MODERN METHODS OF SOFTWARE DEVELOPMENT

DAWID ZIMA

Department of Computer Systems Architecture
Gdansk University of Technology Narutowicza 11/12, 80-233 Gdansk, Poland

(received: 10 June 2015; revised: 10 July 2015;
accepted: 20 July 2015; published online: 1 October 2015)

Abstract: Software development methods consist of such activities like analysis, planning, development, testing, deployment, maintenance and retirement. All of them can be divided into two main categories – traditional and agile. The objective of this paper is to review some of the most popular traditional, agile and open source development methods. Special attention was paid to the common stages of all methods – testing and maintenance.

Keywords: Software development method, Agile, Open Source, Testing, Maintenance, Waterfall, Scrum

1. Introduction

The development of software – regardless of its kind (firmware running on a microprocessor, an operating system, a hardware driver, a video game, an enterprise application solving business problems) – is a very complex task. It requires communication with customers, defining tasks and relations between them, making predictions, *etc.* – in other words – a plan for developing the software. The software development method (or in other words: the software development model or the software development life cycle) is a definition of an abstract process used for the development of software, including activities like: requirement analysis, coding, testing, maintenance, *etc.* A most general model is presented in Figure 1.

Figure 1. General software development model

There are many existing software development methods like the Waterfall, Scrum, Extreme Programming or Spiral model. All of them differ from one other, have their own advantages, disadvantages and scopes of use [1, 2]. For example

some of them are too complex for small projects or emphasize risk management more than others (which might be crucial for some projects). But all of them have something in common: they are used for software development and can be divided into two categories: traditional (considered as "heavyweight") and agile ("lightweight").

The purpose of this paper is to describe some of the most common software development methods and their categories (traditional, agile) for the Centre of Competence named Novel Infrastructure of Workable Applications for the development of applications and services in different areas. Special attention was paid to the Open Source Software development methods (their nature enforces some restrictions that are not seen in a closed-source projects) and to the common stages of all software development methods: testing and maintenance.

2. Traditional software development methods

Traditional software development methods (often called "heavyweight") are the earliest and still commonly used models in many organizations [3]. All of them implement a similar pattern: they consist of sequential (sometimes iterative) well defined stages. They do not emphasize frequent communication with customers and good response to the requirement change – rather well defined use-cases at the beginning of the project. This predictive approach in traditional methods based on the detail requirement analysis results in very expensive changes in client requirements after they are defined (*i.e.* due to market changes or exploration of new use-cases during development). The lack of an adaptive approach (which is common in all agile methods) makes traditional models not suitable for small projects and companies or for projects where requirements are not well defined or are frequently changing.

The Waterfall software development model is considered as traditional and one of the oldest software engineering models. The first formal description of this method was introduced in 1970 by Winston W. Royce [4]. It consists of 5 high level, non-overlapping, sequential stages as shown in Figure 2. This model is linear, which means that each of the phases must be completed before the next one begins. With its linear nature, this model is considered as inflexible and "heavy" due to the separation of requirements analysis and implementation, which excludes a swift response to requirement changes or exploratory development. When the problem is discovered in stage n it is required to step back to stage $n-1$ to make appropriate changes and go through next stages once again. It means that errors committed in early stages are the most expensive ones. Due to its linear nature, the Waterfall model can be applied to projects where all the requirements are known, clear, well defined and the product definition is stable (no changes after definition). Each phase has a clear definition of inputs and outputs, hence, this model is well manageable.

Another example of the traditional development method is the Iterative model [5] where the project is divided into smaller segments being developed in

