Structured software development with the IPIDDDT method

A lean method for small agile organizations

UNIVERSITY OF TURKU
Department of Information Technology
Master of Science in Technology Thesis
September 2010
Jori Ahvonen

Inspectors:
Ville Leppänen
Tuomas Mäkilä

UNIVERSITY OF TURKU
Department of Information Technology

AHVONEN, JORI: Structured software development with the IPIDDDT method - A lean method for small agile organizations

Master's Thesis, 76 pages
Software Development
September 2010

A software development process is a predetermined sequence of steps to create a piece of software. A software development process is used, so that an implementing organization could gain significant benefits. The benefits for software development companies, that can be attributed to software process improvement efforts, are improved predictability in the development effort and improved quality software products. The implementation, maintenance, and management of a software process as well as the software process improvement efforts are expensive. Especially the implementation phase is expensive with a best case scenario of a slow return on investment. Software processes are rare in very small software development companies because of the cost of implementation and an improbable return on investment.

This study presents a new method to enable benefits that are usually related to software process improvement to small companies with a low cost. The study presents reasons for the development of the method, a description of the method, and an implementation process for the method, as well as a theoretical case study of a method implementation.

The study's focus is on describing the method. The theoretical use case is used to illustrate the theory of the method and the implementation process of the method. The study ends with a few conclusions on the method and on the method's implementation process. The main conclusion is that the method requires further study as well as implementation experiments to asses the value of the method.

Keywords: Software Engineering, Software Tools

TURUN YLIOPISTO
Informaatioteknologian laitos

AHVONEN, JORI: Rakennetta ohjelmistokehitykseen IPIDDDT menetelmällä – kevyt menetelmä pienille organisaatioille

Diplomityö, 76s.
Ohjelmistotekniikka
Syyskuu 2010

Ohjelmistokehitysprosessi on ohjelmistojen tuottamiseen käytettävä suunnitelma, jolla pyritään saavuttamaan merkittäviä etuja ohjelmistokehitykseen. Tavoiteltavia etuja ovat mm. ohjelmistokehityksen ennakoitavuus ja tuotettavien ohjelmistojen laadun paraneminen. Ohjelmistokehitysprosessin käyttöönottaminen, ylläpitäminen ja hallitseminen vaatii merkittäviä resursseja. Etenkin ohjelmistoprosessin käyttöönotto vaatii useimmiten omien käytäntöjen muuttamisen lisäksi merkittäviä panostuksia mm. ohjelmistokehitystyökaluihin. Ohjelmistokehitysprosessin ja jatkuvan prosessinparannuksen käyttäminen pienissä organisaatioissa on, edellä mainittujen kustannusten takia, harvinaista.

Tämän tutkielman tavoitteena on esitellä uusi menetelmä, jolla ohjelmistokehitysprosesseihin liitettäviä hyötyjä pyritään saavuttamaan, ilman ohjelmistoprosessien hallitsemiseen liittyviä kustannuksia. Työssä esitellään syyt menetelmän kehittämiselle, menetelmän kuvaus ja menetelmän käyttöönottoprosessi, sekä teoreettinen tapaustutkimus käyttöönotolle.

Työ keskittyy menetelmän kuvaamiseen ja käyttää teoreettista käyttöönottoa menetelmän ja käyttöönottoprosessin käytännönläheisempään kuvaamiseen. Tutkielma arvioi työn lopuksi menetelmän toimivuutta, mutta menetelmän todellisia hyötyjä ei voida arvioida ilman jatkotutkimuksia.

Asiasanat: ohjelmistokehitys, ohjelmistotuotanto, ohjelmistot

Table of Contents

Index of Tables

1 Introduction

Small scale software development in small software development companies, working with small and versatile projects, are a challenging group of companies for software development process improvement. The biggest challenge for software development process improvement is that small companies have smaller resources for software process improvement than large companies. Small software development companies need special types of software process improvement efforts, special types of software development process methodologies, or modified software process improvement targets. Popular ways to address these challenges are using agile software development methodologies or modifying more rigid software process targets like the CMM key process areas to better suite a small organization [Coleman Dangle et al., 2005]. The smaller the company, the harder a beneficial process is to maintain.

Small scale software development is often done without an explicit software development process because the engineering and enactment of a software development process is considered too expensive without actual return on investment. This study presents another approach to software development and software development processes. The approach is to enable process benefits for a certain type of small companies with a small overhead, without an actual software process engineering and improvement effort. This thesis studies and presents a method on how software development tools should be chosen and modified to enable and enforce software process practices. Enabling and enforcing some software development process practices can bring benefits that are typical for software process improvement, to small co-located companies with small resources, that could not sustain a proper software process improvement effort.

This thesis begins by presenting some background and theory in the first Chapters 2 - 5. Chapter 2 presents theory on basic software development concepts that are used later in the thesis. The emphasis is, in the beginning, on software development processes in Section 2.2, and later, in Section 2.4, on software development tools and software development tool selection. Background on the software process methodologies' views on software development tools is presented in Chapter 3. A simple software tool taxonomy is presented in Chapter 4. The software tool taxonomy relies on the IEEE Standard for Developing Software Life Cycle Processes [IEEE std. 1074-1997] as well as on agile principles to create a common ground for discussions in the later parts of the thesis. The last chapter of the background methods is Chapter 5, which presents a simple tool selection process that is used repeatedly in the following chapters.

The background for the new software process management method is presented in Chapter 6. Chapter 6 discusses software process management models and problems associated with them. The new process management method called Implicit Process Improvement with Day-to-Day Development Tools (IPIDDDT) is presented in Chapter 7. Chapter 7 consists of a presentation of the IPIDDDT model as well as the implementation process of IPIDDDT.

The IPIDDDT method is implemented for a theoretical company in Chapter 8. This theoretical case study is done to present the IPIDDDT method implementation in a more concrete form. The method is implemented according to the IPIDDDT implementation process that was presented in Section 7.2. The company and the company's improvement targets are presented in Section 8.2. The tool selection targets a set of process benefits that are selected methodically according to process improvement benefits that have been identified in other studies, an important source for this is "Concepts on Measuring the Benefits of Software Process Improvement" [Rozum, 1993]. The final steps of the IPIDDDT selection are done in Section 8.6.

The thesis discusses, in the final Chapter 9, what impact this method could have on small scale software development, and what further studies could be done to realize the tool selection method in the field.

2 Background on software development processes and tools

2.1 Software development

The creation of software is called software development. Software development is a complicated discipline with a vast number of actions included. The ideal steps that are required to create a new piece of software have been gathered in the IEEE Standard for Developing Software Life Cycle Processes [IEEE std. 1074-1997]. The sequence in which these concepts are used is called a *software life-cycle model*. A software life-cycle model and the information what to emphasize in the software life-cycle model is called a software process methodology.   [IEEE std. 1074-1997], [Brugge and Dutoit, 2004]

> ”A software life-cycle model represents all the activities and work products necessary to develop a software system.” [Brugge and Dutoit, 2004]

A software life-cycle model is an abstraction of the process of delivering a piece of software. It includes a development process, a supporting process, and a managing process. Management of the life-cycle consists of both managing the model of the process, and its actualization as the tracking of activities, roles and work products and defining them.

Life-cycle models are general visions on how the process of developing software should be run. The first documented and discussed life-cycle model was the *waterfall model*. The waterfall model was a model directly derived from the manufacturing and construction industries. The waterfall model is an activity centered model that describes sequential execution of the following steps

1) Requirements specification

2) Analysis

3) Design

4) Implementation

5) Testing and debugging (AKA validation)

6) Installation

7) Maintenance

The steps have a simple feedback loops where a problem in one step would mean a return to the previous phase or to a phase where a solution for that problem could be found. The steps represent different sub-processes of the life-cycle process to develop and maintain a piece of software through its life-cycle. [Royce, 1970]

Several modern life-cycle models are *evolutionary*. An evolutionary life-cycle model repeats a set of steps, similar to the waterfall model, to create working software with a subset of the wanted requirements in each iteration. Evolutionary life-cycle models deliver better reaction to change and the option to end the project at a wanted stage, for instance, when the budget runs out, with working software ready for delivery. On the other side, evolutionary life-cycle models require some special development paradigms to be competitive with the more sequential models because an iterative model repeats all development process steps multiple times, while sequential models repeat them only once, or a few times.

2.2 Agile software development processes

2.2.1 The agile development process

Software process methodologies are an emphasized subset of the software life-cycle processes; collections of practices that define the action of software development projects, according to a life-cycle model. Process methodologies are a defined set of practices that try to define best practices for creating software. Software process methodologies range from some principles and points of emphasis to clearly distinguished exact steps for creating software. [Brugge and Dutoit, 2004]

Agile software development process methodologies are methodologies that emphasize the following items. These items are presented in the [Agile Manifesto, 2001] which is the basis for modern agile software development process methodologies.

“Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan” [Agile Manifesto, 2001]

Tools and processes are given less emphasis in agile development methodologies. Some practices of agile development, such as constant integration and test-driven development, require tools to enable them [Fowler, 2006 (2)]. In addition to tools that are required for agile practices on localized sites, there is a distinct need for tools that enable agile practices such as face-to-face communications when developing software in a non co-located development team. A tool that would enable such communication is for example web conferencing software. [Cockburn, 2004]

The key agile software development practices that set them apart from other practices are according to [Schwaber, 2007]:

1) Iterative development, where each iteration delivers working software

2) The phases in each iteration are nearly concurrent

3) The team uses specific practices to keep the code base fresh and flexible

4) Teams are self managing

5) Lean principles eliminate waste whenever possible.

Popular agile software development methodologies include *Scrum, Extreme Programming* and *Lean software development*. Scrum is a technique that emphasizes project management techniques like scrum of scrums and self managing teams [Abrahamsson et al. 2002]. Extreme programming is a methodology that focuses on agile development practices like test-driven development, pair programming, and continuous integration [Abrahamsson et al. 2002]. Lean software development has its emphasis on constant process development but with a very lean process. The concepts of waste and value are employed in the development of the software process [Larman and Vodde, 2009].

2.2.2 Benefits of using a software development process

The primary aim of traditional software process methodologies is to find a repeatable, predictable process that improves productivity and quality. Agile software processes have deemed such an aim to be impossible and this can be testified especially because of the great versatility of software development, where completely different products can not be produced in the exact same way. Another agile critique of the traditional process aims resides in the fact that software development has inherent instability, the requirements often change in mid-development.

The benefits that are sought through software process improvement are reducing waste, increasing productivity, and rising quality. Organizations that have a stable process (mature organizations) that continuously improve their process reap rewards in superior quality products and increased customer and employee satisfaction. [Rozum, 1993]

Measurable examples of process development benefits are for example more accurate project prediction, better return on investment, reduced time to delivery, fewer required staff resources or reduced cost of rework. The emphasis on these different goals are selected according to the target organization and the used software process. [Rozum, 1993]

Software process methodology differences are evident also in the process benefits that the process aims to attain. An example of the differences on project improvement goals is the value of project prediction. A project prediction is achieved in process methodologies like the *capability maturity model* (CMM) style processes through the thorough documentation of all phases of the process [Byrnes and Phillips, 1996]. A relatively good account of the phases of projects is achieved after documenting a few projects. Resource consumption of different parts of the project can be approximated based on previous projects. The CMM solution can sometimes be a successful solution for software development in organizations where changes mid-development are rare and the project type is very homogenous [Shaikh et al., 2009]. [Coleman Dangle et al., 2005]

A complete process prediction is not possible, nor desirable in Scrum, because the requirements and process predictions change and become more exact throughout the development effort. Scrum is iterative and focuses on delivering working software after each iteration. Iterative development means that the teachings of this exact process are available in measuring development velocity, and the finished functionality can be accurately approximated for the next iteration. Each iteration delivers functionality according to a prioritized list, a list that can be altered between each iteration. The Scrum solution is very effective for versatile projects that have requirements that can change during production, and a limited budget. [Abrahamsson et al. 2002]

2.2.3 Implementation of a software process methodology

A software process methodology implementation begins with the selection of a software process methodology, which is tailored to suit the requirements of the company. The most important questions of the software process tailoring are the basic questions that a methodology strives to answer: “How much planning?”, “How much reuse?” “How much modeling?” and “How much process?”. [Brugge and Dutoit, 2004]

The software process methodology is implemented by the use of a process development process. The process development process includes engineering a software development process according to the tailored software process methodology, enacting the software development process in the company, and initiating software process improvement efforts. The process development process also includes the sub-process of gathering metrics on the process. The engineering of the software process includes authoring a software process documentation that is used for the enactment of the software development process. [Feiler and Humphrey, 1993]

The software process is often reviewed based on the metrics that can be gathered from the software process implementation and application life-cycle management. The software process can be improved according to the reviewed data. A distinct software process improvement process is often set in place in organizations, where the maintenance and improvement of the software process is essential. [Feiler and Humphrey, 1993]

The actual management of the software creation process is called Application Life-cycle Management. The life-cycle management is based on the software process methodology that has been tailored for the company.

