The software architecture role in agile methodologies

Lic. Valerio Adrián Anacleto
Departamento de Computación, Facultad de Ciencias Exactas y Naturales, UBA,
Planta Baja, Pabellón 1, Ciudad Universitaria, (1428) Buenos Aires, Argentina
Also Epidata Consulting S.R.L. - Maipú 521, Buenos Aires, Argentina
adrian@epidataconsulting.com

Abstract

The software architecture role in agile methodologies is not sufficiently documented or formalized by means of a process consistent with the philosophy. The current work's contributions are: 1) Defining guidelines to implement architecture practices in an agile way and so that they can be applied independently of the development process in use. 2) Setting the grounds for the definition of an agile architecture process.

Key words: software architecture, agile methodologies

1 INTRODUCTION

Since the very beginning, agile methodologies have achieved a great amount of followers. Nowadays, a considerable number of projects are being developed following an agile methodology of some sort and a very big proportion of the ones that do not use a formal agile methodology follow, at least, some guidelines and ideas taken from it. Even methodologies that are considered "less agile", such as RUP, put into practice new approximations to make them look like "more agile" methodologies; several examples and references can be found in [17].

Within this context, software architecture as design practice does not seem to be treated with the care it deserves. The proof is the little amount of information available about this practice in the documentation of the different agile methodologies.

Another influential factor is the lack of standardized documentation; at present, a standard ADL (Architecture Description Language) to define architectures does not exist. This implies that a standard formality – as the diagrams for UML classes – does not exist to document architectures either. At this moment, UML has not yet decided which of the many ADL existing languages will be the one included in future versions of UML.

We believe that not having into account software architecture as a process by itself within any software development process generates more risky projects with duties and assignments distributed in an inappropriate way and, above everything else, it is shown by the final product quality.

The current work is organized in the following way: first, we briefly introduce the agile methodologies; next, we present a definition and introduction to software architecture and study the state of the art of this discipline within agile methodologies; then, we mention some software architecture practices that can be used alongside any methodology; and finally, we make some comments about the contributions, future work and conclusions.

2 AGILE METHODOLOGIES

2.1 What is Agile Software Development?

By the end of the nineties, several methodologies began to attract public attention in general. Each one of these methodologies was a combination of old ideas, new ideas and variations of old ideas. But all of them had in common: that they emphasized the collaboration between programmer teams and business experts; that they promoted direct face-to-face communication (as a more efficient way of communicating than through written documentation); frequent releases of new business functionality; self-organized development teams; and ways to structure source codes and development teams so that the most important requirements did not fall into crisis.

By the beginning of 2001, the pioneers and creators of these methodologies gathered everything their methodologies had in common in a unique figure. By using the term "Agile" as a definition that represented them all, they created what they became to call the Manifesto for Agile Software Development [1]. Some of the most important principles defined in this manifesto are: (1) Personal interaction of individuals over processes and tools. (2) Working with the code over written documentation. (3) Customer collaboration over contract negotiation. (4) Responding to change over following and sticking to a plan.

The term "agile" does not describe a particular methodology; instead, it represents the existence of a family of methodologies that share the foundations and guidelines, which include the ones listed before. As examples of agile methodologies, we can mention Scrum, Feature Driven Development, DSDM, Crystal, XP, among many others. They can be found in [16, 15, 14, 12 y 6] respectively.

2.2 Design techniques in agile methodologies

Many of the critiques that agile methodologies currently receive are related to the apparent lack of practices regarding the formal design of the application; or, in the case of methodologies that take design into consideration, its lack of formality in its specification [13].

Some agile methodology supporters justify this lack of formality with the following reasoning: any detailed design involves a considerable expense of time in decisions and aspects that are uncertain until the precise moment they appear [6]. In other words, we could say that traditionally design tried to anticipate change but, the more we try to anticipate change, the more time we will spend in the design, and the more classes there will be in the model; which is translated as: a more complex model to keep updated, with more code lines, more work and, what is even worse, the probability that, when an unforeseen change comes about, even more work will have to be done than if changes had not been taken into account in the design.

Because of that, many of the agile methodologies propose to create an informal design, on paper if necessary. The design has to be done in the simplest possible way. This kind of practice is a part of what in agile methodologies is known under the heading: "to do everything possible to do the least possible" [5].