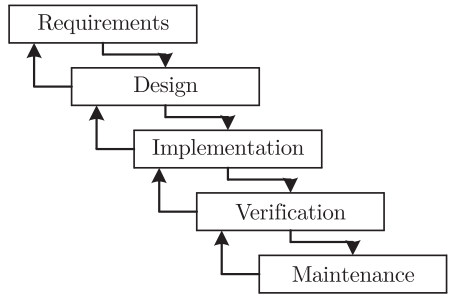

Figure 2. Waterfall Model diagram

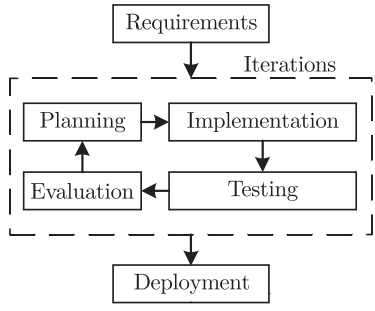

Figure 3. Iterative Model diagram

multiple iterations. Each iteration is similar to the mini-waterfall model. With this approach, software with limited features (incomplete implementation) can be delivered after each iteration which helps to mitigate problems with integration, explore potential issues / new requirements in early phases (iterations) and allows incrementing changes to be monitored. The Spiral Model [6] is a risk-driven software development method with an iterative approach. The main focus on risk assessment is designed to enhance risk avoidance. This model has two main distinguishing features: (1) a "cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk" and (2) a "set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions".

3. Agile software development methods

Agile in terms of software development means rapid and flexible response to change. In February 2001, 17 software developers met together to discuss some of the existing software development methods and published the *Manifesto for Agile Software Development* [7]: "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value: (1) Individuals and interactions over processes and tools, (2) Working software over comprehensive documentation, (3) Customer collaboration over contract negotiation, (4) Responding to change over following a plan. That is,

while there is value in the items on the right, we value the items on the left more."

Furthermore, twelve principles explaining what it is to be Agile has been published [7, 8]: "(1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. (2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. (3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. (4) Business people and developers must work together daily throughout the project. (5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. (6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. (7) Working software is the primary measure of progress. (8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. (9) Continuous attention to technical excellence and good design enhances agility. (10) Simplicity – the art of maximizing the amount of work not done – is essential. (11) The best architectures, requirements, and designs emerge from self-organizing teams. (12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."

Despite four core values from the *Agile Manifesto*, some researchers [9] have determined key agile values. Their research method was based on comments analysis of the agile manifesto signatories from 2005 to 2011. They have found and described top ten values: (1) flexibility, (2) customer-centric, (3) working software, (4) collaboration, (5) simplicity, (6) communication, (7) natural, (8) learning, (9) pragmatism and (10) adaptability.

According to the *Forrester* report [3] agile methods have become mainstream of development approaches. The results of the survey conducted by D. West *et al.* [3] show that 35% of the respondents have stated that Agile most closely reflects their development process (in which Scrum is the most common chose – 10.9%). However, traditional (and "heavier") methods are still present in many organizations (21% for the iterative development and 13% for the waterfall model). The results of the survey conducted in *Nokia* show that "agile methods are here to stay" [10] (60% of respondents would not like to return to the methods used before agile transformation).

There are various existing methods considered as agile, *i.e.*: Extreme Programming, Scrum, Crystal family of methodologies, Feature Driven Development, the Rational Unified Process, Dynamic Systems Development Method, Adaptive Software Development, Agile Modeling. Most of theses methods have been well described and compared by P. Abrahamsson *et al.* [2]. They have also published a set of rules helping answer the question: what makes a development method an agile one. The method must be incremental, cooperative, straightforward and adaptive [2].

