan by feature –** An initial project-wide activity to elaborate the development plan. The features are distributed among developers considering factors such as function dependencies, work load, complexity or risk and project milestones;
4. **Design by feature** – A per-feature activity to produce the feature design package.
5. **Build by feature -** A per-feature activity to produce a completed client-valued function (feature).

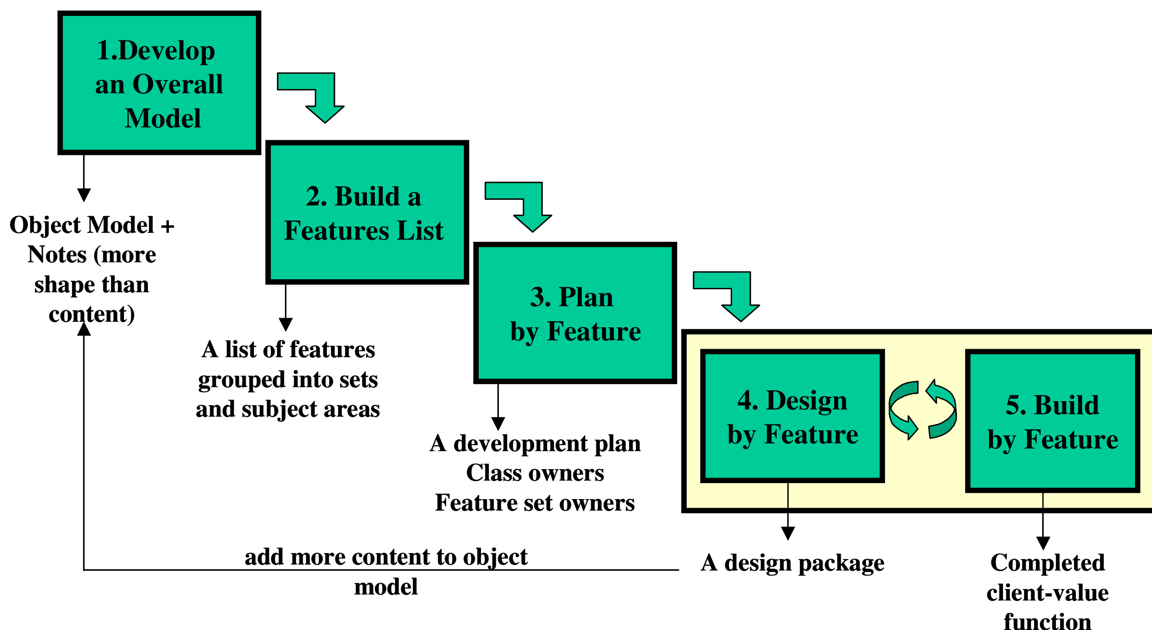

Figure 1 - FDD processes

FDD then involves the following practices [4]:

- **Domain Object Modeling**: Exploration and explanation of the domain of the problem. Results in a framework where features are added.
- **Developing by Feature**: Developing and tracking the progress through a list of small functionally decomposed and client-valued functions.
- **Feature Teams**: Refers to small, dynamically formed teams.
- **Inspection**: Refers to the use of the best-known defect-detection mechanisms.
- **Regular Builds**: Refers to ensuring that there is always a running demonstrable system available. Regular builds form the baseline to which new features are added.
- **Configuration Management**: Enables the identification and historical tracking of the latest versions of each completed source code file.
- **Progress reporting**: Progress is reported based on complete work to all necessary organizational levels.

2.2 Principles of LSD

LSD was drawn from "The Lean Thinking" which emerged from Toyota, in their approach of "Lean Manufacturing" in the 1980´s. LSD stems from the work of Mary and Tom Poppendieck on transferring principles and practices from manufacturing environment to the craft of software development. LSD has seven principles [5]:

- **Eliminate Waste** – The most important principle of lean recommends looking for activities or tasks which add no value to customer. These can be considered as waste. Seeing and eliminating waste is the first step to a lean value stream. Making value flow rapidly from receipt of a request to delivery is a fundamental principle of lean thinking.
- **Amplify Learning -** A lean development environment focuses on increasing feedback, and thus learning. The primary way to do this in software development is with *short*, *full-cycle* iterations. Short means a week to a month. Full cycle means the iteration results in working software: tested, integrated, deployable code. There should be a bias to deploy every iteration into production, but when that is not possible, end users should simulate use of the software in a production-equivalent environment. [6]
- **Delay Commitment** – Encapsulation and loose coupling are the key mechanisms for delaying commitment. Although these techniques have been known for many years, object-oriented design brought them to the forefront of software development. To these we add refactoring (improving design as code is developed) and automated testing, which are essential for keeping code changeable not only during development, but throughout its lifetime.
- **Deliver Fast** - The ability to deliver fast is the mark of excellent operational capability. The whole idea of delaying commitment is to make every decision as

late as possible, allowing you to make decisions based on the most current knowledge. It makes no sense to delay commitment if you can't deliver fast. Speed decreases the length of the feedback loop and means you are acting on the most current information possible.

- **Empower the Team** - When the flow of work is rapid and responsive, there is no time for central control. The work environment should be structured so work and workers are self-directing. People, not systems, develop software.

2.3 Pragmatic programming approach

Pragmatic programming (PP) [7] is indeed a set of programming "best practices". It consists of a collection of 70 short tips, focusing on day-to-day problems. These practices take a pragmatic perspective and place focus on incremental, iterative development, rigorous testing, and user-centered design.

According to [8]: "*Walking on water and developing software from a specification are easy if both are frozen. But in today's business climate, everything is undergoing constant change; nothing is frozen. While the new Agile methodologies (include eXtreme Programming, or "XP") are a huge step in the right direction, they still don't offer a silver bullet. You can fail with an Agile methodology just as easily as you can fail with the Rational Unified Process, or with CMM-inspired methodologies. Something's missing. We call that missing ingredient Pragmatic Programming, an approach that is language, operating-system, and methodology neutral. You can adopt Pragmatic Programming practices in any environment and begin to write better code, faster--and even have more fun in the process.* "

3. Software productivity lens: Traditional or Agile processes

Around 1980, Barry Boehm published data showing how the cost of change and rework seems to increase approximately tenfold for each software development phase requiring rework activities as a result of the change [9]. A typical software project experiences a 25% change in requirements [10]. Waterfall model can be given as an example of a traditional process approach that does not seem to cope well with frequent changes because its rationale is to prevent rework stemming from changes and defects by getting things right upfront [11]. But, in practice, when no incremental or iterative development is running, only at the end of the project the product is released. That leads to make large amounts of rework appear late in the development when cost of change is very high. Although agile methodologies have come to fill in this gap, scientifically grounded empirical evidence is, however, still very limited [12]. A trend to balance agility and discipline has come up to unite their strengths [13] resulting in an increase in software productivity. A good example can be CMMI balancing both plan-driven (traditional) and agile methods [14]. Lockheed Martin M&DS evolved from SW CMM ML2 (1993) to ML 3 (1996) to CMMI ML5 (2002) with an increase in software productivity of 30% [15]. In his studies, Bradford Clark [16] shows that a one-CMM-level improvement by itself accounts for only an 11% increase in productivity. Improvements on software process might respond to low amount of percentage on software productivity. Human factors, on the other hand, still appear as a strong software productivity determinant. We examine LSD principles and PP best practices which involves human factors issues such as leadership, motivation and so on. This was the reason we chose them. They can add

value by giving tips on how to better manage personnel issues then resulting in a boost in software productivity.

4. Software Productivity Aspects

4.1 The Issue of software productivity

Many software engineering studies and experiments have been undertaken to determine which factors affect software productivity. Nevertheless, a consensus is yet to be reached because these specific studies or experiments lead to misleading or paradox information due to context-dependent variables. Software productivity lacks certainty on those variables often because of mismeasurement: How and what you measure determines how much productivity you see [17]. Caper Jones [18] states that an analysis of common units of measure for assessing program quality and programmer productivity reveals that some standard measures are intrinsically paradoxical. Although the amount of misleading data, we all agree that identifying and fostering software productivity improvements can be thought of as a way to reduce noise level activities. Noise level activities can be seen as high cost, time-consuming or non-productive time-stealing tasks. An example can be fixing a bug in an object-oriented project where no version control takes place and class concurrency use is large. Learning to recognize, identify, and correct noise-level activities can have a tremendous positive impact on the organization's ability to deliver projects on time, within budget, with the desired features [19]. Unfortunately, sometimes measures that organizations typically take to improve the productivity or reduce noise level activities have the opposite effect, i.e. pressure people to put in more hours, mechanize the process of product development, compromise the quality of the product and standardize procedures. These steps potentially make work less enjoyable and less satisfying. Then improving productivity risks worsen turnover [20]. This is addressed by methodologies such as FDD, LSD and PP, because all their practices and principles come from many years of successful software industry day-to-day projects. In this paper, we suggest some aspects from FDD, LSD and Pragmatic Programming which can be bind together to tackle noise level activities or improve software productivity.