2.3 The code is NOT the design

Due to the strong critiques received because of the lack of formal design, some interesting counterproposals that stand for the agile methodologies' position have been written down. Most answers revolve around the following suggestion made by Fowler, which can be found in [17]. His suggestion states the following: "So is design dead? Not by any means, but the nature of design has changed. Agile design looks for the following skills: (1) A constant desire to keep code as clear and simple as possible. (2) Refactoring skills so you can confidently make improvements whenever you see the need. (3) A good knowledge of patterns: not just the solutions but also appreciating when to use them and how to evolve into them. (4) Knowing how to communicate the design to the people who need to understand it, using code, diagrams and above all: conversation."

Another justification a little less formal but anyway of widespread approval is the following: "the source code is the design" [19][20] [21]. Here, the argument is the following: the source code can be seen as the final and visible result of the design, in which all of its aspects are expressed. This idea, which can be found in [21], is taken to its extremes by some agile methodologies, where design does not exist as a separate development stage or as a generator of some formal artifact. It is just considered something informal, useful to convey ideas among developers. The justification is that, with a constant refactoring, good design is gradually "found".

These ideas define the opinion about design of many agile methodologies and, above all, of many agile methodology supporters. Although these opinions are highly controversial, it is interesting to keep their evolving approach against the predictive posture that design has historically had.

2.4 Grasping and taking ideas

The code is not the design because it only shows us a view of it (the view of the code) while there is, among other things, the dynamic. There are also several models, for instance: the one of components, of analysis, of use cases, etc.

Saying that the source code is the design is like saying that a finished building is its design. There are things that cannot be seen. The synergy generated by the parts as a whole prevents people from seeing certain aspects (views and models); the whole makes vision blunt due to the lack of abstraction, just as the wing of a bird in movement does not make people understand why the bird can fly. Its inner structure or skeleton cannot be analyzed: we need to detach from the feathered being that flies to analyze its design; the aerodynamic of its feathers cannot be understood; the distribution of its muscles cannot be seen; the circulatory apparatus configuration cannot be grasped at all. That is to say that we do not have views. Something similar occurs with the source code. Despite the fact that we can see the inner structure of the code and extract information, we cannot see the inner structure of the components it uses, and besides, it is not practical to extract information from the structure. Abstraction is necessary to understand the whole step by step and in an effective way.

Moreover, if we add up the communication factor with the project stakeholders, an issue that will be discussed later, communication gets even more complex instead of being simplified. It is complex to tell a manager to look at the bird in the sky to understand why and how the bird flies.

Despite the extreme positions and, in some cases, extremist and even simple-minded positions

among which we can find, on the one hand, agile methodologies stating that the source code is the design and, on the other hand, the traditionalists proclaiming that design should consider change we can keep some ideas, which will be taken into account through the rest of this work. These ideas are: not to design so much for the purpose of change and do take into account what is necessary to design, when and in which level of detail; to accept that the requirements change and to handle the situation in the most agile way, always well-disposed to change without reticence; to use the iterative and incremental approach, about which references can be found since 1977 (see [10]).

3 SOFTWARE ARCHITECTURE PRACTICES, DESIGN AND AGILE METHODOLOGIES

3.1 Definition of architecture

We will take the following definition of architecture: “The software architecture of a program or a system is the structure or structures of the system, and it comprises the software components, the visible external properties of these elements and their relationships” [11]. We consider that, in practice, it is the most useful of the several definitions of architecture that can be found today.

The definition of software architecture in an early stage of a development process will provide, among others, the following benefits: [11] (1) work break down: separating tasks and assigning them to work teams (this is mainly due to the fact that, when the main components of the application have been identified, work teams can be assigned to them); (2) making estimation easier; (3) looking ahead: to minimize risks; (4) dealing with non functional requirements.

3.2 The double role of architecture in a software development

From our point of view, architecture fulfills a double role. On the one hand, relying on architecture is useful for the developer or development team engaged in a module as context and reference. On the other hand, architecture defines the basic guidelines of the project by providing a solid framework, allowing the interchange of professionals among work teams and defining work standards, without a high overload of bureaucracy. At the same time, what could be considered a bureaucracy overload should be put down in documents meant for the project board. These documents are required by the board in most cases. In this work, the suggestion is to write these documents in the appropriate stages of the project and taking benefit by relying on strong (strong does not imply inflexible or non-agile) architectural guidelines along the whole development.

By taking into account the architectural guidelines, the inner design of the component can be handled in an agile way by leaving the formal specification aside (if the complexity and/or the context of the particular component requires it) until the moment it becomes necessary.

3.3 Architecture and its current role within agile methodologies

It is difficult to find resources on how to define an architecture in a software development project led by an agile methodology. The available documentation in many cases just includes minimal guidelines, as in the metaphor notion of XP [6].