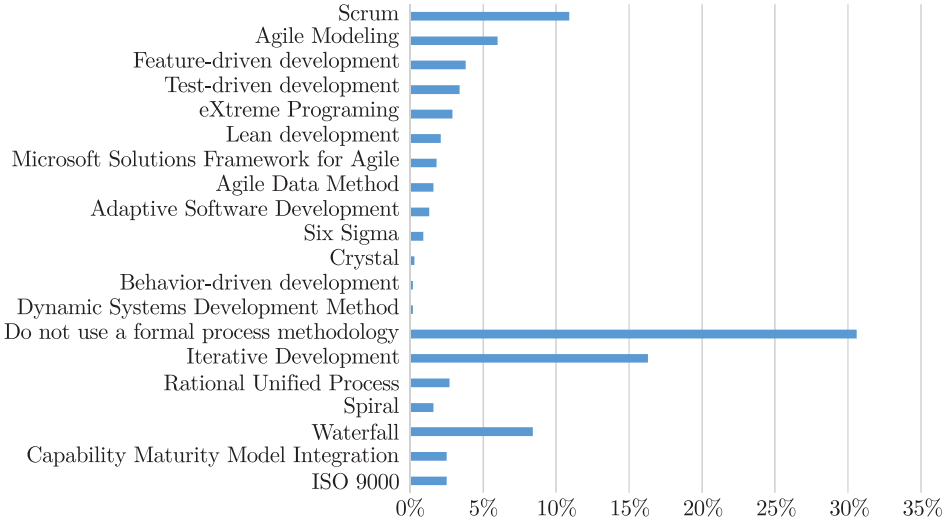

Figure 4. Forrester survey results [9]

As previously stated, according to the *Forrester* report [3], Scrum is the most common choice of the development method from all the agile techniques – 10.9% of the developers have stated that Scrum most closely reflects the development process that they are currently using (all agile methods together have reached 35%).

Scrum is an iterative and incremental process framework for software development [11]. The Scrum Team consists of three different roles: the Product Owner (responsible for managing the Product Backlog), the Development Team (3–9 self-organizing, cross-functional developers creating Increments of the Product after each Sprint) and a Scrum Master (responsible for ensuring Scrum rules are followed).

The centre point of a Scrum model is Sprint. Sprint is a time-box of one month or less (usually two weeks) after which Increment of the product is created. Each Sprint consist of: Sprint Planning (creating Sprint backlog based on Product Backlog), Daily Scrums (daily, 15 minutes long meetings for developers to synchronize their work), development work, Sprint Review (inspection of Increment at the end of the Sprint), Sprint Retrospective (opportunity for the Scrum Team to inspect the Scrum process and improve next Sprints).

Product Backlog is a centralized point where all the requirements for the product are stored. It is managed by the Product Owner. It consists of all features, fixes, requirements of the product represented as Product Backlog Items with: description, order, estimate and value. The Product Backlog may evolve as the time goes on – *i.e.* new requirements may be defined or more accurate estimations may be made. The Sprint Backlog consists of selected Product Backlog items and is created during the Sprint Planning phase of each Sprint.

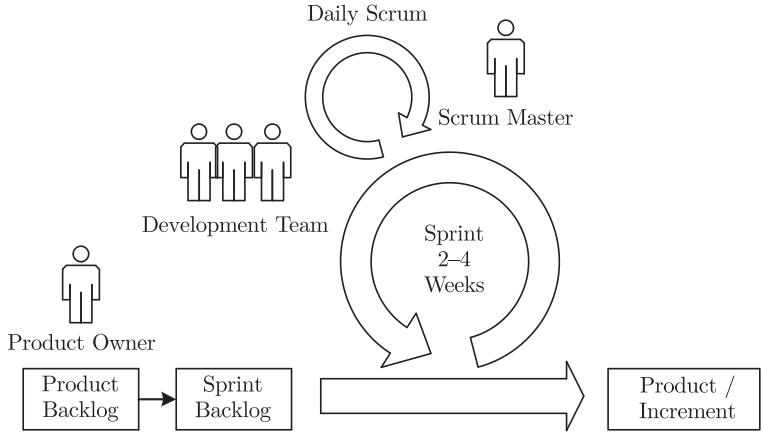

Figure 5. Scrum Process

4. Open source software development methods

Open Source Software (OSS) is software the source code of which is publically available and anyone can make a change to it. According to the official definition of OSS by the *Open Source Initiative* organization [12] it is not only the public availability of the source code that makes software open source, but it is also the software license that must comply with the set of rules such as free redistribution, *etc.* Examples of popular OSS licenses are: Apache License 2.0, GNU General Public License (GPL), GNU Library or "Lesser" General Public License (LGPL), MIT license. Some of the most popular OSS includes: Linux Kernel, the Android operating system, Apache Web Server or MySql – the world's most popular open source database.