4.2 Feature Driven Development

In this section, we outline some aspects in the FDD that can help achieve a higher software productivity levels or reduce noise level activities.

- **Features** - FDD supports features which are small client-valued functions. In the Develop an Overall Model process, the decomposition of the domain area into features is made by nurturing functionalities which adds value to the customer with real tangible benefits to the business. Through this perspective, a higher productivity level can be achieved by avoiding the implementation of useless functionalities which can be a very costly activity.

- **Class Code Ownership** - Another point is the FDD supports class code ownership so the allocation of developers is done based on the owners of the classes (developers) likely to be involved in the development of the feature(s),

The Chief Programmer who is assigned a set of features forms a Feature Team. Considering projects with low turnover, this can speed up development time because the feature is likely to be done by someone who is specialist on the classes responsible to implement it without interruption.

- **Frequent Releases, Regular Builds -** As stated earlier, features should fit in a short timeframe, i.e. 15-day period. This allows for a faster feedback on the pace of the implementation activities. Also, regular builds allow for a rapid detection of failures when different modules are integrated. By receiving these feedbacks, the project manager can act upon the problem as soon as it occurs. This avoids having to troubleshoot an issue when the project is almost finished which can lead to enormous waste of time.

- **Configuration Management –** To avoid unnecessary rework on code implementation or maintenance because of code version unmanageability. Applying no version control can result on a significant augment on code maintenance or implementation time.

- **Progress Reporting** – A very important FDD characteristic that permits the track of project progress. It consists of several tools to support automatic generation of reports to allow fast and intuitive colored visualization [21] of the progress of features implementation. It constitutes a powerful tool for project managers to proactively take decisions and greatly diminishes the time spent on the preparation of manual reports. An example is depicted in **Figure 2**.

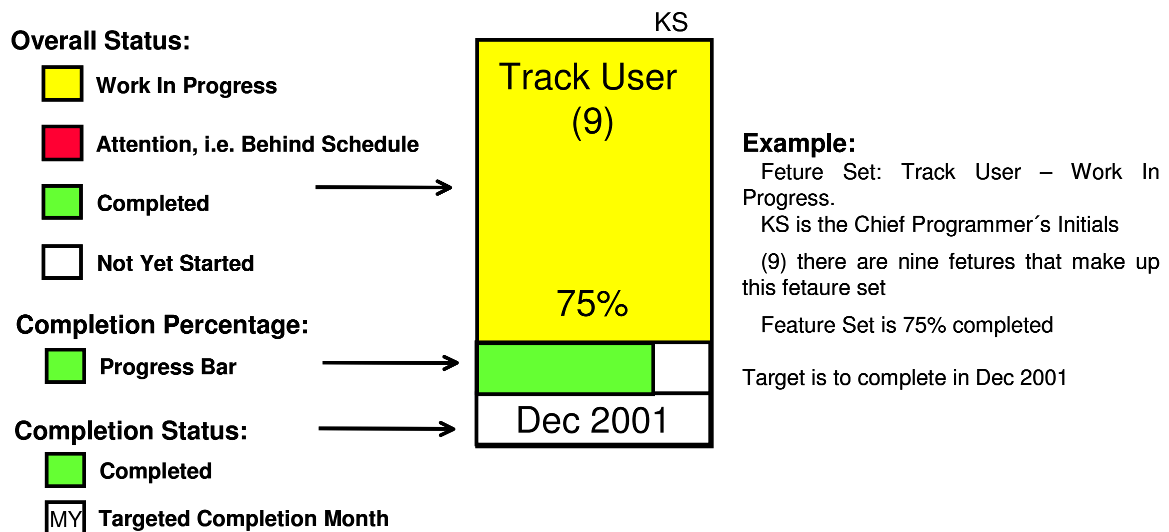

Figure 2 - Graphical Progress Summary Report

4.3 Lean Software Development

The principles of Lean Software cannot be viewed as a recipe for software development, but guideposts for devising appropriate practices for your environment.[6] Lean is responsible for colossal improvements in productivity and quality over the past 32 years and is used successfully by industries that range from manufacturing and logistics to