2.3 Application life-cycle management

Application life-cycle management is a discipline for coordinating the activities of creating software. Application life-cycle management can be defined as:

> ”The coordination of development life-cycle activities, including requirements, modeling, development, build, and testing, through; 1) enforcement of processes that span these activities; 2) management of relationships between development artifacts used or produced by these activities; and 3) reporting on progress of the development effort as a whole.” [Schwaber, 2006]

Traditionally application life-cycle management creates management overhead, or then the discipline is maintained on a level that is very unobtrusive but also limits the view inside the projects. The application of the life-cycle management discipline can be simplified by integrating different tools. The dedicated application life-cycle management tools strive to automate the integration of tools to simplify and automate all processes of the application life cycle management. Complete application life-cycle management has only become possible through the development of more complex management tools for software development.

2.4 Software development tools

2.4.1 The aim of software development tools

Software development tools are an essential part of software development. Software development tools strive to simplify and automate the work of delivering software. Some tools are necessary parts of software development, for instance, a compiler, while other tools are less compulsory and try to manage a process of the software life-cycle, for instance, a requirements management tool. The number of different software development tasks that are automated by software development tools is growing constantly.

The reliance on software development tools is a defining factor in software development. The reliance affects the software development effort by making it more efficient, but it also imposes some external constraints and processes on the development effort. To accomplish a certain task with a certain tool, requires a work flow that suits the tool in question. Another constraint is the skill required to use a certain tool. Tools can not be switched at will because to truly master a work flow and a software development tool requires both time and skills.

2.4.2 Categorization of software development tools

Software development tools can be classified in different ways. Software development tools are traditionally classified in commercial categorizations for software development tools. The categorizations are usually not very specific, in relation to the features or functionality of the software because the organization is very difficult, partly because features of different tools can place them in multiple categories or in no categories, and because new tools evolve frequently and older tools wither away. In addition several tools exist in the public domain, where the classification is not necessary for commerce.

Software development tools can be classified according to the functionality present in the tools but it can also be classified according to what problems it strives to solve. Comprehensive systems of classification include very comprehensive attempts, for instance, the “A guide to the classification and assessment of software engineering tools“ [Firth et al., 1987].

2.4.3 Selecting software development tools

The processes of selecting software development tools are versatile. Most methodical tool selection processes are based on large companies and large company needs, or they are simply too complex for simple selections. An emphasis is often on tool supplier reliability, creation of a tool selection implementation plan, and other activities that are not directly related to the actual tools, but the organizations that supply, implement, and use the tools.

Other methods of selecting a software development tools are selections that do not focus on organizations. The focus on organizations can be a burden especially in cases where organizations are relatively small and agile. Other cases that are better of with a very simple selection methods that focus on features and maintainability are open source tools. The cost effect in open source tools is mostly in the skill that is required to maintain the software as well as the skill required by the users.

2.5 Definition of small scale software development

The software development industry is an industry where the stakeholders, stakeholders resources and projects sizes and types vary to a very large degree. One end of the spectrum is the enterprise companies that produce enterprise software. Enterprise software products are developed in many years and the resources that are put into the development effort are very large. The other end of the spectrum presents single developers that produce small pieces of software for a few users. The resource consumption in these very small projects is very small. This thesis concentrates on small scale software development.

Small scale software development is defined in this study as software development that requires less than two months of work for a single developer. Small scale software development is identified by a small total workload and few roles. A third defining factor is that the roles are generally handled by a single person, if the project requires an architect, then usually only one person has the role of an architect, and the one person creates all architectural artifacts.

Such small scale software development, as described above, signifies a few things for the projects. Firstly, a small total workload means that all activities, other than actual development, should be kept to a minimum because all development margins are small. Secondly, the fact that roles map to single person, or a few persons, and the development artifacts are created mostly by single individuals, reduces the need for communications patterns. Thirdly, because of project size and deployment size, it is not customary to have rigorous testing or need for extensive deployment models. Finally project sizes can equal the smallest typical iteration size (three weeks), and are done in one iteration.

3 Agile software process methodologies' relationship to software development tools

Software development tools exist to enable or enhance the software development process in some way. Software development process methodologies have different views on the role and use of software development tools. This chapter will discuss the role of software development tools in general, the role of software development in agile software development, and some specific concepts of lean software development, that can be applied to software development tool selection.

3.1 Agile software development's view on tools

3.1.1 Agile software development and tools

Agile software development processes emphasize people over tools, even though the right tools are crucial for many agile concepts, such as unit testing tools for test driven development, and build tools for continuous integration [Barnett and Schwaber, 2004]. It is typical for the agile tool set that the entire tool set integrate seamlessly and that the tool set is easy and lightweight to use.

> “... Agile shops focus their investments on tools that the entire team will use, layering Agile project management tools on top of testing tools on top of build management tools on top of software configuration management tools. Agile teams also tend to value lightweight tools more than non-Agile shops do, looking for tools that have very narrowly focused feature sets and open pathways for integration.” [Schwaber, 2007]

The key agile software development practices that set them apart from other practices are according to [Schwaber, 2007], are the same reasons that effect the selection of software development tools:

1) Iterative development, where each iteration delivers working software

2) The phases in each iteration are nearly concurrent

3) The team uses specific practices to keep the code base fresh and flexible

4) Teams are self managing

5) Lean principles eliminate waste whenever possible

Each iteration, in iterative software development, contain most phases of the basic waterfall model. The overall length of the development project has pressure to grow in iterative development, because phases like testing and installation are done more than once, and always for a new subset of the code as well as the integrated code base from the previous iteration. Some practices and automatizations should be implemented, to make individual iterations more effective. For instance, an extensive practice, that includes automatizations and makes phases of development more concurrent, is *continuous integration.* Continuous integration is a software development practice where members of a development team submit their changes to the source code to a common software configuration management system. A constant integration server automatically creates a new build from the changes to source code and runs all tests on that software. Constant integration integrates the changes of all developers and tests existing and new functionality constantly. Implied in this practice are the following software development tools: constant integration server, software configuration management, build tool, functional testing tools, and unit testing tools. [Fowler, 2006 (2)]

Agile practices to keep the code base fresh and flexible are, for example, test-driven development, code re-factoring and peer code review. These practices require at least the inclusion of software configuration management as well as unit testing tools and possibly peer code review tools. [Schwaber, 2007]

Special requirements are imposed on tools by the fact that the development teams are self managing and concurrently work with all phases of the software development project. The tools must be simple to use, or at least familiar to all developers, so that they can be used by the developers themselves instead of just specialized users. The tools should also be automated and integrated to each other, so that the use of tools will not create waste. [Schwaber, 2007]

Documentation in all parts of the development life-cycle was one of the factors that initiated the agile manifesto. The distinct type of documenting in agile processes is described in the quote below.

> ”There are two keys to successful documentation on agile projects. The first is finding the point of "just enough" documentation. This is difficult to determine and will vary by project. Fortunately, the iterative nature of agile development allows you to experiment until you get it right. The second key to successful agile documentation is to not get attached to it or have unrealistic hopes of keeping it updated. Documentation must be created to serve a specific purpose, and after it has served that purpose you'll all probably have more important things to do than keep updating the documents. It may seem counter intuitive, but it's often better to produce fresh documentation the next time some is clearly required. A side benefit of starting over each time you need to document part of your project is that it's great incentive to keep your documentation efficient!” [Simons, 2002]

3.1.2 Agile tools in practice

A very varied set of agile tools are used in practice. The agile basic ideology strongly emphasizes that chosen tools are efficient in the operation that they provide. This section presents two different authorities' views on which software development tools are essential in agile development.

The first authority's tool category selection is presented in Table 1. The table is from an article by “Forrester Research” called “Agile Development Teams Need Tools, Too” [Barnett and Schwaber, 2004]. The table lists software development tool categories that are that are useful in agile software development. Table 1 table divides the tools, that agile software development organizations benefit from using, in the categories of “Must Have”, “Should Have”, and “Nice To Have” agile tools. This division presents the view that agile development is difficult or impossible without the tools presented under the heading “Must Have”. The heading “Should Have” includes tools that benefit development vastly but agile development is possible without these tools. Finally the “Nice To Have” agile tools benefit development in some scenarios, but the lack of these does not signify automatically a significantly worse agile development experience.

”Must Have” Agile Tools	Software Configuration Management Tools
	Unit Testing Tools
	Build Tools And Build Management Systems
”Should Have” Agile Tools	Project Management Tools
”Nice To Have” Agile Tools	Development Tools
	Automated Functional Testing Tools

Table 1: Tools for agile development according to [Barnett and Schwaber, 2004]

The “Must Have” agile tools are “Software Configuration Management Tools”, “Unit Testing Tools” and “Build Tools And Build Management Systems”. Software configuration management tools are tools that store the state of software development artifacts, like the state of a source code file, at each commit to the repository. Developing software without software configuration management is very rare in modern software development. Unit testing tools are a must have for agile development because agile development is iterative, and non-automated testing would make agile development very slow. Agile development also emphasizes constant source code refactoring. Source code that is refactored must be tested again, testing would slow down development, if the code is not already covered by unit tests. Build tools and build management systems enable the building and integration of software code automatically and running test suites on the source code automatically. The automatizing of build management practices is crucial for agile iterative development because iterative development requires short test, integration, and build cycles.

The “Should Have” agile tools are project management tools. The management of iterations and development as well as estimation of the development velocity is difficult without the use of development tools. Pen and paper, solutions based on post-it notes, and/or spreadsheets are often considered sufficient solutions for many project management needs. These tools are lacking in cases where all developers are not co-located.

“Nice To Have” agile tools are development tools and automated functional testing tools. Development tools include tools that are integrated to the IDE and enable development tasks, an example is a modeling tool. Automated functional testing tools enable and automate in the same way that the unit testing tools, but on a higher abstraction level. Functional testing tools are only “nice to have” because they do not provide the same degree of support as unit testing tools in, for instance, refactoring.

Another authority's view, on which software development tools should be used in agile software development, is presented in Table 2. The other quoted authority is Alistair Cockburn in “What the Agile Toolbox Contains” [Cockburn, 2004]. The view of software development tools in this table is very similar to the previous selection in Table 1. The most important differences are that communications tools were not in the scope of Table 1, but they have a significant place in Table 2. Another significant difference is that this table includes documenting tools, IDE, and performance profiling tools.

Scope	Usage	Exemplary Tool
Communication tools	Instant Messaging	Trillian
	Group Discussion	WikiWiki
Documenting / collaboration	Generic drawing	Dia, Impress
Project Tracking		Xplanner, VersionOne
Designing-programming tools	Configuration management / Version control	CVS
	Unit test harness	Junit
	Accept test	Fit, FitNesse
	Automated task manager	ANT
	Automated Build System	Cruise Control
	IDE	Eclipse
	Performance profiling tools	Jmeter, Jprofiler, Jprobe

Table 2: Tools for agile development according to [Cockburn, 2004]

Communication is a very important part of agile software development. The only time when a wide array of communications tools is not necessary is when the team and product owners are co-located. Communications tools like “instant messaging” and “group discussion” are essential when some stakeholders or developers are on distributed locations. Software documentation is also a form of communication, that is named in this second table. The documentation tools that are presented in the table are simple generic drawing tools for modeling and documenting issues in a concise and descriptive (agile) way.

Performance profiling tools, and an IDE are also software development tools that where not explicitly named in Table 1. Performance profiling tools are essential in software that that require a certain performance level, a performance level is often represented in the nonfunctional requirements. The testing of the performance level is necessary in the development of performance critical software, but optional in the case of nonperformance critical software. An IDE (integrated development environment) is used in most modern software development, especially more established languages have multiple development environments that automate both the use of the software development kit, as well as the use of other development tools. An IDE was probably not mentioned in the first table because there is nothing especially agile about an IDE, an IDE is a requirement for all professional software development.

3.2 Lean software development and selecting software development tools

Selecting a subset of software development tools from a more standardized set can be difficult. One methodological way is to use the basic concepts of a software process methodology and extending the use of those concepts to tool selection. This section discusses what the lean software development methodology would signify for software development tool selection.

3.2.1 Lean software development concepts

Selecting a tool set to lighten the process from a standard agile process is not trivial. Some agile methodologies, such as Scrum, do not mention any tools and the founding concepts of agile software development try to diminish the value of tools [Agile Manifesto, 2001]. Still tools are essential in software development, a developer needs at least a way to write source code and a software development kit for the target language and platform. A way to select a tool subset from a larger pool is to use concepts from lean software development and extend those concepts to tool selection.