As previously mentioned, the source code of OSS is publicly available and anyone can make a change into it. It means that a large community of volunteers may be involved into the OSS development process. This is the main difference between open source software and closed source software – classical software development methods cannot be used for OSS development. However, S. Koch [13] in his researches has found many similarities in OSS development to agile software development principles (close collaboration with users, frequent release of the working code, self-organizing teams). There are many existing OSS development models – some of them have been well described and compared with each other by M. Saini *et al.* [14].

E. S. Raymond [15] on the basis of his observations of the *fetchmail* and *Linux Kernel* development process has divided the OSS development into two main models: The Cathedral and the Bazaar. In the Cathedral model the software source code is available after each version has been released – before the release source code is known only to developers and the project is developed using "closed-style" methods ("I believed that the most important software (...) needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time.").

In the Bazaar model, software is developed by the community *i.e.* via Internet ("No quiet, reverent cathedral-building here – rather, (…) a great babbling bazaar of differing agendas and approaches (…) out of which a coherent and stable system could seemingly emerge only by a succession of miracles.").

A. Capiluppi and M. Michlmayr [16] have proposed a model that combines the above mentioned Cathedral and Bazaar models (the main assumption is that both the models are not mutually exclusive). The project is started in a Cathedral way, and when an appropriate level of maturity is reached, the software development method transits to the Bazaar model. They emphasize that the transition must be commenced at the right time to attract volunteers (community) – when the prototype is functional and still there is some work to be done. If the project is not stable or is too advanced (*i.e.* all features have been implemented) potential contributors may not want to be involved. A. Capiluppi *et al.* have analyzed two open source projects: *Arla* and *Wine*. Adding new modules to the *Wine* project by the core development team helped to succeed transition to the Bazaar model. In *Arla* transition it failed because the number of new modules was decreasing. This shows that fresh developers from the community tend to work on new functionalities rather than on old modules. To successfully transit an open source project from the Cathedral to the Bazaar model, core developers have to attract new developers by adding new modules and provide new directions for the project.

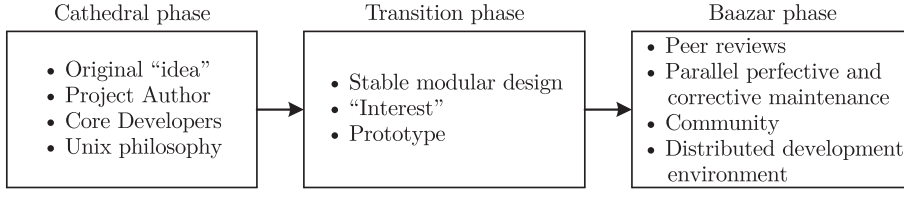

Figure 6. Transition from the Cathedral to the Baazar model [16]

Another example of an OSS development model is the one used in the *FreeBSD* operating system proposed by N. Jørgensen [17]. This model does not define how the code is written, the requirements are gathered or how it is designed – it is hidden in the "Code" stage and left for the developers. It rather shows how the change is integrated into the project after having been developed. The Jørgensen model consists of six sequential phases that applies to each code change: Code, Review, Pre-commit Test, Development Release, Parallel Debugging and Production Release. After the code is written, reviewed and tested (*i.e.* to ensure that the build is not broken) in the first three stages, it is committed to the development branch (fourth stage) and then inspected by other involved developers to find bugs ("Parallel Debugging" stage). When the code is ready (there are no other contraindications), then it is merged from the development branch to the production branch.

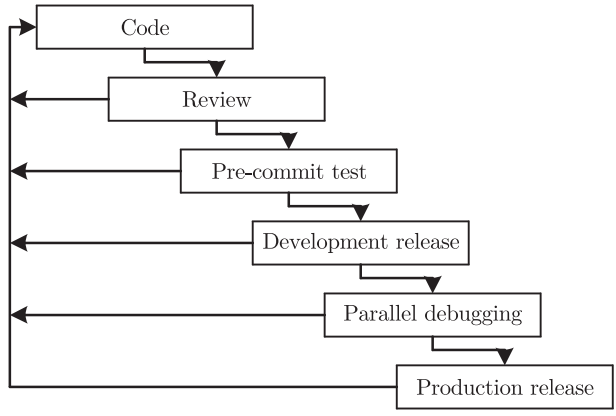

Figure 7. N. Jørgensen FreeBSD development model [17]