product development. Mary and Tom Poppendieck [22] provide twenty two tools for converting lean principles into agile software development practices. In this section, we will discuss these principles/tools evidencing issues about software productivity:

Lean Principles	Tools
1) Eliminate Waste	1) Seeing Waste; 2) Value Stream Mapping.
2) Amplify Learning	3) Feedback; 4) Iterations; 5) Synchronization; 6) Set-Based Development.
3) Decide as Late as Possible	7) Options Thinking; 8) The Last Responsible Moment; 9) Making Decisions.
4) Deliver as Fast as Possible	10) Pull Systems; 11) Queuing Theory; 12) Cost of Delay.
5) Empower the Team	13) Self-Determination; 14) Motivation; 15) Leadership; 16) Expertise.
6) Build Integrity In	17) Perceived Integrity; 18) Conceptual Integrity; 19) Refactoring; 20) Testing.
7) See the Whole	21) Measurements; 22) Contracts.

Figure 3 - LSD with 7 Principles and 22 Tools

- **Eliminate Waste**
  - *See Waste, Value Stream Mapping* – Identifying activities which add no client value and eliminating them before they can be considered for a project is a great way to obtain software productivity gains. In Figure 4, The Standish Group [23] shows that 45% of functions built are never used. So, it is expected to think that productivity gains are likely to occur largely by avoiding non-value adding activities. Value Stream Mapping (VSM) is a visualization tool oriented to the Toyota version of Lean Manufacturing (Toyota Production System). It helps to understand and streamline work processes using the tools and techniques of Lean Manufacturing [24]. It is a broader technique to find value added time versus wait time activities.

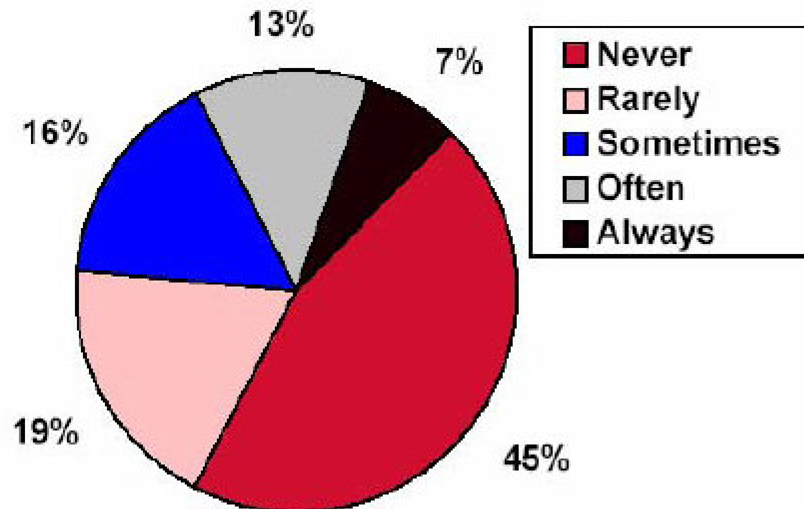

Figure 4 - Features and Function usage. The Standish Group International 2002.