The lean software development concepts of “Kaizen” or continuous improvement, and the balancing of “values and waste” are concepts that will be used to select tools for a more efficient software process. The principles of Kaizen are as follows from [Larman and Vodde, 2009]:

> “1. choose and practice techniques the team has agreed to try, until they are well understood - that is, master standardized work.
> 2. experiment until you find a better way
> 3. repeat forever”

What to improve during Kaizen is explained in the concepts values and waste also from [Larman and Vodde, 2009].

> “Value - The moments of action or thought creating the product that the customer is willing to pay for. “
> “Waste - All other moments or actions that do not add value but consume resources.”

From these concepts of lean development come value stream mapping. A value stream mapping is the illustration of the elements of value that travel through the system to deduce the amount of value and waste in the system that can be summed up to the value ratio of a project. Value ratio is the percent of value of the total time of development. Traditionally a very large portion of development is waste and only under 10 percent is actual value. The easier way to improve the value ratio is to cut down waste because waste can be up to 90 percent of the total development.

3.2.2 Waste in software development

The following actions are a list of non-value-adding action categories according to [Larman and Vodde, 2009].

> “ 1. Overproduction of solutions or features, or of elements ahead of the next step; duplication
> 2. Waiting, delay
> 3. Hand-off, conveyance, moving
> 4. Extra processing (includes extra processes), relearning, reinvention
> 5. Partially done work, work in progress (WIP) or design in progress (DIP)
> 6. Task switching, motion between tasks; interrupt-based multitasking

7. Defects, testing and correction after creation of the product
8. Under-realizing people’s potential and varied skill, insight, ideas, suggestions
9. Knowledge and information scatter or loss
10. Wishful thinking (for example, that plans, estimates, and specifications are ‘correct’)”

Non-value-adding categories are a learning aid to see waste in the software development process. The software process can then be improved by *Improving through removing non-value-adding actions.* All waste is not waste that must be banished but some waste is temporary necessary waste. Temporary necessary waste is for example to test some features after release (non-value-adding action 7.), if testing for those same effects would be too expensive before release. [Larman and Vodde, 2009]

3.2.3 Tool value listing

The tools and process should be assessed and optimized frequently according to the concept of Kaizen. The Table 3 of values and waste concepts for software development tools is presented below, the table is based on the concepts of Kaizen presented previously. This table elaborates on what value and waste is for tool selection.

The table presents in the value column a value that the tool can provide and in the waste column a similar waste category. The value and waste categories are based on the waste and value categories of general lean software development.

Value	**Waste**
Use of the tool must bring value to the customer	… or reduce waste
The tools must do what is needed	… but not more
The tools must be easy to learn	… and have no excess complexity
The same tools can be used for multiple tasks	… and need to be learned only once
The tools should be fast	… and reduce waiting
The tools must integrate	… so similar work will not be duplicated

Table 3: Value - waste table of lean software development

4 Software development tool categorization

Tool taxonomy is a complex discipline, which aims to group tools according to some system of ordering. Multiple ways to group tools exists, for example, by features or by the phase of the software life-cycle the tool aims to address. Existing taxonomies and categorizations where deemed too complex for the purposes of this study.

This thesis presents a simple functional organization. The functional organization aims to group tools in groups that are based on software life-cycle process activities. The groups define a set of functionality, that the tools that are placed in that group addresses is some way. The grouping is not unambiguous, so it is not a taxonomy but a grouping. This grouping is a tool to help discuss the tools by grouping them according to their main functionality. The tool groups are 1) P*roject management*, 2) C*ommunication* and 3) *Development activities*. These tool groups are based on the phases of the software life-cycle as presented in [IEEE std. 1074-1997]. The groups are formed according to an agile tooling rationale where code producing activities are emphasized.

Project management comprises of software life-cycle process activities like the tool's capability to facilitate managerial tasks that run over several projects, project activities that are employed for single projects, and activities for providing visibility to the project and process. This group comprises the functionality in [IEEE std. 1074-1997] section “A.1 Process Management Activity Groups” and section “A.3.1.3 Prioritize and Integrate Software Requirements”. The project management functional group consists of activities that can be aided by process and project management tools. [IEEE std. 1074-1997]

Communication is the functional group for all tools that enable communication and documentation, and activities that only require communication like the [IEEE std. 1074-1997] activity group “A.2.1 Concept exploration activities”. This includes most documentation except for the documentation of requirements that is a group on its own. In the [IEEE std. 1074-1997] model communication spans over the complete life-cycle. The communications activities include activities that are aided primarily by communications tools.

Development activities is a functional group that span many activities. Development activities span, in [IEEE std. 1074-1997], the groups “A.3 Development Activity Groups” and “A.4 Post Development Activity Groups” as well as sections “A.5.1 Evaluation Activities”, “A.5.2 Software Configuration Management Activities”, and “A.5.3 Documentation Development Activities”. The development activities are activities that are aided by developer tools. [IEEE std. 1074-1997]

Tools and tooling functionality are organized in the subsections that follow. The tool groups are defined so that a commercial tool category, for instance, software configuration management tools, is presented only in one functional tool group, even though the group would fit in more than one tool group.

4.1 Project management

The project management activity group is comprised of the functionality in [IEEE std. 1074-1997] section “A.1 Project Management Activity Groups” and section “A.3.1.3 Prioritize and Integrate Software Requirements”. Project management activity groups are: process authoring and management activities (later referred to as *process management*), project and process *resource management and requirements prioritization*, and project and process estimations and metrics (later *process visibility*). All activity groups mentioned, are activity groups that are employed to some extent in almost all software development, and all can be aided by software development tools. The tools that can aid in these activity groups are gathered in the subsections of this section.

Project management tools are essential because a predictable process is difficult to achieve without tools. A predictable process requires the process to be managed, without any process management, a process can hardly be said to exist. Resource management is a discipline that can be very simple, emailing to-do lists to other developers, but it can also be very complex in large organizations with varying skills and skill requirements for tasks and processes. Resource management can also incorporate limiting access and controlling the resources on a deeper level: creating traceability and accountability. Process visibility is a third very important functionality group. The improvement of the process and project planning, as well as the assessment of waste in the process is very difficult without verifiable data from previous projects, that is why all types of visibility is essential to project management. The gathering of metrics is somewhat tedious and it is always good to automate by tools when possible.

4.1.1 Process management

Process management includes the activities of selecting and managing a software life-cycle process. The management of the software life-cycle process can be described as authoring and improving the process, publishing the process, and managing the enactment of the process. Tools exist that help the process management activities, tools can span some or all process management activities. More complex authoring tools model the process on a more exact level, to enable other development tools to use the process definition, and aid in enacting the process.

Tool categories that help in process modeling are from simpler to more complex: word processors, web development tools, process authoring tools, and integrated application life-cycle management tools. Tool categories that help publish the process are web browsers, web servers and integrated application life-cycle management tools. Finally tool categories that help enacting the process are tools that enforce work flow. Work flow enforcement can be aided by tools that enable communication like task lists or application life-cycle management tools that force some work flow by requiring input at all phases in development.

4.1.2 Resource management and requirements prioritization

Activities that are included in resource management are activities that related to knowledge and control of available and required resources for the projects. A way to understand this is by using the concepts of role, artifact and task. Roles are skills that are required in some part of the project and applicable human resources are mapped to these skill sets. Artifacts are resources that are needed or created during the project like requirements documentation. Access to create, modify, or view artifacts can be limited to some roles in certain tasks, and this resource control can be extended to create accountability automatically. [Zhu et al., 2006]

Requirements prioritization is an integral part of this activity group as well. The requirements prioritization allows for the selection of wanted functionality according to available resources and the value of the said functionality.

Tool categories that enable and help in resource management and prioritization (from less to more complex) are: email, spreadsheet, file access control, project management tools, project portfolio management tools, and application life-cycle management tools. Email, spreadsheets, and file access control provide lists of required roles, available human resources, and some way to limit access to some resources. These simple tools are separate and require a lot of manual labor to use.

Project management tools, project portfolio management tools, and application life-cycle management tools provide integrated support for the activities listed previously. The process and project are input to the management tool and resource control and role information is all contained in the system. The more complex tools offer additional aid in managing multiple simultaneous projects.

4.1.3 Process visibility

Visibility activities include the gathering of many different types of metrics from both single projects and process level activities. Visibility into a single project contains project velocity, resource consumption, project size estimation, and follow-up. Visibility into single projects is essential to keep all stakeholders aware of progress and to notice discrepancies between estimates and actual effort. The other type of visibility is the visibility into the software life-cycle process. Creating metrics and possibilities to report on the success and development of the process, enable the activities of software development process improvement and enable the assessing the success of the process. Software process enactment and the value of process improvement activities are impossible to asses without visibility.

Tool categories that enable process visibility are spreadsheets, documents, time tracking tools, process management tools, and application life-cycle management tools. All these tools enable some level of tracking and reporting. The important difference is that most tools require a lot of work to deliver tangible benefits.

4.2 Communication

Communication is an integral part of software development, communication is an area that overlaps all other areas (project-, requirements-, and source code management) described in Chapter 4, as well as some activity groups like [IEEE std. 1074-1997] group “A.5.3 Documentation Development Activities”, which use only communications tools. Other communications centered activity groups are requirements exploration and authoring. Requirements exploration are the [IEEE std. 1074-1997] groups “A.2.1 Concept Exploration Activities”, “A.4.3.1 Identify software improvement needs”, and “A.4.3.2 Implement problem reporting method“ which create the basis for the requirements authoring activities in [IEEE std. 1074-1997] group “A.3.1 Requirements activities”.

Communication is separated as a distinct functionality group in this study because dedicated communication tools are used. Essential to communication is that it is efficient at the moment, stored as required, and can be found when needed. Different communications mediums are for instance instant messaging, discussions boards, wikis, real time audio communication and web conferencing. Communication is stretched across time and place, in the fact that communication tools can control versions and store communications for searching.

Documentation needs in functionality vary in different software processes. The most comprehensive solutions are communications servers where all different types of communication are stored in the same data repository and meta data connect that data to people, roles, artifacts, tasks and other communications. In the other end of the spectrum most communication is handled face-to-face and some communication is handled with specialized tools and only persisted when explicitly required. Communications tools are divided in this project to the subgroups of *Documentation, Direct communication,* and W*iki and knowledge base.*

4.2.1 Persistent documentation

Documentation is an essential way of delivering information in a structured and persistent way. Documentation includes technical documentation and development documentation as well as customer and end user documentation.

The technical documentation is often formed of requirements documentation, modeling of the software, and documentation of solutions, integrations and frameworks. Requirements documentation can be documented on many different levels and with tools ranging from small physical cards to vast structured documents. The modeling of the software is made of abstractions that enable understanding software. Models are often made of the software architecture on different levels, different architecture models might include software architecture, software integration, business infrastructure, and data architecture. The solutions are often documented to maintain an understanding of the complete software. Solutions documentations often contain contain. The documentation of integrations, integration interfaces and frameworks which can be used for different types of integration are also very important to document thoroughly.

Customer and end user documentation is documentation that is written based on the need and requirements of the end users. The technical parts often include instructions for use like installation instructions, integration instructions, and user interface instructions.

The actual software code is also a form of documentation, both the code comments as well as the actual code. Tests, test results and other metric data of the software can also be considered documentation of the software. Documentation is often a dominant form of communication from the developers through time and to the product stakeholders. The documentation created in the activities in [IEEE std. 1074-1997] group “A.3.1 Requirements activities” are a good example of persistent documentation.

Documentation is produced with tools like word processors, documentations tools, reporting tools, and communications servers.

4.2.2 On demand documentation

There is a constant influx of data to a large project, especially a dispersed project requires a large amount of data to be created, stored, and used. Some data that is created on demand is never intended as actual documentation, but is unstructured or structured only through keywords, and will be deleted when it has served its purpose.

The large majority of on demand documentation is written by developers for themselves and managed in personal files. Other on demand documentation resources are discussion boards or instant messaging logs. On demand documentation is informal, but it is still often useful for someone else in the course of development or in maintenance of the software. The on demand documents become searchable artifacts if they are transferred to communications servers or other shared documenting resources.

Tools that aid with wiki and knowledge base support are wikis and knowledge base software as well as communications servers.

4.2.3 Direct communication

Direct communication are the forms of more directed communication between human resources like meetings, phone discussions, instant messaging, and email. Direct communication can be between developers, for developer - product owner communication or communication between stake holders. Direct communications are very important and include often face-to-face communication. Direct communication is sometimes but definitely not always persisted.

Direct communication is one of the most important factors in less process driven methodologies. The more strongly the process is process driven, the more exact documentation of all information is done. Communication is the key issue in agile projects from the start of requirements and modeling to final acceptance tests. All information must be transferred between the developers and the product owners so the actual business requirements are transferred as well as possible to the final product.