5. Software testing

The main purpose of testing is to detect errors [18]. The testing process is not able to confirm that the system works well in all conditions, but is able to show that it will not work under certain conditions. The testing may also verify whether the tested software behaves in accordance with the specified requirements applied by developers during the design and implementation phases. It also provides information about the quality of the product and its condition. Having such knowledge can help to address a variety of business decisions, *i.e.* whether to continue a further project or the release date of the new version of the software is at risk and so on. Testing can be implemented at any time of the software development life cycle, however, the testing model is associated with the adopted methodology of software development.

All the software testing methods can be divided into three main categories: black-box, gray-box and white-box testing [18]. In the black-box testing method software is examined without having any knowledge about internal implementation or algorithms – so the software is treated as a black-box that transforms the provided input into the expected output (validation of functionality). Commonly this method is applied to higher level of tests (*i.e.* acceptance tests). White-box testing aims to validate the internal mechanism of the application, so it assumes having knowledge about low level implementation details. A good example of a white-box test are unit tests (validation of the application flow at a unit level) or integration tests (validation of the application flow between units). Gray-box testing is a combination of white and black boxes.

According to the *SWEBOK* (Software Engineering Body of Knowledge) guide [19] the testing levels can be distinguished by the object of testing (Unit Testing, Integration Testing, System Testing) or the purpose of testing (Acceptance Testing, Installation Testing, Alpha and Beta Testing, Regression Testing, Performance Testing, Security Testing, Stress Testing, Back-To-Back Testing, Re-

covery Testing, Interface Testing, Usability and Human Computer Interaction Testing).

As mentioned previously, testing can be implemented at any time of the software development life cycle, however, there are some development models that have strong focus on testing. One of such methods is TDD [20] (Test-Driven Development, classified as Agile). The idea of the TDD is simple: write a test (which will initially fail), then write code (implementing the desired functionality) to make test pass, refactor, repeat until all functionalities are implemented.

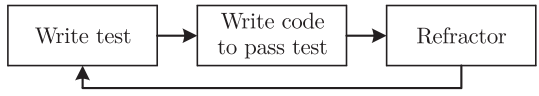

Figure 8. TDD diagram

Another example of a software development method emphasizing tests is BDD (Behavior-Driven Development) proposed by Dan North [21]. It may be considered as an extended version of TDD where, instead of testing units, each system behavior is examined. In this technique having well specified requirements is crucial (in a formalized way, *i.e.* using the *Gherkin* syntax and the *Given-When-Then* pattern). When all system use-cases are specified, before implementing them, a written acceptance test should be performed to cover each functionality (similar to TDD).

6. Software maintenance

A software product has to be maintained after release (delivered to customer) – faults and errors have to be fixed, performance has to be tuned over the time, the product has to be adapted to changes of the environment (*i.e.* API changes in external systems that the developed product has to cooperate with) or customer requirements. Maintenance by the IEEE Standard has been defined as: "the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment" [22]. The software maintenance phase is very often considered as the most expensive and time consuming stage.

According to the research of B. P. Lientz and E. B. Swanson [23] all maintenance activities may be categorized into four main categories (shown in Figure 9): adaptive (changes related to the evolving environment), perfective (functional changes of software after release, performance tunings), corrective (fixing errors and faults), preventive (activities for better software maintainability in the future).

The IEEE standard 1219–1998 [24] defines the Maintenance Process which consists of activities and tasks necessary to modify an existing software product while preserving its integrity (shown in Figure 10). The Maintenance Process Implementation is a phase in which the maintainer develops and documents all strategies and procedures that will be used during the software lifecycle

maintenance stage. After this phase there are three subsequent activities called iteratively when there is a need for modification. Each modification request is analyzed (*i.e.* its impact on organization or other systems), implemented, tested, verified, reviewed and approved.