Originally, architecture was not taken into account in agile methodologies and if it was, it was just

in a tangential and superficial way. For example, in the book by Kent Beck [6], only two pages are devoted to architecture. As time went by, the architecture role as an agent capable of anticipating and establishing order started to emerge in several agile methodology works but, in many cases, with some disdain and with a quite different notion than the current profile of a software architect.

3.4 Every decision in software engineering implies a trade-off

The agile methodology approaches can be inclined to set a little trap. This trap consists of speeding up the project development in such a way that the project itself is in the end a kind of continuous refactoring, in which its control and its estimation become chaotic. We can illustrate this by means of a simple analogy: let us imagine an algorithm that works by brute force. Its logic operates the following way: the algorithm is going to move forward until it finds a solution. It checks whether the solution is the optimal one; if it is not, it continues looking for the next solution. Nonetheless, there are other algorithms that even though they also use brute force, they provide context (local) information and offer a forward-looking approach, so that they can anticipate and achieve the desired objective and consequently, they can speed up the search. For instance, by taking into account the local information, the algorithm will be able to realize in the middle of the process that the chosen path is not the one it is looking for; therefore, it will start following a new path (solution) before reaching the end of the first one.

In those cases, if too much context information is taken pretending to know too much about the future, the fact of having that information might make the algorithm even slower than the brute-force algorithm.

In the metaphor mentioned above, the path the project is supposed to develop is the path that has to be discovered by the algorithm. In fact, in both cases, the path is unknown, since we will not know until the end if the decisions taken were the appropriate ones. That is the basis of a considerable portion of the methodology suppositions: both the agile and the traditional ones.

The algorithm is the methodology that has to be implemented. The algorithm made up by only brute force is an agile methodology incorrectly applied: it does not think or look beyond; it just has into account the local information (concerning the module of a reduced work team). The algorithm that tries to see too far into the future is the typical over-specification where there is an inclination to change documentation in a continuous and repetitive way because the very same documentation is repeated.

The brute force algorithm with the proper heuristic is the one that uses global and local information and that foresees the future, at least in short term. Beyond doubt, the proper heuristic depends on the project. Interesting ideas concerning methodologies adaptation according to the type of project can be found in [12]. This topic is related to the need of providing a context for software engineering works based on project taxonomies [9].

By means of the use of architecture, we are looking to define a heuristic, which should be adapted according to the project taxonomy, that allows to speed up even more the methodology in question and that provides all the benefits previously specified regarding the use of architecture practices in a software project.

4 RECOMMENDATIONS TO SPEED UP SOFTWARE ARCHITECTURE PRACTICES

In a background where software development is inclined towards formalization and automation of engineering processes that provide the security and capability of prediction of other disciplines like civil engineering, software architecture can and must perform a fundamental role. It would be of great importance to count with a software architecture process capable of merging with any software development methodology, in which the benefits of being applied in each stage and the consequences can be clearly identified.

In the following passages, we mention some guidelines and useful practices that can be used in any software development process, regardless the chosen methodology. Some of these guidelines were taken from other authors, in which cases there will be a source reference. The rest are the product of our own experience or are practices in existence, recommended to be applied in any project since they are considered to be highly effective regarding their impact on speeding up processes, which determines the advantage of using them.

4.1 Do not make the role hierarchical

Being an architect is just a role. This role can be fulfilled by anyone with the necessary skills. It is of utmost importance not to give a divine conception to the role. Whoever fulfills it is not in a higher position in the organization chart than the rest of the team members.

4.2 Only one architect to define the architecture

Architecture must be the product of just one architect or a reduced group with a well-defined leader [11]. Macro architecture, with a more abstract conception, has to be done, if possible, by only one architect. This provides an agile component by avoiding long discussions. Regarding the architect, it is his duty to share, delegate and inform less abstract matters and he can lean on other development team members to take decisions. This role can be fulfilled by any architect of the team. It is important to underline that: less abstraction dos not imply, under any point of view at all, less significance. It is also interesting to highlight that, within an organization engaged in software development, the architect role can be fulfilled by any member provided that this member is qualified, and it can be quite interesting to change these roles in different projects [8].

4.3 Quality requirements

Architects must have a list of functional requirements for the system and a list of the priorities of quality attributes (such as modularity, changeability, etc.) [11]. Quality attributes can be categorized and should not be limited just to the quality of the final product, but also to the requirements of the quality of design, code, testing, performance, etc.