Tools that help with direct communication are e-mail, instant messaging, telephone and web conference software.

4.3 Development activities

Source code management is a functional group that span most activities in [IEEE std. 1074-1997] groups “A.3 Development Activity Groups” and “A.4 Post Development Activity Groups” as well as sections “A.5.1 Evaluation Activities”, and “A.5.2 Software Configuration Management Activities”. [IEEE std. 1074-1997]

Source code management includes a wide range of activities with the common denominator that the activities revolve around the actual writing of, and management of, source code as well as the deployment of the source code. Another important factor is that the tools that are used in these activities integrate closely with the developer IDE. This section is divided in to the sub activities of *Writing source code, Deployment,* and *Quality assurance.*

4.3.1 Writing software source code

Software source code writing activities include writing software source code and managing versions of the source code (software configuration management). These activities are integral to software development and need efficient tools to be handled properly. Writing software source code is mostly done in an integrated development environment, where the text editor is integrated with at least the development kit, of the source code language, and possibly with syntax highlighting, build tools, and run time environments. Modern integrated development environments are modular and include tooling for a wide variety of development needs.

Software configuration management (or revision control) is the task of tracking and controlling changes in software. Software configuration management enables the company to securely develop software and different versions of software, while creating branches of development easily and merging changes across branches and versions. Software configuration management, to the extent described above, is impossible without software configuration management tools.

The tools for writing software source code are from more simple to more complex: text editor, version control system, source code generator, and integrated development environment.

4.3.2 Deployment

Deployment activities include the building of the source code, managing the integration of all required resources, and deploying the software to a server or installing the software. Building the software includes solving dependencies and updating all required resources to build a software program. The building of the source code can often be done using multiple different tools. The software can often be built by a development kit for the programming language or more complex building tools that enable more elaborate functionality.

All different pieces of source code and other resources that constitutes a software program can be developed independently. Independent development of artifacts means that multiple developers work on the same program. The integration of code from multiple developers can be tedious and the management of this integration is addressed by practices like continuous integration. The building of software code can be automated by the use of automated build tools that create builds at standard intervals or when changes occur. Build tools can also initiate quality assurance tasks.

Tools to aid in software deployment activities include build tools, deployment tools, repository managers, and constant integration tools.

4.3.3 Quality assurance

Quality assurance activities include a number of activities related to verifying the quality of code. Some popular quality assurance activities include testing the software on different levels, making code reviews, running source code analysis, and measuring software performance. Modern testing is done using testing frameworks for different levels of testing. The testing frameworks enable a structured way to test software and enable testing disciplines to evolve.

One of the lowest levels of testing is unit testing. Unit tests strive to verify the functionality of a single module (unit). Unit tests help to avoid degradation of code in addition to verifying the functionality of a module. Other important testing levels are acceptance tests that verify the functionality that has been described in the requirements of the software. Acceptance tests are often done in web environments by tools that test the user interface with different browsers. A third important testing level is performance testing that often tests the nonfunctional requirements like response times and numbers of concurrent users.

Other important quality assurance practices include source code analysis and review. The analysis of source code can be automated to verify that the source code has been written according to coding conventions. Many different metrics can be used to strive for better software code. Code reviews are the reviews of software code by someone who has not written the software code. Reviews can be done in multiple levels of process ranging from a very strict review according to an established process or in a very informal way asynchronously.

Tools that help in quality assurance are testing frameworks for all different levels on the software from unit tests to acceptance testing and stress testing utilities as well as code analysis tools and code review tools.

4.4 Tool functionality and categorization table

This section presents a table of the most important functionality according to the previous sections. All later references to functionality groups refer to the functionality discussed in this chapter. Table 4 is created based on this chapter.

The table contains the headings and subheadings of this chapter and an exemplary tool class that is suitable for this functionality. The table combined this information to provide a table of reference of this chapter. The table does not strive to be exhaustive of all activities in general software development but an aid for more meaningful elaboration of the concepts in the following chapters of this thesis.

Tool Capability	Tool Functionality	Exemplary tool category
Project Management	Process Management	Process engineering framework
	Resource Management and Requirements Prioritization	Project management tool
	Process Visibility	Application life-cycle tool
Communication	Persistent documentation	Enterprise content management
	On Demand Documentation	Wiki software
	Direct Communication	Instant messaging
Development Activities	Writing Software Source Code	Integrated development environment
	Deployment	Deployment tool
	Quality Assurance	Unit testing tool

Table 4: Tool functionality table

5 A software development tool selection process

The tool selections of this study will be done using a basic tool selection process. The tool selection process consists of the following steps.

1. Define goals of the tool selection effort
2. Create evaluation criterion for the tool selection
3. Define metrics for the requirements
4. Select tools for evaluation
5. Review individual tools
6. Order the tools
7. Select the best tool

The first step of defining the goals of the selection process is the most important part of the selection. The justification of the selection effort is defined in the goals of the selection, if the goals are not properly set, then the tool selection can not address the issues that have prompted the selection.

The second step of the selection process is creating evaluation criterion for the tool selection. The evaluation criterion that are selected aim to reflect the main goals as well as possible. The goals of the selection effort are often not directly usable as criteria in the realm of the tools, which is why the goals must be translated in to evaluation criterion. The evaluation criterion can be a simple list of criteria like an example set of criteria in Table 5 or a complex matrix of hierarchical criterion and criterion groups. A more general set of selection criteria is suitable for selecting many different types of software development tools, while more complex criterion are often used to compare similar tools and find subtle differences in functionality.

1. Ease of Use
2. Power
3. Robustness
4. Functionality
5. Ease of Insertion
6. Quality of Commercial support

Table 5: Example of tool evaluation criteria [Firth et al., 1987]

Defining metrics for the selection criteria is the next step in the selection process. The metrics are used to assign different relative values to the criteria that are used. The ordering of tools becomes more meaningful with criteria that is relatively quantified. The actual selection and ordering of the tools are very difficult if the criterion do not have a good set of metrics.

The next phase in the process consists of selecting an appropriate pool of tools for evaluation. The pooling of tools, in a more complex selection situation, will be managed by a selection process like the one described here. The pooling is important because evaluating all possible tools is impossible. The selection of tools to a pool is made, in less complex scenarios, using a light set of general criteria, based on the actual selection criteria for the complete evaluation. This light set of general criteria can be for instance, the cost of a selection tool, and the tool category.

The value of a tool must be determined to differentiate the tools in the selection pool. Determining the value of an individual tool is done by reviewing the tool against the selection criteria and metrics. A tool is awarded a value according to the review.

The tools can be naturally ordered after reviewing all tools in the selection pool. If the selection metrics allow for weighting different selections criteria, then the tool order can be compared with different weighting of the metrics.

6 Software process management and improvement

6.1 Process authoring, publishing, enactment, enforcement and improvement

The benefits, that are sought through software process improvement, are reducing waste, increasing productivity, and rising quality. Organizations that have a stable process (mature organizations) and that continuously improve their process can reap rewards in superior quality products and increased customer and employee satisfaction. Measurable examples of process development benefits are for example more accurate project prediction, better return on investment, reduced time to delivery, fewer required staff resources or reduced cost of rework. The emphasis on these different goals are selected according to the target organization and the used software process methodology. [Rozum, 1993] [Coleman Dangle et al., 2005]

An organization has to define processes for tasks and record results of executed processes, to reap the benefits that process improvement promise. However, defining processes and recording results can be done in many different ways and on multiple levels of granularity. The software process is often authored and published using dedicated tools to gain the best possible process definition. The process definition can be written using a *software process meta model.* A well-known meta-model is the S*oftware & Systems Process Engineering Meta-Model* (SPEM) [SPEM 2.0].

SPEM defines the different parts of the process meta model. The process structure is defined in SPEM as follows:

> **Process Structure:** This package defines the base for all process models. It supports the creation of simple and flexible process models. Its core data structure is a breakdown or decomposition of nested Activities that maintain lists of references to performing Role classes as well as input and output Work Product classes for each Activity. In addition, it provides mechanisms for process reuse such as the dynamic binding of process patterns that allow users to assemble processes with sets of dynamically linked Activities. These structures are used to represent high-level and basic processes that are

> not textually documented. The structures are ideal for the ad-hoc assembly of processes, especially the representation of agile processes and self-organizing team approaches. [SPEM 2.0]

The software development process is published in some form, once it has been authored. Publishing the software process is done using standard documents, hypertext documents, application life-cycle management tools, or a combination of some of the previous and other tools. Publishing the software process for enactment by a human is called a process script, and the published process for enactment by a machine is called a process program [Feiler and Humphrey, 1993]. The publication of the software development process makes the process available to the users of the process. The publication format impacts the way the process is enacted.

When a software project is executed using a published software process, then the software process can be said to be enacted [Feiler and Humphrey, 1993]. A software process enactment is strongly influenced by the publication of the software process. If the software process has been published using some form of static document, then the enactment is enforced by human resources. If the software process is published to development tools, then it is often also enforced in those tools.

When a software process is in use and metrics of the process are gathered, then some form of process improvement can be made. Improving the process is done by repeating the phases of changing the old process, authoring a new process, and publishing it. [Feiler and Humphrey, 1993]

Most process models are incompatible with small companies. Maturity based models for software improvement like CMM can be disastrous for small companies [Shaikh et al. 2009]. Other process management solutions are also problematic in very small companies. One of the reasons is the software process maintenance. The next chapters will discuss the problems and propose a solution for process management in small companies.

6.2 Problems and solutions to process maintenance in small companies

6.2.1 Software process maintenance issues with dedicated tools

A software process that has been authored using a software process meta model, can be published in a process tool that can enforce the enactment of the process. Writing, publishing and enforcing a software process model with dedicated tools, like application life-cycle management tools, have some drawbacks when they are used in small companies with versatile projects.

Software development process authoring, in systems where a dedicated process tool is used, requires both human resources and infrastructure for the process authoring. The tools are used by process engineers and require some extra skills. The process authoring and maintenance is easier with a process authoring tool than with plain old documents. If the processes are authored on an fine granularity level, then a very large number of processes must be authored for companies with diverse projects. The authoring feedback cycle can become very long if there are multiple processes for different project types and limited resources for process authoring and software process improvement.

Software process publishing and enforcement are often integrated into an application life-cycle management tool. The integration of publishing means that software development tasks are visible in the application life-cycle management tools. The enforcement becomes automated to some degree by the application life-cycle tool, with the requirement of artifacts or acknowledgments in the tool. Application life-cycle management tool integration presents some problems for smaller companies. A life-cycle is managed automatically by an application life-cycle management tool for development tools that integrate seamlessly into the management tool. Most application life-cycle management tools have a limited set of supported tools for integration. Being dependent on tools that integrate with the application life-cycle management solution can form dangerous vendor lock-in or a preference for more complex or expensive tooling, than what is actually needed. Some resources are also required for maintenance of the skills needed to use the application life-cycle tools and the other development tools, when integrated to the application life-cycle management tool.

6.2.2 Software process maintenance issues with a separate process document

A software process that has been authored to a separate resource, for instance an HTML page, faces its own set of challenges for a small software development company. This is a competing way to, the previously mentioned, dedicated application life-cycle management tools, and can be combined with other methods of process management. Authoring a separate software development process provides a lot of freedom, for example, because the process tools do not require working integrations with the software development tools. On the other hand, authoring a separate process does require more human resources, to verify the enactment and enforcement of the software development process, and the actual verification can be very difficult.

Authoring a software process document separately does not necessarily require a dedicated authoring tool, and it gives a lot of freedom of the form authored process. Authoring a software process without dedicated authoring tools, such as authoring the process in a text document or HTML, becomes very tedious or impossible to maintain if the process requires active modification, updating, or the process is subject to constant software process improvement.

The publishing of a software development process can be done in printed documents or on a web page. A problem with the separately published process is that the management of the published resources can be difficult. The problems manifest themselves especially in situations where a new version of the process is published and older versions exists, for instance, in printed form. In these cases the propagation of changes must be managed with human resources.

The enactment and enforcement of a separate process have a few challenges. To actually enact the process all developers must actively be aware of the process, understand the process, and perform according to the process. The enactment also includes the propagation of changes in the process to the organization. All mentioned enactment tasks require a lot of resources. For instance, resources are required to notify people of the process and developers use time to read and understand the process.

Another challenge, for the separately authored process, is the enforcement of the process, to actually verify that a process is followed. The verification, that a process is followed, is difficult and requires a verification process and resources to follow the verification process. Resources that are required for enforcement are, for instance, that people fill in reports on what they do and process engineers verify the results.

All issues mentioned in this section apply for all separately published processes, but especially in cases where the organization grows larger. The complexity of the process maintenance, and especially the process enactment and enforcement efforts, are difficult and expensive with a separately authored software process.