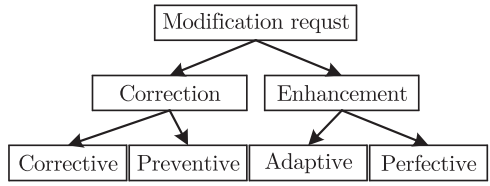

Figure 9. Modification Request

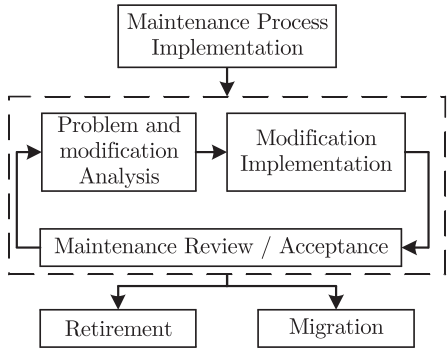

Figure 10. Maintenance Process

Some researchers (*i.e.* T. Stalhane *et al.* [25]), on the basis of literature review and their own studies, have identified several cost drivers for corrective maintenance: the size of the system to be maintained, the complexity of the system, the phase where a defect is discovered, the maintainers' experience with the maintained system and the application domain of the system, tool and process support. They have also prepared recommendations for the industry for reducing the corrective maintenance effort by: classifying defects into orthogonal categories, analyzing how frequently defects occur for each category and analyzing cost drivers and productivity of corrective maintenance for each category.

S. Velmourougan *et al.* [26] prepared a software development life cycle model to improve maintainability called M-SDLC (Maintainability-Software Development Life Cycle). They have proved in their studies that adoption of best maintenance practices during each phase of the software development lifecycle can decrease the number of failures (and needed maintenance effort) after software delivery.

7. Conclusion

All software development methods may be divided into two categories: traditional (older and "heavier") and agile (new approach, "lighter"). The described models are presented and compared in Table 1. During the last few years, after

Table 1. Comparison of described software development methods

Method Name	Classification	Type	Short Description
Waterfall	Traditional	Linear	• 5 high level, non-overlapping, sequential stages • each of the phases must be completed before the next one begins • when the problem is discovered in stage n it is required to step back to the stage $n-1$
Iterative development	Traditional	Incremental	• project is divided into smaller segments being developed in multiple iterations • iteration is similar to the mini-waterfall model • software with limited features can be delivered after each iteration
Spiral Model	Traditional	Incremental	• risk-driven software development method • main focus on risk assessment
Scrum	Agile	Incremental	• product is created in Sprints (2–4 weeks time-box) • Sprint consist of: Planning, Daily Scrums, development work, Review and Retrospective • Product Backlog stores all client requirements
The Cathedral model	Open Source	Incremental	• software source code is available after each release • before release software is developed using a "closed-style" • possible transition to the Bazaar model
The Bazaar model	Open Source	Incremental	• software developed by the community (*i.e.* via Internet) • "(…) a great babbling bazaar of differing agendas and approaches (…) out of which a coherent and stable system could seemingly emerge only by a succession of miracles"
FreeBSD development model	Open Source	Incremental	• does not define how the code is written, requirements are gathered or how it is designed • six sequential phases that applies to each code change • "Parallel Debugging" stage

formalizing what agile methods are in 2001, a trend of growing popularity of agile methods has been seen in many organizations [3]. The results of the survey conducted in *Nokia* [10] on more than 1000 respondents from 7 countries (in Europe, North America and Asia) during the period of transformation from non-agile to agile appears to confirm the statement that this is not a temporary fascination but a strong, growing trend becoming mainstream. Most of the respondents agree

on the generally claimed benefits of agile methods (higher satisfaction, a feeling of effectiveness, improved quality and transparency) and 60% would not like to return to the previous methods. However, classical methods (*i.e.* Waterfall or iterative model) are still popular in many companies [3]. Agile has been the subject of many researches in a past decade. T. Dingsøyr *et al.* [27] have examined many publications and citations about agile to see the progress of researches that was made since the Agile Manifesto was introduced in 2001.