- **Amplify Learning**
    - *Feedback, Iterations, Synchronizations and Set-Based Development* – As soon as problem develops, feedback loops must take place. For instance, instead of letting defects accumulate, running tests as soon as the code is written illustrates prompt feedback. The longer it takes to identify defects the more costly to fix them at later phases. So fast feedbacks are invaluable to foster higher productivity levels. The same rationale applies to iterations, synchronizations. Set-based development relies on a principle of communication about constraints not choices. The intention is to defer choices until they have to be made. The idea here is to avoid wasting time looking for choices.
- **Decide As Late As Possible**
    - *Options Thinking, Making Decisions, The Last Responsible Moment* – It means to delay a decision, keeping your options open as long as practical, but no longer. Deciding upon learning not prediction. Drilling down to details too fast can lead to big mistakes. For instance, avoid depth-first approach when establishing the requirements. Intuitive decision approach should be preferred over rational decision that normally ignores the instincts of experienced people. In terms of software productivity, these are abstract issues because they are based upon human factors, i.e. self-awareness, tactfulness, strategic abilities that when conducted well can lead to avoiding waste of time and resources.
- **Deliver As Fast As Possible**
    - *Pull Systems, Queuing Theory, and Cost of Delay* – In a fast-moving environment only coordinated work on a self-organization basis works, i.e. where people can act upon a visual control or management by sight and everyone knows what needs to be done, the problems and progress of the project, *Pull Systems*. Also some degree of parallelism to serve customer requests and short deliveries are recommended according to the *Queuing Theory*. An economic model may be created to outline the benefits of fast responsiveness, *Cost of Delay*.

- **Empower The Team**
  - *Self-Determination, Motivation, Leadership, Expertise*. – Tom De Marco gives an approach of success with this formula: 1) get the right people; 2) make them happy so they will not leave; 3) turn them loose [20]. These issues are about turning people loose to achieve a higher level of software productivity. Human factors contribute decisively to increase productivity.
- **Build Integrity In**
  - *Perceived Integrity*, *Conceptual Integrity*, *Refactoring*, *Testing* – One key differentiator for developing superior products is integrity. Domain-Driven Design which is also advocated by FDD appears here as well as strong emphasis on testing and refactoring procedures. Refactoring and testing are important tips from Pragmatic Programming. When high integrity is perceived by the customer and modules or components of the system works smoothly integrated, it is a sign that good software productivity procedures have taken place.
- **See The Whole**
  - *Measurements, Contracts* – Measuring Individual Performance must be avoided. Partial Measurements that do not convey an Informational Measure Systems must be left off. A negotiable scope contract instead of a fixed one. The more one tries to control the scope, the more it expands. [25]

4.4 Pragmatic Programming tips

Pragmatic Programming has several tips regarding many development areas, i.e. project management, software configuration management, requirements engineering and programming practices itself. We concentrated on two tips for handling the requirements issue as follows:

- **Don´t Gather Requirements, Dig For Them -** Avoid the common way of gathering requirements, which is just collecting them without thinking in order to verify whether those same requirements should be considered for implementation or left off.
- **Use a Requirements Template –** By using a formal template, you can be sure that you include all the information you need in a use case: performance characteristics, other involved parties, priority, frequency, and various errors and exceptions that can crop up ("nonfunctional requirements") [7].

5. A Simple Framework

Our simple framework comprises well-know process disciplines: Requirements, Analysis & Design, Build or Implementation, Tests and Project Tracking. Each line from a discipline means an already explained practice (section 4) which can stem from FDD, LSD and PP, or a combination of these.

- *Requirements*
    - Don´t Gather Requirements, Dig For Them (PP)
    - Eliminate Waste by avoiding non-client valued functions (LSD)
    - Use Requirements Templates (PP)
    - Don’t take a first-depth approach (LSD)
- *Analysis & Design*
    - Domain Object Modeling (FDD)
    - Build a Features List(FDD)
    - Plan By Feature (FDD)
    - Class Ownership (FDD and LSD)
    - Design By Feature (FDD).
- *Build or Implementation*
    - Build By Feature(FDD)
    - Regular Builds(FDD)
    - Build Integrity In
        - Refactoring (LSD and PP)
- *Tests*
    - Build Integrity In
        - Inspection (FDD and LSD)
- *Project Tracking*
    - Use Automated Tools for Progress Reporting (FDD)

6. Conclusion and Future Works

We believe FDD strengths combined with LSD and PP successful practices can augment software productivity levels. Nevertheless, we must consider that this is more valid for projects where high technically skilled personnel is at hand and there is some degree of experience in tailoring up their process according to the characteristics of the project involved. Human factors play a determinant role for increasing software productivity. One interesting aspect is that we suggest a framework where not all process steps are explicitly determined. This is because some activities such as “*Don´t Gather Requirements, Dig For Them*” or “*Eliminate Waste By Avoiding non-client valued functions*” are abstract and rely on personnel skills. We intend to develop a tool to implement progress tracking and development team allocation in an automatic way, making it easier to implement FDD processes. Using this tool, we expect to conduct an experiment where we can instantiate a process shaped by this framework and measure productivity using a largely adopted methodology to measure *ratio input*, such as Function Points and compare to other similar projects run on a traditional process basis in order to confirm the existence of a significant raise on software productivity between the two approaches. A questionnaire will be created and make available to the customer as well as development team and executive senior management to identify the most important parts that they noticed it could higher software productivity levels.