6.2.3 A software process maintenance solution for small companies

I propose a third method to author, publish, and enforce parts of a software process, the method is to author, publish and enforce the process practices in day-to-day development tools and integrate software process improvement into daily development work. This method is called Implicit Process Improvement with Day-to-Day Development Tools or IPIDDDT. IPIDDDT strives to solve the problems, that other software process maintenance methods present, for small companies with versatile projects and very limited resources. The solutions that are presented here reflect the issues that are discussed in the previous Sections 6.2.1 and 6.2.2.

The solutions to process authoring in the new IPIDDDT method is to find a way to author the process using day-to-day development tools, while at the same time making them instantly available and visible for the developers. This means that the development tools must be selected and the settings of the tool altered, as well as a template that is elaborated on so that the way to solve a certain task is always done in a similar manner. Multiple processes are easy to maintain because the processes also use the same tools as others and the granularity is always limited by the tools modifiability and capability in making templates.

The solution for publishing a process is to maintain all settings and templates in a common repository. Any issue that do not have an existing solution can be added to the repository with ease. The publication of the process in the day-to-day development tools enables the process to be easily accessible and removes the need for extra knowledge of tools just for the sake of the process.

Enactment and enforcement of the process is solved because only things that are in the development tools are enforced. If the developer uses the tools and templates from the common repository, then no additional enforcement should be needed. A developer can disregard the process settings or templates, but disregarding the process is not prompted by efficiency penalties.

The staleness and slow feedback cycles of the process maintenance are addressed by a community type solutions to the maintenance of the process. The authored process is available to all in the repository and it can be updated and improved by all users. The process is readily available for optimization, and it is easy to maintain and keep relevant, with little extra effort. The most important issue is, as well as the access, that the culture of process improvement has to be active in the company.

The problems of traditional process improvement efforts for small companies are listed in Table 6. The table lists problems and solutions that the IPIDDDT model could provide.

Problem	**Solution**
The process management requires dedicated infrastructure	The process management is transferred to the day-to-day development tools
The process enactment requires extra effort from developers	The process enactment is integral in the development work
The process improvement requires dedicated resources	The process improvement is integrated in day-to-day work
Numerous project type processes require a large maintenance effort	Project tasks are defined as processes only when needed
Complete methodologies address non-issues	Process areas are selected only when actual ROI is evident
Process improvement requires a dedicated effort	Process improvement is integrated into day-to-day work
Software process authoring requires extra effort	Process authoring is integrated into day-to-day development tools

Table 6: Problem - solution table for the IPIDDDT method

The drawbacks to the IPIDDDT solution are numerous from the perspective of a general solution for all software process requirements, but IPIDDDT is not an explicit process as such. On the other hand, the IPIDDDT drawbacks are quite few in the case of a small company, that fits the profile of multiple project and process types, little or no dedicated resources for the software process, and a small number of concurrent developers on projects. The strengths and weaknesses of the proposed solution are presented in Table 7.

Strengths	Weaknesses
Process management and improvement is driven by developers for developers	Process enforcement is limited in the tools
Process management forces the process to addresses few key areas	Some complete and complex methodologies are difficult or impossible to implement with tools alone
Multiple process fragments enable a multitude of projects with low overhead	No single homogenous complete repeatable process
Cheap constant process improvement is part of the system	High-level predictability and automatic metrics are difficult to implement
Control and focus on the actual development activities.	No control over managerial or organizational processes

Table 7: Strengths - weaknesses table for the IPIDDDT method

The IPIDDDT method aims to provide just enough process stabilization by enforcing some form of discipline through tool selection and practices related to those tools. The selection of tools must be done with much care so that the overhead that is removed from traditional process management is actual reduced waste.

The IPIDDDT method has a focus on actual developer practices and not on managerial and organizational issues. The method is not suitable for large organizations because it has no clear dimension in organizational questions. The IPIDDDT method could be combined with Scrum to deliver a more scalable and complete process.

This system of solutions builds upon the agile concepts where the developers must have freedom and independence in their work, which is limited by peer selected methods and tools, but is is extended to the process realm as well. The process improvement targets are set up by the developers and leaders but the actual process is authored and maintained by the developers. This method is presented in Chapter 7.

7 The IPIDDDT method

7.1 Implicit process improvement through day-to-day development tools

7.1.1 IPIDDDT basic principles

IPIDDDT is a method to enforce some software development practices in an organization with the use of selecting software development tools and implementing tool downloading and managing practices. The software development process is not explicitly authored as a complete process, but it is programmed into the tools that the developers use, without an actual process definition language. The process is constantly changing because all developers can modify the process at will. The freedom of the tool settings to change and the fact that the process is not authored but programmed into the tools, signify that the process is not explicit but implicit.

One of the cornerstones of IPIDDDT is tool selection. Tool selection embraces the fact that the tool that is used shapes the way a task is done. Through this knowledge process management is turned upside down. In a traditional model of process management task is authored to a process, and published. A developer on a project uses the process to identify a task and the process specifies how that task will be done. The following sequence takes place in the IPIDDDT model.

Recurring tasks are identified and a suitable tool is selected for the work. Then when the task recurs in a new project, the selected tool already identifies a way to solve that task. The IPIDDDT model embraces the fact that the selected tool has an impact on how the task is viewed and how the task is solved. The tools settings are modified to reflect the solution.

Recurring tasks may be more or less complex and only selecting an appropriate tool does not guarantee a certain type of solution to the task. Templates and tool configurations are the solution for more direct control. Templates and tool configurations are used to direct the solutions, so that similar tasks are solved in a similar way, which enables better prediction on the resource consumption and outcome of the task, and enables knowledge transfer.

Process improvement is an integral part of IPIDDDT because when using a template in day-to-day development, developers improve on the templates as needed. In this way the templates never get stale. The selection, incorporation and customizing of libraries as well as creating own libraries of reusable components support the software process.

7.1.2 IPIDDDT examples

Examples of ways to define a process with the use of development tools are:

- Requiring certain artifacts at specified moments of development
  - Unit tests by made with a testing framework
  - Acceptance tests for functional requirements
  - HTML mockup before development process
- Requiring the use of a certain tool or template for communicating certain issues
  - Use of version control
  - Use of bug tracking
  - Listing requirements in Wiki
- Configuring development tools in the repository to enable some specific actions
  - Downloading all development tools from a tool repository, where the tools are configured for the organizations needs.
  - Starting projects and tasks with wizards that are configured for the organization. For instance, starting a web project with a wizard that has a certain package structure which includes test packages and a deployment descriptor that includes references to the default libraries.
  - Using artifact repositories and automated dependency management for external and internal libraries. Artifact repositories create control over which libraries and versions are used.

7.2 IPIDDDT implementation

7.2.1 IPIDDDT implementation process

Implementing the IPIDDDT method, for a software development company, can be done using an implementation process. A suitable implementation process is presented in Section 7.2. The implementation process defines a set of steps that enable a company to start using the IPIDDDT model. The implementation process aims to be short and applicable to the target organization type - a small agile organization. The described process can seem complicated because the process is explained only in brief and on high abstraction level, but it is illustrated more verbosely in the implementation example in Chapter 8.

The IPIDDDT method implementation assumes that a target company exists (or is defined in a theoretical case), that the target company is relatively small, and the target company strives to gain some benefit through agile practices. The IPIDDDT method implementation process is split up into five steps, some steps can be further divided into smaller sub-process steps. The main steps are

1. Selecting general selection criteria to define initial pool
2. Selecting the extreme pools for the the tool selection
3. Tool selection according to the IPIDDDT targets
4. IPIDDDT modification and customization of the tools
5. IPIDDDT management and iteration.

The IPIDDDT target definition and IPIDDDT tool selection follow, in general, the general tool selection process that is defined in Chapter 5, and the selection process includes three iterations of the tool selection process. The IPIDDDT model uses the software development tool categorization of Chapter 4, to select tools for certain categories and to help discuss the tool selection in general.

7.2.2 Selecting general selection criteria to define initial pool

Creating the general pooling of tools by selecting a set of very general criteria for the tools. These general criteria can be criteria like “The company does not have in-house infrastructure, no tools that require in-house infrastructure are considered.” The criteria can also be very general like “The tools should be considered 'agile'”. This set of criteria is only a general reference for the tool criteria and is used mainly in creating the tool selection pool of the first phases. The tool pooling is not done explicitly in this section, but the tools that are selected in the next sections are considered in reference to this pooling criteria, as well as the primary criteria of that selection.

The sub-process steps are

- 1.1 Define the organizations size, resources, and scope.
- 1.2 Define the organizations general process type.
- 1.3 Refine the organization size and general process type to general criteria that serve as a tool for creating the general pool of tools.

The first step of the software process Step 1.1 consists defining of the organizations size, resources, and scope. The definition of the company is required for selecting feasible tools for a certain company with certain resources.

Process Step 1.2 requires to define the company's general process methodology, or the desired general process methodology. This methodology is the general methodology where solutions to the process improvement goals are searched and the methodology will be essential in the selection of tools in each subsequent selection.

The process Step 1.3 is the refining of the findings of Steps 1.1 and 1.2 to tool requirements. These tool requirements form the implicit tool pool for the first two selections, and the secondary selection criteria for all subsequent selections.

7.2.3 Selecting the extreme pools for the the tool selection

The next two tool selections are more specific pooling selections. The selections aim to form two extremes that are sustainable in this type of software development. The goal of the first selection is to be a very process oriented selection, which includes all software development tools that can be justified in this type of development. The goals of the second selection is to select the bare minimum tool set, that is conceivable in this type of development. These two extremes will form the tool selection pool for the tool selection of implementation process Step 3, even though the last selection is not confined between these two extremes.

The two pooling selections that strive to find the extremes follow mostly the general tool selection method that was presented in Section 5. The differences to the general tool selection method are that I) the primary goals are already established by the tool selection rationale and the criteria from the first process step, and II) the final phases of review and selection are done less rigorously, to provide a tool category and a good exemplary tool to represent that category.

The steps of this process completed twice, firstly the a) -process and secondly the b) -process. The steps of the process are the following:

2.1 Using the goals of a) “extensive tool selection” and b) “lean tool selection” to create the main requirements for the tools

2.2 Using the implicit tool pool from the process Step 1.

2.3 Selecting a tool group in reference to each tool category.

The first step of the selection process is finding the goals of the selection. The goal of the first selection is to find a very process oriented tool set, which includes all software development tools that can be justified in this type of development. The goal of the second selections is to select the bare minimum tool set that is conceivable in this type of development.

The criteria of the first process step and the extensive tool selection rationale for the selected process methodology, are modified into criteria, in step a). In step b), lean concepts are used to determine if the selected tools are actually necessary, or could the target company manage with simpler of fewer tools.

The metrics for these selections is a very hierarchical system where the primary criteria is the criteria mentioned as the goals, and the secondary criteria are all the ordered criteria from the first process step. The first process step includes also the implicit pooling that limit some tools as being out of the scope of this organization.

The selection is done using the criteria that was presented in the previous process step, and the implicit tool pool that is formed by the criteria of the first process step. The selection and ordering of tools is done for each category separately and using the tool taxonomy that is presented in Chapter 4.

The next step of the selection process is the selection of tools, in this case only tool groups are selected and perhaps an exemplary tool that embody most criteria well. The tool groups that are selected form the two extremes for the tool selection pool of the next process step.

7.2.4 Tool selection according to the IPIDDDT targets

Selecting tools according to the IPIDDDT rationale is the next process step after creating the extreme of pools tools in the previous process step. The tool selection of the IPIDDDT method follows the general tool selection method of Section 5.

The steps of the selection process for the IPIDDDT method are:

3.1 The selection of process improvement targets and refining them to the primary requirements for the tool selection of the organization.

3.2 Combine the primary requirements and the requirements of the first process step (Step 1) as the secondary requirements to a hierarchy to define the requirements and metrics.

3.3 Use the range of tools for each taxonomy from the results of the previous process step (Step 2) as the selection pool.

3.4 Review the tools from the pool against the requirements and selection pools to select the best tools.

The first step of the sub-process, Step 3.1 selection of process improvement targets and refining them to requirements for the organization, is a sub-process in itself. The sub-process consists of the following self-explanatory steps:

3.1.1 Selecting a set of company targets or benefits that can be reached through process improvement.

3.1.2 Verifying each target is measurable, so striving for the targets can be considered methodical process improvement.

3.1.3 The targets will be broken up to more concrete results that can be translated to concrete tool requirements or process methodology practices.

The next process Step 3.2 forms the metrics and complete requirements for the tool selection. The step combines the primary requirements that are formed in Step 3.1 and the secondary requirements that are formed in the process Step 1 are combined in a hierarchical system. This hierarchical system is the requirements and metrics for the selections process.