Choosing the right software development model for a project might be essential and have great impact thereon (positive or negative). There is no universal model for all kind of software projects, each of them has their own advantages, disadvantages and scope of use. The Extended Decision Support Matrix proposed by P. M. Khan *et al.* [28] may be helpful in choosing the right model.

In the Centre of Competence named Novel Infrastructure of Workable Applications (NIWA) operated at the Gdańsk University of Technology the preferable software development methods are: agile and open source. The Scrum approach is recommended to improve some platform components of NIWA and the *Redmine* tool is largely used to design new user applications. In case of virtual teams a Bazar approach is sometimes taken into consideration. However, for developing many work flow applications a new software developing approach has been created. It bases on the composition of the available IT services and is called *SOSE* (Service Oriented Software Engineering). It could be considered in another paper.

References

[1] Munassar N M A and Govardhan A 2010 *Int. J. Comp. Sci. Issues* **7** (5)
[2] Abrahamsson P, Salo O, Ronkainen J and Warsta J 2002 *Agile Software Development Methods: Review and Analysis*, VTT Publications, **478**
[3] West D, Grant T, Gerush M and D'Silva D 2010 *Agile Development: Mainstream Adoption Has Changed Agility*, Forrester Research
[4] Royce W W 1970 *proc. IEEE WESCON* **26** (8)
[5] Larman C and Basili V R 2003 *Computer* **36** (6) 47
[6] Boehm B 2000 *Spiral development: Experience, principles and refinements*, CMU's Software Engineering Institute
[7] Beck K, Beedle M, Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin R C, Mellor S, Schwaber K, Sutherland J and Thomas D 2014 *Manifesto for Agile Software Development*, http://agilemanifesto.org/ (online)
[8] Fowler M and Highsmith J 2001 *Software Development* **9** (8) 28
[9] Madi T, Dahalin Z and Baharom F 2011 *Content analysis on agile values: A perception from software practitioners*, *5th Malaysian Conference in Software Engineering*
[10] Laanti M, Salo O and Abrahamsson P 2011 *Information and Software Technology* **53** (3) 276
[11] Schwaber K and Sutherland J *Scrum Guide*, http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf (online)
[12] Open Source Initiative 2015, Available: http://opensource.org/ (online)

[13] Koch S 2004 *Agile Principles and Open Source Software Development: A Theoretical and Empirical Discussion*, (in: Extreme Programming and Agile Processes in Software Engineering), Springer 85
[14] Saini M and Kaur K 2014 *International Journal of Software Engineering and Its Applications* **8** (3) 417
[15] Raymond E S 2015 *The Cathedral and the Bazaar*, http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html (online)
[16] Capiluppi A and Michlmayr M 2007 *IFIP* **234** 31
[17] Jørgensen N 2001 *Information Systems Journal* **11** (4) 321
[18] Myers G J and Sandler C 2004 *The Art of Software Testing*, John Wiley & Sons
[19] Bourque P and F R E 2014 *Guide to the Software Engineering Body of Knowledge, Version 3.0*, IEEE Computer Society
[20] Beck K 2003 *Test-driven Development: By Example*, Addison-Wesley Professional
[21] North D 2015 *Introducing BDD*, http://dannorth.net/introducing-bdd/ (online)
[22] I.S. 610.12 1990 *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Computer Society Press
[23] Lientz B P and Swanson E B 1980 *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*, Addison-Wesley
[24] 1219-1998 – IEEE Standard for Software Maintenance 1998, IEEE Computer Society
[25] Li J, Stalhane T, Kristiansen J M W and Conradi R 2010 *26^{th} IEEE International Conference on Software Maintenance*, Timisoara
[26] Velmourougan S, Dhavachelvan P, Baskaran R and Ravikumar B 2014 *Software Development Life Cycle Model to Improve Maintainability of Software Applications*, *4^{th} ICACC*
[27] Dingsoyr T, Nerur S, Balijepally V and Moe N 2012 *Journal of Systems and Software* **85** (6) 1213
[28] Khan P M and Beg M M S 2013 *Extended Decision Support Matrix for Selection of SDLC-Models on Traditional and Agile Software Development Projects*, Third International Conference on Advanced Computing & Communication Technologies