4.4 Using Software Architecture Analysis Method

Using the Software Architecture Analysis Method (SAAM) [11] to analyze different candidate architectures is an extremely simple method and it is very useful to communicate ideas. It can be used instead of more complex methods like ATAM [22].

4.5 Treating quality requirements as risks

Many quality requirements, especially those related to performance, usability, load, availability, etc. can be treated as risks. That is to say that the fact that one of them is not accomplished implies a risk. At the same time, these requirements can be categorized and administered in the same way than a risk, by using a quite simple form in which the impact, its mitigation cost, the probability, etc. are pointed out.

This form can be reproduced from any risk administration form and it should have and extra column to specify the chosen architecture. Different architectures will have different effects on quality requirements and these requirements have to be taken into account. This kind of form will be useful to run a SAAM test on the different possible architectures.

4.6 Using the risk categorization from the SEI

The SEI (Software Engineering Institute) offers a risk categorization concerning computer science projects, which can be found in [3]. By using this categorization, some overlooked risks can be discovered, but it can also be used to identify quality requirements. For instance, in the taxonomy we can find “availability”. The fact that the application does not fulfill the availability requirements the client expects can be seen as a risk. Identifying which is the quality requirement the client expects and writing it formally is a quality requirement. Likewise, other quality requirements can be identified concerning: interfaces, performance, testability, environment, schedule and staff, among others.

4.7 Keeping a proper administration of the project risks

There are simple tasks, some can be found in [2]. There are also many guidelines within the various agile methodologies. We believe that one of the best resources about risk administration every architect should read is in [4]. Even though implementing the whole group of recommendations can be not agile at all, bearing in mind their existence may prevent some headaches.

4.8 Keeping aside the temptation of clairvoyance (Solve the current problem)

Do not try to predict the future. Make only the architecture of what you are sure will not change, at least in a short term. Nowadays, the creation of application architectures is evolving into service specification and the creation of transverse architectures (SOA). This brings about the creation of transverse architectures for services and that have an evolving nature. Trying to predict the future of the company and how its business will be in a regular term is not the job of the architect or the developers. However, dealing with things by means of the appropriate adaptation of architecture is.

4.9 Keeping aside the myth of common sense

Usually, software engineering decisions are justified by means of the halo of common sense. Although common sense is necessary in everyday life, for the physicist it can be a matter of common sense to say that E = MC2, but for most of the mortals it is not. The question is that understanding why something develops in a particular way does not make it a matter of common sense. Similarly, it does not belong to common sense that the creation of a pattern or a project estimation is not a matter of common sense (if you forgive the repetition). In fact, since

programming began, it took years to arrive at the notion of pattern and achieve serious estimation methods; and this does not mean that the software engineers that preceded these concepts lacked common sense. The declarations that are usually attributed to common sense are based on preconceived knowledge and every statement and judgment is based on this previously acquired knowledge [23].

4.10 Seeing architecture as a service

Architecture can be seen as a service that supports business processes. Following this conception, architecture should support the business and should be able to evolve with it. From this point of view, the evolving character of an architecture or lack of it can be seen as a quality requirement.

4.11 Building transverse architecture teams

It could be good practice to build architecture teams with people from different modules, who work part time doing architecture tasks and sharing their opinions about the architecture. This approach provides the opportunity of letting the whole development team know the architecture better, making everyone involved in it and improving the architecture because of the feedback of the ones involved [8].

4.12 Speaking Patterns

Use the patterns as part of everyday language at work, at least the best-known ones. Nowadays, it is difficult to see a developer who has not read a book about patterns. Communication is a main factor within a team that aims to use an agile methodology. It is important to be able to raise the abstraction level and know that every member of the development team understands when someone says: “this is a composite”, “a delegate” or a “DAO”. Some design decisions can oppose quality attributes of the design and of the application [7].

4.13 Using Attribute-Based Architectural Style

An Attribute-Based Architectural Style (ABAS) [24] can be thought of as a pre-analyzed part of an architecture. ABAS is made up by: (1) an Architectural Pattern (Style); (2) the description of the software components and their relationships; (3) quality attributes concerning these components and the way they connect with each other; (4) an analytic framework that allows to reason about the quality attributes. The use of ABAS allows using again parts of already defined architectures. Moreover, it allows a certain, verified and measurable approximation to the quality attributes related with the architectural style. It is feasible to think that the correct use of architectural styles will speed up the decision-taking process about architectures and make it easier. The use of ABAS will also make communication easier, just like “speaking patterns”.