Process Step 3.3 presents the tool selection pool. The pool is formed from the results of process Step number 2. Process Step 2 a) presented an extreme with the most elaborate tool set conceivable with the primary limitations from process Step 1, and Step 2 b) presents a very lean set of tools with the same primary limitations. The selection pool, or range of tools is the range that is positioned between these two extremes, the results from 2 a) and 2 b). The pool is not totally constraining and it can be exceeded at need, but the range presents a good reference.

Process Step 3.4 consists of reviewing tools and tool groups between the extreme tool selections from Steps 2 a) and 2 b). The review is done using the tool categorization for each category separately but taking the other selections into account. A final tool selection is done using this review of tools.

7.2.5 IPIDDDT tool configuration and customization

The next process Step "4 IPIDDDT tool configuration and customization" includes tasks to enable the process. This process phase is very important because the actual day-to-day development tool part of IPIDDDT is defined in this process step.

The software development tool configuration sub-process steps are:

4.1 Establish process for tool and tool configurations distribution.

4.2 Set up initial tools (selected in process Step 3) in the repository of Step 4.1 and modify and configure the tools.

4.X Maintain and improve the IPIDDDT configuration in day-to-day operation.

The first process Step 4.1 strives to establish a process to distribute the software and the software configurations. Creating the distribution process includes the implementation of a repository structure, to maintain the software that is used, the configurations to that software and to maintain all templates. The process of software distribution can be created, after the implementation of the repositories. The distribution process defines how the repositories are used and how the changes made in the repository will be propagated throughout the organization.

The second Step 4.2 consists of setting up the repositories and the tools initial configurations. This step varies a lot depending on the tools, repositories, and the organization type.

The final Step 4.X is an ongoing step, that should be started and continued until the next iteration of the improvement effort. The Step 4.X consists of maintaining and improving the configurations and settings of the software development tools that have been selected. This process is also able to switch tools, in case a newer version or a better tool emerges. The switching of tools, like the tools that function as repositories for the process tools, can be difficult or impossible in normal use. These tools, that can not be switched in normal use, must be switched in the large improvement iterations of implementation process Step 5.

7.2.6 IPIDDDT method iteration in constant process improvement

The process Step 5, or the iteration of the IPIDDDT implementation, is done after a substantial time or change in the composition of the organization of the initial setup of IPIDDDT. The aim of this process step is to select new tools and re-align the tooling and the goals according to the changing environment of the organization.

The sub-process of process Step 5 consist of

> 5.1 Verify progress based on the improvement metrics defined in process Step 3.2.
>
> 5.2 Update the IPIDDDT goals according to reviewed priorities and progress and rerun the process steps 3 – 5, or in case of organizational changes rerun steps

Process Step 5.1 is verifying the progress based on the improvement metrics set up in process Step 3.2. The progress is verified by evaluating the metrics, and selecting which goals have been attained, which goals have not been reached and an analysis into the possible reasons of the success of the IPIDDDT effort.

The second Step 5.2 stands for the iteration of the IPIDDDT implementation process. The IPIDDDT implementation process will be iterated completely in the case of significant changes in the organization or in case of problems in the initial tool setup. If the improvement effort has been a general success and the organization structure has stayed mainly the same, then only the last process Steps 3 - 5 are necessary to repeat.

7.2.7 The IPIDDDT implementation process chart

The IPIDDDT implementation process steps have been gathered on the level of the main process and the first level sub-process steps in the following Table 8. The process steps have been presented in more detail in the previous sub-sections of Section 7.2. The table is not a complete reference, as some sub-process steps are not visible, but the table gives a good overview of the process steps.

Process step	Sub process step
1 Selecting general selection criteria to define initial pool.	1.1 Define the organizations size.
	1.2 Define the organizations general process type.
	1.3 Refine the organization size and general process type rationale to general tool selection criteria, that serve as a tool for creating the general pool of tools and the secondary criteria for each selection.
2 Selecting the extreme pools for the the tool selection. First process steps starting with a) and then starting with b).	2.1 a) Using the goals of "extensive tool selection" to create the main requirements for the tools.
	2.1 b) Using the goals of "lean tool selection" to create the main requirements for the tools.
	2.2 Using the implicit pool of tools defined by process step number 1.
	2.3 Select a tool group in reference to each tool category in the tool categorization.
3 Tool selection according to the IPIDDDT targets.	3.1 The selection of process improvement targets and refining them to the primary requirements for the tool selection of the organization. (Consists of a sub-process.)
	3.2 Define and setup metrics for the improvement goals, to enable verification of the improvement effort.
	3.3 Combine the primary requirements and the requirements of the first (1) process step as the secondary requirements to a hierarchy to create the requirements and metrics.
	3.4 Use the range of tools for each taxonomy from the results of the previous process step (2) as the selection pool.
	3.5 Review the tools from the pool against the requirements and selection pools to select the best tools for each category.
4 IPIDDDT modification	4.1 Establish process for tool installation and configurations distribution
	4.2 Set up initial tools (selected in process step 3) in repository and modify and configure the tools.
	4.X Maintain and improve the IPIDDDT configuration in day-to-day operation.
5 IPIDDDT process improvement process	5.1 Review progress based on the improvement metrics set up in process step 3.2 section.
	5.2 Update the IPIDDDT goals according to reviewed priorities and progress and rerun the process steps 3 – 5, or in case of organizational changes rerun steps

Table 8: Steps of the IPIDDDT method implementation process

8 Implementing IPIDDDT for a theoretical case

8.1 IPIDDDT implementation

A method called IPIDDDT has been presented in Chapter 7. The IPIDDDT method is a method to enable process benefits with an implicit process that is programmed into the tools of the software development organization. This Chapter 8 presents a theoretical case where the IPIDDDT method is implemented. The method implementation is done on a high abstraction level because the scope of this thesis is not to make an actual implementation, but to present the method in a more concrete format than the presentation in Chapter 7.

The method implementation follows the process steps of the implementation process. Step 1. “Selecting general selection criteria to define initial pool” is presented in Section 7.2.2, the step is implemented in sections 8.2.2 and 8.3. The second Step 2. “Selecting the extreme pools for the the tool selection” of the process is presented in Section 7.2.3, and used in Sections 8.2.2 and 8.2.3, as well as Step 2 a) in Section 8.4 and Step 2 b) in Section 8.5. The third Step 3. “Tool selection according to the IPIDDDT targets” is presented in Section 7.2.4 and used in Section 8.6.

The method implementation is not performed completely because of the high abstraction level as well as the difficulty of implementing some steps without a concrete organization. The steps that are outside the scope of this theoretical implementation are Step 4. “IPIDDDT modification” and Step 5. “IPIDDDT process improvement process“. Step 4 is only briefly brushed upon in Section 8.7 while the theory was presented in Section 7.2.5. Step 5, that was presented in Section 7.2.6, is a step that is not discussed at all in Chapter 8.

The theoretical nature effects some aspects of the process implementation case. The affected areas are

I) The following company is presented on a general level, and everything that is required is extrapolated from that.

II) The metrics are a very general subjective ordering.

III) Tool groups are more important than actual tools.

IV) The review of tools is done on a very high abstraction level and simply for the purpose of presenting a method.

V) The review, ordering and selection are done simultaneously in the sections where the selection is presented. These process implementation details should be done in a more exact fashion in a true implementation case.

8.2 Describing the target company and project

8.2.1 Target justification

The target project and process are described according to the authors personal experiences in small scale software development. The project and process improvement targets are examples of issues that often need to be addressed in small scale software development. The target company and process and project targets are presented to enable a goal oriented presentation of the IPIDDDT method implementation.

8.2.2 Small project development

This section enacts the IPIDDDT implementation process Step 1.1 where the organization size and type is defined. This Chapter 8 will try to implement the IPIDDDT method for a theoretical software development company that represents a larger group of companies. The implementing theoretical company will be defined in this section.

The small scale development is done in a small software development company with very small software development projects. A small project is a project that requires less than two months of work for a single developer. The project also requires only one software developer and possibly an architect at the beginning of the project. The roles are handled by a single person and that one person creates all artifacts of that role. The developers communicate directly with the customer. The projects are made in a single iteration. The company's employee number is between 10 and 20 people. Returning customers often buy another project on the same piece of software as the original project.

The company's projects are versatile web projects, most of the projects are written in the Java programming language, and the rest in PHP or other web oriented languages. The company installs the projects on externally hosted servers and maintains some of the solutions.

8.2.3 Agile software development for small project development

This section enacts the IPIDDDT implementation process Step 1.2 “Define the organizations general process type”. The process methodology that is selected to achieve these targets is a generic agile software development process. A generic agile software process stands for a software process that has the main agile traits. The main agile traits are presented in Section 2.2. The process methodology is not a defined software process, but a generic agile software process because the company size and type as well as the process enactment system can not enact a complete software process methodology. The tools that are selected are not selected to enable complete methodologies but only certain principles and practices.

The lean software development tool selection principles are applied to modify the selection of agile software development tools so that they apply better to the small company and small project scenario that is presented in the previous Section 8.2.2.

8.2.4 Targeted process benefits

An explicit software process is always designed to achieve some goals for the enacting company. The targets that can (and should) be achieved must be defined in order to create a process that meets those needs. Process benefits in general were presented in Section 2.2.2.

The process benefits that this company seeks are defined in three ways for the IPIDDDT method. These three aspects are applied in Table 9.

1) There must be a clear process benefit that is aimed for, and that can be reached with process improvement.

2) Each benefit must be measurable, so true process improvement can exist.

3) Actual methods to achieve these goals are presented.

Process Benefit	**Benefit break up**	**Actual methods or practices**
Increase in productivity and employee satisfaction. *Measured by accuracy of project prediction, return on investment for projects, and shorter time to delivery.*	The company will use fewer tools that serve more than one purpose.	Multipurpose tools
	Methods, skills and knowledge should be distributed across he organization.	Peer code review
		Tools that enable sharing (wiki, version control, communication)
Better quality. *Measured by savings in quality control.*	Less defects found in the final phases of product delivery.	Test-driven development
		Peer code review
Increase in customer satisfaction *Measured by direct discussion and number of returning customers.*	The final product meets customer expectations better.	Agile requirements process with face-to-face communication, requirements mockups, and requirements as functional test cases.
	Information and source code of finished projects should be available to more than one employee.	Information management, code management, and program deployment from old projects must be standardized, and stored in tools that enable sharing.
Increase in employee satisfaction. *Measured by meetings and average employment time*	Employee skills must be up to date.	New technologies and methods that suit company portfolio are tested on most projects.
	Skills and knowledge must increase in company.	Peer code review
		Pair programming

Table 9: Process benefit deduction for the target company

The most important process methods or practices that the tools should deliver can be deduced from the last column of Table 9. The practices from the last column have been placed in Table 10 below. Table 10 contains the practices that are enabled and enforced through tool selection in the IPIDDDT selection phase in later sections.

1. Multipurpose tools
2. Peer code review
3. Tools that enable sharing and communication
4. Test-driven development
5. Functional testing
6. Information management should be standardized
7. Source code management standardized
8. New technologies and methods that suit company portfolio are used in most projects

Table 10: Targeted software process methods and principles for the target company

8.3 General tool selection criteria

This section presents Step 1.3 “Refine the organization size and general process type rationale to general tool selection criteria that serve as a tool for creating the general pool of tools and the secondary criteria for each selection.” in the implementation process. This step was presented in Section 7.2.2 of the implementation process. The theory for the tool selection was presented in Chapter 5.

The tools are selected in this study using a set of selection criteria. The selection criteria are not based on evaluation of the tools, but on a perceived value and view of the tools. The most important selection criteria is different in all sections, while the rest of the criteria are stable. The criteria are emphasized according to the target company and type described in Section 8.2.

The company and project sizes are small which makes expensive tooling unsustainable. Because of these reasons, the “Total cost of ownership including infrastructure” is one of the most important factors to consider, in the selection. The total cost of ownership includes at least initial costs, licenses, maintenance costs, infrastructure, and training.

The selection is made according to a perceived value of tools instead of an actual evaluation. The perceived value is affected strongly by the fame of the tools, and the number of articles naming them in the web communities. The most important reference community for this selection is the agile community. An important criterion for the tool selection is “High perceived value, esteem, or fame in the agile community”. A high esteem often rises from being standards compliant, which enables the tool to be easily exchangeable to another standards compliant tool, which will protect an organization from vendor lock-in.

The tools that are selected must be “Easy to use and versatile” because of the company size and project versatility. Small projects mean that there are very small margins for training to use new tools, especially complicated tools that are expensive to learn.