7. References

[1] Palmer, S. R. and Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development*. Upper Saddle River, NJ, Prentice-Hall.

[2] http://www.agilemodeling.com/essays/fdd.htm, Feature Driven Development (FDD) and Agile Modeling. Scott W. Ambler

[3] http://www.nebulon.com/articles/fdd/latestprocesses.html, The Latest FDD Processes, Nebulon Pty Ltd.

[4] P. Abraharnsson, O. Salo, J. Ronkainen, and J. Warsta, *Agile software development methods: Review and Analysis*. Espoo, Finland: Technical 252. Research Centre of Finland, VTT Publications 478, Available online: http://www.inf.vtt.fffpdf/publications/2002/P478.pdf

[5] Poppendieck, M. Lean Development & the Predictability Paradox, Poppendieck LLC, 2003. Available online: http://www.poppendieck.com/pdfs/Predictability_Paradox.pdf.

[6] Poppendieck, M. Lean Software Development, C++ Magazine Methodology Issue (Publication Fall 2003). Available online: http://www.poppendieck.com/pdfs/Lean_Software_Development.pdf.

[7] A. Hunt, Thomas, D., *The Pragmatic Programmer:* Addison-Wesley, 2000

[8] http://www.pragmaticprogrammer.com/courses/ppoverview.html, Briefings and Presentations: Pragmatic Programming Overview.

[9] B. Boehm, Software Engineering Economics, Prentice Hall PTR, October 1981

[10] Barry W. Boehm and Philip N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering,* October 1988.

[11] Ching, C., Making More Money: An Introduction to Agile Development, XPDay, November 2005. Available online: http://www.clarkeching.com/2005/11/index.html

[12] Salo, O., Abrahamsson, P. Empirical Evaluation of Agile Software Development: The Controlled Case Study Approach, VTT Technical Research Centre of Finland, Finland

[13] P. Abrahamsson, J. Warsta, Mikko T. Siponen and J. Ronkainen. New Directions on Agile Methods: A Comparative Analysis. Technical Research Centre of Finland, VTT Electronics.

[14] Boehm/Turner, Balancing Agility and Discipline: A Guide for the Perplexed, Addison Wesley, Boston, 2003.

[15] Capability Maturity Model Integration (CMMI) Overview, Software Engineering Institute, Carnegie Mellon. Available online: http://www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview05.pdf

[16] Clark, B.K. ; Quantifying the effects of process improvement on effort , IEEE Volume 17, Issue 6, Nov.-Dec. 2000 Page(s):65 - 70 Digital Object Identifier 10.1109/52.895170

[17] Walt S. (1994). Understanding Software Productivity, University of Southern California, Los Angeles, USA.

[18] Jones, C. Measuring Programming Quality and Productivity, IBM Systems Journal, 1978.

[19] Felsing M., The Hidden Cost of Application Development: Using Process as a Productivity Tool, Processexchange, Inc., 2003. Available online: http://www.processexchange.com

[20] DeMarco, T. and Lister, T. *Peopleware: Productive Projects and Teams*, 2nd Ed., Dorset House Publishing.

[21] Coad P., Lefebyre E., De Luca, J. Java Modeling In Color With UML: Enterprise Components And Processes. Prentice Hall.

[22] Mary P. and Tom P., Lean Software Development: An Agile Toolkit, Addison-Wesley Professional, 2003

[23] J. Johnson, Features and Function Usage, Standish Group International, 2002.

[24] http://www.strategosinc.com/value_stream_mapping1.htm, Article & Guide to Value Stream Mapping (VSM).

[25] Christoph S. (2004), Lean Software Development, Institute Central Region 2004, Herrenberg, Germany. IBM Corporation.