4.14 Creating Tracer Bullets

In most cases where there is a user interface, it usually turns to be quite useful to build a prototype. This prototype is generally used to validate the requirements with the user. Now, in many methodologies, the prototype is cast away after the requirements have been validated. Here we propose to use an approximation as the one defined in [2], in which the prototype called “tracer bullet” serves to polish the requirements, but it should also be used to validate the architecture and

run tests. The “tracer bullet” is part of the final application. It is not discarded and it evolves until the final form of the application is reached.

4.15 Defining a highly professional level language

Using a highly professional level language by means of common knowledge of the techniques and tools used by the team members. This practice speeds up the communication process among the different parts. It is true that there is a trade-off decision between the language and the required learning of a new team member. This learning can be considered an investment in a regular term.

4.16 Sharing difficult decisions

Counting on an architect who shares difficult decisions with the architecture team members and even with the rest of the team whenever the decision is too complex.

4.17 Getting involved only in the decisions of higher architectural impact

Selecting or counting on an architect who: does not deflect his attention from decisions with a high architectural impact; does not turn into a delaying factor; does not get involved in nonsense discussions with no benefits; leaves aside discussions and focuses on the implementation of tools, IDEs and work methodology, unless they have an impact on architecture; stays alert to the implications and validation of the proper use of the defined architecture.

4.18 Confidence on the team

The architect must know his team members so that he can gain the necessary confidence. This way, the architect can and must trust his team because the team, in turn, trusts him to define the best possible architecture within the context of the project. The professional trust bonds are difficult to keep if the esteem is not reciprocal.

4.19 Testing the quality requirements

Architecture must aim to keep the quality requirements. Similarly, there are also unit tests. It can turn out to be useful to define batteries of tests that validate the quality requirements. For example, the average response time, the amount of concurrent users, etc. The tests can be automated and included within the normal process of SCM.

4.20 Checkpoints and integration tests

Auditing architecture and its use constantly will produce that the new requirements are taken into account. Defining checkpoints in the integration tests can be quite useful and practical. Each instance of integration brings about the need of running the battery of architecture tests that validates and certifies quality requirements.

4.21 Do not transfer responsibilities

A professional performance in a particular role grants him rights as well as obligations. An architect role requires the obligation of not to transfer responsibilities for the decisions taken and the final

result of the architecture. If the architecture does not fulfill the specific requirements, it is the only and exclusive responsibility of the architect.

4.22 Making an understandable architecture

Architecture is also a means of communication and it serves as a link between technical and non-technical profiles by documenting architecture in a clear way. Using stereotypes [18] conveniently can be of great help.

4.23 Simple architecture

Keeping architecture the simplest as possible: If simplicity is translated as an easier way to use architecture, to understand the concepts involved, and if it is well documented, it is likely that the productivity level increases. Keeping architecture simple is another quality requirement.

4.24 Defining the requirements clearly

Defining the quality requirements clearly: architecture must be able to be measured on the basis of a well documented quality-requirement specification. The quality requirement must be testable. For example: saying “the system must be stable” regarding the quality requirement *stability* is not formal neither testable; but a quality requirement that states “the system must have 99.99 availability” is a testable requirement.

4.25 Incremental releases

Setting up incremental releases that are certified by the architecture team: each release must be architecturally valid. This means that the release must respect the current architecture. It is necessary to take into account that architecture will most probably evolve and change because it is set within the frames of an agile environment, but it will always have to respect the quality requirements.

4.26 Improving your skills constantly

It is a good exercise to imagine for a moment that our profession is not related to software development but to medicine. If we had to perform a surgical operation: which methods, techniques and tools should we use? Should we subject a patient to a surgical operation that will leave a scar for the rest of his life if he can be treated with some modern technique, like laser? If you were the patient, would you prefer to be operated by an updated surgeon? Or would it be the same?

Software development, just like medicine, is a profession based on updated knowledge. The skills in both areas should be measured by the amount of knowledge and its proper use. Therefore, it is feasible to believe that, by means of learning constantly and recognizing our weaknesses to improve, we will be able to fulfill our task in a better way.

Some of the good personal qualities that should be looked forward in an agile software architect would be: (1) Knowing the dominion of the problem to be solved; (2) Being a good communicator (which implies: being a good listener); (3) Knowing how to persuade (provided one is right); (4)

Having project manager skills but being able to avoid getting entangled with these tasks in everyday work; (5) Thinking that one can always improve in the technical as well as human aspects [23].