The hosting of tools is an important consideration that is closely tied to the total cost of ownership. A tool that is hosted in-house requires infrastructure and manpower as well as skills to maintain. These fixed costs can be problematic for a small company. On the other hand, the externally hosted solutions can be far more expensive than an in-house solution in a medium to long term scope. Versatility in hosting options is an important consideration, which enables the company to choose a suitable tool to affect their cost structure.

The basic selection criteria that where gathered in this section are presented in Table 11. The table is referenced frequently in the following chapters.

1. Agility in Section 8.4, 1. Waste-Value ratio in Section 8.5, 1. Value according to IPIDDDT principles in Section 8.6.
2. Java tools with good usability in PHP and other web languages and concepts.
3. Total cost of ownership including infrastructure.
4. High perceived value, esteem, or fame in the agile community, with standards compliance.
5. Ease of use and versatility.
6. Multiple hosting options → the tool can be hosted on site or at an external service.

Table 11: General tool selection criteria for the target company

8.4 Tool selection according to Agile principles

The selection of tools according to an agile criteria is done in this section, which enacts the implementation process Step 2 a) “Selecting the extreme pools for the tool selection, for the extensive tool selection”. The theory of this section is presented in Section 7.2.3.

8.4.1 Selection criteria

The most important rationale or criteria for the selection is 1. Agile criteria in Table 11. The agile criteria signifies that tools that support agile practices are emphasized and no tools should be selected that require a rigid process or that need a lot of work to maintain. The secondary selection criteria 2. - 6. are presented in Table 11.

The basis for the selection of the agile tool set is based on Section 3.1. The most important factors in selecting tools with agile development in mind are the following: The tools should work automatically and integrate seamlessly. Agile processes require that the developers do not have to make an extra effort for the sake of the process, but rather work as usual but according to the process guidelines, which itself creates predictability and other process benefits. Another important aspect of agile software development is that tools are not used for the sake of tools, but the tools have to deliver tangible benefits and they can not create more than minimal overhead.

8.4.2 Project management tools

Project management tools were split up to the subcategories of Process management, Resource management and requirements prioritization, and Process visibility in Section 4. The different categories are discussed in this section and the most appropriate tools are presented for each category.

Agile process management tools that support agile processes can be application life-cycle management tools like the Jazz framework which aim to support any type of software processes and manage the process as a whole. The most extensive and rigid application life-cycle management tools that enable agile processes are however rarely used because their value to waste ratio is not evident, especially in smaller organizations and diverse projects. Process authoring is also done with faster solutions than actual process authoring tools. The most important process management tools for agile companies are resource mapping and tracking across projects. Resource mapping and tracking is often done using only the selected project management tools. The processes employed for agile companies are authored to simple documents. Managing the process with tools is not necessary in agile development.

According to [Barnett and Schwaber, 2004] some sort of project management tools are “should have” agile software development tools. Agile project management tools can be the lightest possible tools, spreadsheets and wikis. More complex tools that are employed are tailor made agile project management tools or agile application life-cycle management tool sets. Agile project management tools are often wiki extensions that follow the basics of a project: requirements, tasks, task assignment and project velocity. Many more complex solutions combine the traditional software process and project tools with templates and project models for agile projects. A complete industry has also been built upon the notion of agile project management tools.

Agile software process methodologies manage resources and requirements in integrated project management tools. If project management tools are not used then the other solutions are lightweight solutions like spreadsheets and wikis. Spreadsheets and wikis are often employed as support to the project management tools or, in case no project management tools are used, then they are used as the sole means to handle resource and requirements management. Agile software development often includes requirements both in the form of images, mockups, and acceptance tests, these types of requirements are references in the requirements prioritization solutions.

Process visibility metrics are essential, for any company that strives to reap any process related benefits, like information on development velocity and project estimation. The process visibility is addressed in software development tools, in agile companies, in the same way as other development companies. Integrated project management tools for agile development contain some type of velocity measurement. Visibility in the process is integrated into agile project management tools to some degree, but to actually be aware of the process development, some extra followups are needed with spreadsheets.

The process authoring and publishing solution that is selected for this company has to be a system where the production of content and publishing is simple, fast and immediately available to all. A good solution is an intranet content management system that can be hosted on site or bought with a software as a service model. Possible solutions that fit are WordPress type blogs or a Joomla type content management system.

The best project management solutions for agile companies are intuitive and effective development tools. The tools that are chosen for a small company with limited resources for process management are 1) intuitive and simple, 2) can be easily purchased as a service, and 3) supports the agile mindset. One such product would be the Mingle project management platform from ThoughtWorks. The project management solution described might have problems to enable enough visibility to the process or managing a large number of different types of resources, so spreadsheets will be applied for the areas where project management has issues.

8.4.3 Communication

Communications tools were split up to the subcategories of Persistent documentation, On demand documentation, and Direct communication in Section 4. The different categories are discussed in this section and the most appropriate tools are presented for each category.

Persistent documentation in agile software development is minimal. A concept that embodies the principles of agile documentation is “just enough documentation” which stands for creating the most efficient amount of documentation and scrapping documentation that has served its purpose, documents should also aim to be “just barely good enough” to be as effective as possible. Documents that are known to be of value for a longer period are stored persistently as well as essential documentation of the final product, but documentation of details or information with a lesser value is minimized in favor of “on demand documentation”. The commented source code as well as many other types of executable documentation like unit tests are preferred to ordinary plain old documents. The tooling for source code commenting and unit tests are discussed in Section 8.4.4. [Ambler, 2010]

On demand documentation is a very important part of documentation in agile software development. Most documents that are written on demand are supporting a more direct type of communication that is the preferred way to operate. Other on demand methods of communication include any communication medium at hand like improvised modeling with post-it notes and documenting the results with a digital camera. [Ambler, 2010]

Direct communication is the main type of communication in agile software development. Other forms of communication are often only for supporting direct communication. Enabling direct communication between developers, as well as developers and product owners, is essential because of the strong emphasis on direct communication in agile development. Some agile practices present a user story (requirement) as a promise for a future discussion between the developer and the customer, instead of the actual fully described requirement. [Ambler, 2010]

The communication tools that are selected for this company need to be simple, fast and support all the standard document formats that are used. All different types of communication must be supported towards the clients as well as inside the company.

The following tool set was chosen for a small company and agile mindset. OpenOffice.org office tools were selected, for persistent communication tools, because the tools are versatile, support standards based formats, and the total cost of the tool is low. The selection for on demand documentation is MediaWiki because of the ease of use and sharing, and the versatile licensing options. Standard e-mail is used for most non-live-communication with the user's preferred (free) client. Skype is used for instant messaging and VOIP because of the versatility of the tool, the licensing options, and the wide popularity of the program. Finally the presentation and conferencing needs are handled with Vyew web-conferencing.

8.4.4 Development activities

Tools related to development activities were split up to the subcategories of Software source code management, Deployment, and Quality assurance in Section 4. The different categories are discussed in this section and the most appropriate tools are presented for each category.

Agile software development considers source code management to be very important. Source code management tools are used in modern software development and many important agile concepts such as constant integration are impossible without source code management tools like software configuration management tools. Many agile practices like self-documenting code, self-testing code, automatic deployment, and constant integration affect and are affected by the different development activities including source code management. Most software development is done using integrated development environments. The value of the integrated development environment is emphasized in agile software development because development tools must integrate more seamlessly than in traditional development. An agile development IDE must at least integrate the writing of source code to the software configuration management tools and the unit testing tools.

Agile deployment management manages the configuration, building and deployment of software to different servers as well as running the required automated quality assurance. The level of automation and use of tools is strongly related to the development practices that are used. A wide tool set is required, for example, in a complete constant integration cycle with automated dependency management. The complete constant integration cycle, with automated dependency management, requires an automated build tool that fetches source code from the software configuration management tools, fetches dependencies with dependency management, deploys the changes using a build tool, runs the required quality assurance with testing tools, and publishes the new version in the dependency management repository. [Fowler, 2006 (2)]

Quality assurance of software code is an integral part of all software development. Quality assurance in agile companies is integrated to the development cycle. Integrated and automated testing tools are required for sustainable agile development, where all the code is tested in all iterations of the development effort. Integrated software development signifies self-testing code and automated tests. Some unit testing frameworks are needed to enable self-testing code on a unit level. Acceptance testing frameworks are required to verify the functionality related to the customer requirements of the software. Web based services require performance tests as well, for the nonfunctional requirements of the software. Finally, user interface tests are required in most web projects that have a user interface. All tests are run on all committed code at short intervals in the case of constant integration. [Fowler, 2006 (2)]

The development solution that is selected for this company is a system where the production of code is simple in a standard Integrated Development Environment (IDE). The very broadly supported and available Eclipse IDE is selected because of the large number of development languages and community support that are available. The source code will be managed on a subversion server for software configuration management, automatic backup, as well as support for continuous integration.

The deployment tools that are selected are the CruiseControl framework that has gained much popularity and supports constant integration, and it will be integrated with the Maven build tool and a repository manager called Nexus, a repository manager for Maven's dependency management. The source code will be automatically deployed to a quality assurance server with CruiseControl and Maven. The quality assurance tools that are selected are the xUnit family for unit testing, FitNesse for acceptance testing, Selenium for user interface testing, and Jmeter for performance testing.

8.4.5 Summary of selected tools

A set of tools where selected for an agile company according to the rationale of agility as presented in Section 3.1 and the general criteria presented in Section 8.3, Table 12 contains that selection. The selection is the upper bound of the tool selection pool according to the IPIDDDT implementation process.

The general tools that were selected are common agile software development tools. The tools are not to be taken literally as the best tools, but rather as representatives of their tool types in this phase of the IPIDDDT process.

Activity	Sub Activity	Tool
Project management	Process management	Mingle, WordPress
	Resource management and requirements prioritization	Mingle, OpenOffice.org Calc
	Process visibility	Mingle, OpenOffice.org Calc
Communication	Persistent documentation	OpenOffice.org write
	On demand documentation	MediaWiki
	Direct communication	E-mail, skype (voip and instant messaging), Vyew (web-conference)
Development activities	Writing software source code	Subversion, Eclipse
	Deployment	CruiseControl, Maven, Nexus
	Quality assurance	Junit, FitNesse, Jmeter, Selenium

Table 12: Tool selection for a small agile company

8.5 Removing excess tools with concepts from Lean software development

8.5.1 Selection criteria and rationale

Section 8.5 enacts the implementation process Step 2 b) “Selecting the extreme pools for the tool selection, for the lean tool selection” the theory of this section is presented in Section 7.2.3. I elaborated in Section 3.2 on what lean software development is, and how lean software development concepts could be used to select software development tools. This section will modify the set of agile development tools presented in the previous chapter for a small software company. The tools will be selected using the concept of Value-Stream mapping. A Value-Stream mapping will reduce the tools that cause significant waste without significant value. Significant value will be assessed in relation to the size of company as presented in Section 8.2.2 and methods that the target company uses. The secondary criteria are presented in Table 11.

Software development in a small software company requires only a subset of the software tools needed in agile software development in general. A company that has, on each project, only one or two employees that are co-located do not need the same tool infrastructure as a globally dispersed team that creates large pieces of software with a large number of iterations. The tools that are necessary and proportionate for this type of organization will be chosen without much reference to the software process. Enforcing the software process through tooling will be addressed in the next Section 8.6.

8.5.2 Project management tools

Project management tools were split up to the subcategories of Process management, Resource management and requirements prioritization, and Process visibility in Chapter 4. The different categories are discussed in this section and the most appropriate tools are presented for each category.

Process management in a small company requires that managerial tasks and resource allocation is as easy and simple as possible, especially when the employee number and project sizes are small, otherwise the process management would be wasteful. Another important reason for process management is the visibility into several projects, and that the information persist to a useful degree. A very simple solution is sufficient when the number of employees is very small and the number of simultaneous projects is limited. A tool that will suffice is spreadsheets because the management of projects is limited to very few users and the management of projects is in the hands of the single users. All management that bring actual value to the customer can be handled through spreadsheets.

Managing a small company's resources can be done with simple tools like spreadsheets when the employee number and other resources are limited. Another important consideration is if the resources are managed by a single person or a few co-located people. The requirements prioritization can also be handled by a spreadsheet that is located on a shared drive. There are few reasons for a more complex solution in a small company.

Process visibility is questionable to be of significance in this size and versatility of projects because, for example, the predictability of small versatile projects is very poor in proportion to the complete project size. Prediction of the next iteration is impossible in cases where there is only a single iteration and predictions for other projects are difficult in cases where the projects differ from each other significantly and the project is processed in an agile way. Process visibility is not managed in any other way than with the existing spreadsheets.

8.5.3 Communication

Communications tools were split up to the subcategories of Persistent documentation, On demand documentation, and Direct communication in Chapter 4. The different categories are discussed in this section and the most appropriate tools are presented for each category.