5 CONTRIBUTIONS

In this work, we have shown the need to implement software architecture traditional practices from an agile approach in projects that work with any kind of methodology. We have presented some useful guidelines for the implementation of architecture practices in agile way. It is not the object of this work to discuss how these guidelines have to be applied or in which stage of the process. In this work, the bases are set for the development of an agile architecture process that is able to be implemented within the context of any methodology and/or any development process taken from practice.

6 FUTURE WORKS

As future work, we would like to create an agile software architecture process that can be attached to any kind of development process in use. Many of the tips mentioned in this work can be expanded, showing examples of their use within the particular context of a proper problem domain.

7 CONCLUSIONS

In this work, we have shown the need to define a software architecture process that can mingle with any existing development methodology. We have also discussed the need for this process to be agile in such a way that it can be implemented in agile as well as traditional processes. As a first approximation to the resolution of this problem, we have mentioned some guidelines that should be taken into account whenever an architecture has to be made for any software development project.

Some guidelines have been compiled from the current bibliography concerning the topic in question – in which cases the corresponding reference number is shown within the same paragraph – and others are original contributions of the present work. Each one of the suggested guidelines is based either on accepted industry standards or on works of recognized academic level. The originality of the proposal lies not only in its use for speeding up an architectural process, but also in the several guidelines that have been introduced.

REFERENCES

[1] Manifesto for Agile Software Development: http://www.agilemanifesto.org/

[2] Andrew Hunt, David Thomas. The pragmatic programmer, Addison-Wesley Professional; 1st edition (October 20, 1999)

[3] Brian Gallagher. Taxonomy of operational risk. http://www.sei.cmu.edu/risk/taxonomy.pdf

[4] Risk Management at SEI: http://www.sei.cmu.edu/risk/main.html

[5] Fowler M. Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997

[6] Beck, Kent. Extreme Programming explained. Reading, Mass: Addison-Wesley Longman,

Inc. 2000.

[7] Valerio Adrián Anacleto. Che Loco, alcánzame un Coso ó sobre ambigüedades cosométricas, 2006 http://www.epidataconsulting.com/tikiwiki/tiki-read_article.php?articleId=16

[8] Valerio Adrián Anacleto. Asignación caótica de profesionales a proyectos. 2006

[9] Valerio Adrián Anacleto. La carencia de taxonomías en la ingeniería de software, 2006 http://www.epidataconsulting.com/tikiwiki/tiki-read_article.php?articleId=2

[10] Jacobson, Ivar; Booch, Grady; Rumbaugh, James. The Unified Software Development Process. Addision-Wesley, 1999.

[11] Bass, Len. Clements, Paul. Kazman, Rick. Software Architecture in Practice. Addison Wesley. 1998.

[12] Alistair Cockburn. Crystal Clear : A Human-Powered Methodology for Small Teams, Addison-Wesley Professional (October 19, 2004)

[13] Christine Hofmeister, Robert Nord, Dilip Soni, Applied Software Architecture. Addison-Wesley . 1st edition 1999

[14] DSDM Consortium, Jennifer Stapleton. DSDM: Business Focused Development, 2nd Edition, 2002

[15] Stephen R Palmer, John M. Felsing. A Practical Guide to Feature-Driven Development (The Coad Series). clearPrentice Hall PTR; 1st edition (February 11, 2002)

[16] Ken Schwaber, Mike Beedle. Agile Software Development with SCRUM, Prentice Hall; 1st edition (October 15, 2001)

[17] Martin Fowler; ,The New Methodology, 2005 http://www.martinfowler.com/articles/newMethodology.html

[18] Dan Pilone y Neil Pitman, UML 2.0 in a Nutshell - First Edition June 2005

[19] The Source Code Is The Design:http://c2.com/cgi/wiki?TheSourceCodeIsTheDesign

[20] Jack W. Reeves. Code as Desgin, 1992 http://www.developerdotstar.com/mag/articles/reeves_design_main.html

[21] Malan, Ruth, and Dana Bredemeyer, Less is More with Minimalist Architecture, IEEE's IT Professional, September/October 2002.

[22] Rick Kazman et all. Steps in an Architecture Trade off Analysis Method: Quality Attribute Models and Analysis, Technical Report, CMU/SEI-97-TR-029

[23] Eliyahu M Goldratt. The Goal: A Process of Ongoing Improvement 2nd edition

[24] Mark Klein ,Rick Kazman. Attribute-Based Architectural Styles, Technical Report, /SEI-99-TR-022: http://www.sei.cmu.edu/publications/documents/99.reports/99tr022/99tr022abstract.html