Persistent documentation will be managed with file shares and office tools. Creating and storing documentation in a more complex fashion is too expensive for this simple type of development. The solution for managing the documentation does not significantly change between this and the previous agile section, but it differs in motivation.

On-demand documentation will be managed in the most suitable fashion selected independently by the developers. The on demand documentation will not be shared through the company because single developers with only a few projects manage their own on demand documentation in their own projects. The coherent storing and searchability of this data is not valuable in the waste-value sense of the word.

Direct communication will be managed with email, instant messaging, and VOIP. The value of web conferencing is not evident in a company of this size. Web conferencing is not required in internal communication because each projects only consists of co-located developers. An on-demand solution is sufficient in the case of broadcast type customer communications, if such need would arise.

8.5.4 Development activities

Tools related to development activities were split up to the subcategories of Software source code management, Deployment, and Quality assurance in Chapter 4. The different categories are discussed in this section and the most appropriate tools are presented for each category.

Software source code will be handled with the eclipse IDE that will be the standard tool, source code management is use full and will be handled with distributed source code management. Developers will manage their code locally with git and make backup copies with git too. Code will be swapped between developers local git repositories when switching projects or building on old code.

Deployment will be managed with simple build tools because versatile projects require unique deployment settings and the project sizes are less then one iteration. Automated quality assurance as well as continuous integration do not bring significant benefits to projects that last less than one iteration.

Automated quality assurance presents a new level of complication to software development. The complexity of automated quality assurance is added to iterative development because it is done more than once. The project size in this company was very small – less than one iteration and automated quality assurance must be considered waste. Unit testing is a discipline which helps to prevent the code from degrading on further development. Unit testing is also considered a waste in the case of small versatile projects without any reasons to suspect continued development. Other quality assurance tools that are used manually will have to be employed to meet customer requirements.

8.5.5 Summary of selected tools

A set of tools has been selected in this chapter. The tool selection was made by applying lean concepts to the agile tool selection that was done in Section 8.4. The new tool set is identified by being very light both in skills and expenses, imposing minimal control, and allowing nothing to come in the way of development. The tools that are selected after applying the lean concepts are presented in Table 13.

The selection that was made in this chapter emphasized immediate waste and value, while disregarding any projections to future development. In lean concepts projected value would be discussed with the concepts of temporary, or necessary waste, and they would not be so hastily set aside. The next Section 8.6, that presents a way of expanding the tool set through process needs, employ implicitly the concepts of temporary or necessary waste for greater benefits at a later point in time.

Activity	Sub Activity	Tool
Project management	Process management	OpenOffice.org Calc
	Resource management and requirements prioritization	OpenOffice.org Calc
	Process visibility	OpenOffice.org Calc
Communication	Persistent documentation	OpenOffice.org write
	On demand documentation	-
	Direct communication	E-mail, skype (voip and instant messaging)
Development activities	Writing software source code	Subversion, Eclipse
	Deployment	Ant
	Quality assurance	FitNesse, Jmeter, Selenium

Table 13: Tool selection table after applying Lean concepts

8.6 The tool selection according to the IPIDDDT principles

8.6.1 Selection criteria and rationale

Section 8.6 enacts the implementation process Step 3 “Tool selection according to the IPIDDDT targets” the theory of the implementation process step is presented in Section 7.2.4. The aim of Section 8.6 is to find a middle ground between the two selection extremes from implementation Steps 2 a) and 2 b). The selection of tools will be done, so that the tool selections enforces and enables software development practices that where discussed in Section 8.2.4 and gathered in Table 10. The other selection criteria are presented in Table 11.

8.6.2 Project management tools

Project management tools were split up to the subcategories of Process management, Resource management and requirements prioritization, and Process visibility in Chapter 4. The different categories are discussed in this section and the most appropriate tools are presented for each category.

The process management tools that where selected in the previous phases where either Mingle and WordPress for an agile company and spreadsheets when the immediate value of the tool was measured with lean concepts. This section will try to identify how or what process management tools might enable or enforce a benefit that was presented in Table 10.

The best tool for process management is the one that enables or enforces most of the process methods and principles, while keeping in mind the type of company that is described. The process method 1. “Multipurpose tools” is best handled by the Mingle solution. Mingle can be used for many communications needs and requirements management needs. It enables also automatically some of the process metrics that are needed for the process visibility. Mingle is the solution that excels in criteria number 3. “Tools that enable sharing and communication” as well as provides a part solution for the criteria number 6. “Information management should be standardized”. Mingle also provides room to grow as well as the immediate benefits. Other tools do not perform better in the other process methods and principles either so Mingle is the best solution in this case.

Resource management and process visibility use very similar tools to the process management section. The Mingle solution is the best solution according to the the same rationale as in the previous paragraph.

The problems associated with Mingle are that its automatic metrics do not provide complete visibility into all areas that might be interesting and that it is not open source nor free software. On the other hand mingle has high esteem in the agile community. In this case the process benefits outweigh these issues clearly and Mingle is the suggested solution for this company.

In the previous chapters OpenOffice.org Calc would be used for visibility and other tasks where mingle would not be able to reach. In reference to the criteria presented, an extra tool would be wasteful in many ways and must be cut from the selection.

8.6.3 Communication

Communications tools were split up to the subcategories of Persistent documentation, On demand documentation, and Direct communication in Chapter 4. The different categories are discussed in this section and the most appropriate tools are presented for each category.

The previously selected communications tools for persistent documentation where office tools for writing. Office tools are a standard way of communicating to customers and end users and can not be disregarded. Other types of documentation should be used in cases where other types of documentation are possible. Other types of documentation are for instance HTML documents and HTML mockups to present the user interface in stead of plain old documents. This setting is well in line with method: 1. “Multipurpose tools” from Table 10, for the IDE for writing HTML. Graphics that are required for the documentation should be created with the same tools as development graphics are produced, a suitable tool for this is GIMP.

On demand documentation could be produced on a common server where it is available to all or in a non structured way without centralized storage or sharing. The process benefits that are aimed for especially in the view of “3. sharing and communication” and “6. Information management should be standardized” favor a centralized model where information related to certain projects is stored in a structured manner. In this case a centralized project management system that includes a wiki has already been chosen so the process target of “1. multipurpose tools” helps to define the solution as Mingle.

Direct communications are important and they are handled in both the agile scenario and the leaner version in much the same way. The process targets do not require any changes to that selection, so the set from the previous phase: e-mail and Skype stands.

8.6.4 Development activities

Tools related to development activities were split up to the subcategories of Software source code management, Deployment, and Quality assurance in Chapter 4. The different categories are discussed in this section and the most appropriate tools are presented for each category.

The management of development activities is the most important value adding process of software development and tools to support it must be selected with much care. The methods and principles of Table 10 address a number of issues especially in the processes relating to development activities. The practices of “2. peer code review”, “4. Test driven development”, “5. Functional testing”, and “7. Source code management standardized”, are all mainly development activities.

Writing software source code is done in an integrated development environment. The integrated development environment that was proposed in the previous phases was Eclipse. Eclipse is the best solution for a small company even in reference to the process methods because the process methods do not bring new issues that would effect the previous outcome.

The software configuration management tools that where presented previously where Apache Subversion and Git. Apache Subversion is selected as a centralized software configuration management tool and Git as a distributed one. There are both pros and cons for the both solutions in the process methods and principles. “2. Peer code review” could benefit from a distributed system where the reviewer could verify the code first from a distributed repository before putting it in a quality assurance repository. On the other hand “7. Source code management standardized” points to a tool with a more standard process of operating like a centralized system. Apache Subversion is selected because the peer code review can be managed in a centralized system as well and automatic quality assurance like constant integration benefits from a centralized management solution [Fowler, 2006 (2)].

The deployment tools that were presented previously where Maven, Nexus, and Cruise Control versus simply Ant. Cruise Control is a very efficient tool for practices like constant integration, but in the case of a small company where there is usually only one programmer on each project, constant integration is not necessary. Automating the dependency management and creating a way to enforce dependency and software requirements is valuable for the process methods and principles “6. Information management should be standardized”, “7. Source code management should be standardized”, as well as “8. New technologies and methods that suit the company portfolio are used in most projects” so Maven and Nexus are justified. The process methods and principles of “4. Test-driven development” and “5. Functional testing” can be handled well by Maven and the tools for quality assurance. The tools for quality assurance are the same in both previous cases so there are no changes to those. The quality assurance tools are the xUnit frameworks, FitNesse, Jmeter, and Selenium.

8.6.5 Summary of selected tools

Tools that where selected for the target company and process with the IPIDDDT method are presented in Table 14. The tools that were selected present a good set of tools that would aid or benefit a company seeking the kinds of process benefits that were described in Section 8.2.4. The solution set present a very small set of tools, developers would work basically in two tools the project management system and the IDE, all other tools would integrate inside these main tools.

The selection of a handful of tools does not signify process enforcement. But if the selection includes unit testing tools then some level of unit testing is enforced. The enforcement of the more strict process methodologies evolve from a simultaneous use of tool selection and template and light process the way it was presented in Section 7.1.

Activity	Sub Activity	Tool
Project management	Process management	Mingle
	Resource management and requirements prioritization	Mingle
	Process visibility	Mingle
Communication	Persistent documentation	OpenOffice.org write, eclipseIDE, GIMP
	On demand documentation	Mingle
	Direct communication	E-mail, skype (voip and instant messaging)
Development activities	Writing software source code	Subversion, Eclipse
	Deployment	Maven, Nexus
	Quality assurance	Xunit, FitNesse, Jmeter, Selenium

Table 14: Tool selection table after applying the IPIDDDT method

8.7 IPIDDDT practices after tool selection

The tool selection alone can not enable and enforce the software development practices that the target company strives for. A way to further enable and enforce practices was presented in Section 7.2.5. The select software development tools need to be integrated, configured, and managed to further enable and enforce software development practices.

The integration, configuration, and management of the tool set could probably be done using the Maven build tools and the Nexus repository manager, another more complicated file management solution, some custom made tools, or a combination of the previous. The actual implementation of these process steps, is not within the scope of this study, for this theoretical case.

9 Conclusions

9.1 Conclusions on the IPIDDDT method

The IPIDDDT implementation process, that was presented in Section 7.2, was used, in Chapter 8 for a theoretical case, to illustrate the use of the IPIDDDT method for a company. The implementation was done on a fairly high level of abstraction and the tools that were selected where examples of their tool class more than actual selected tools. The selection was easy to do with the high level criteria. The general criteria would probably be more complex and specific in a real world scenario.

The selected tools that were placed in the selection pools where logical and tools that could well be seen in these kinds of scenarios. The final selection was done according to the rationale presented and supported the selected methods. The selection method has no clear problems from the point of view of the theoretical implementation case.

9.2 Further studies

The IPIDDDT method principles were presented on a high theoretical level but only the tool selection was addressed in more detail. An important further study would go into the actual modification of the development tools as well as the management of those tools and changes in repositories. These two issues are more concrete, and related to the actual software development tools, and the issues are outside the scope of this thesis, but the extent of possible modification as well as the ease of management of the tool and change repositories is a question for the success of the IPIDDDT method.

An important next step is to do an actual case implementation, where the actual problems of the solution would be solved. The method has yet only been proposed and sketched out, a further implementation test would probably call for modifications in the actual method as well as the implementation process.

9.3 The possible significance of the IPIDDDT method

The IPIDDDT method is not revolutionary in what it does. To select tools and to customize those tools for use with a software development practice like test-driven development is not uncommon. The revolution is in the methodical inversion of the order and prioritization the process selection and improvement effort.

There are two distinct methods that are good to contrast with the IPIDDDT method. The first method is the method to adapt a software development practice in a company. 1) A software development practice that is considered beneficial is identified. 2) The practice is adapted to the target company and the company's tools, new tools are incorporated if needed.

The second method is the traditional software development company consensus for ordering of events in the case of implementing a software process and software process improvement. 1) A process engineering process is used to create a software process. 2) The company practices and tools are adapted to the new process. 3) The process is enacted in the company. 4) Process metrics are gathered and a process improvement process is initiated.

The IPIDDDT method proposes a third way in the subsection of the two methods presented above. The method has the following steps. 1) Define attainable process benefits. 2) Choose practices that support reaching those benefits. 3) Choose the tool set of the company to most effectively support and enable the selected practices. 4) Improve the implicit process (tools, configurations and templates) constantly.

A few main differences between the three methods are presented below in Table 15. The first method for adapting a single software development practice is under the heading “Practice”, the implementation of a software process methodology is under the heading “Process”, and the IPIDDDT methodology of using tools to implement some practices that provide process rewards is under the heading of “IPIDDDT”.

Practice	Process	IPIDDDT
“We change how we do something and it results in some change somewhere.”	“We change everything we do and do it in a structured way and everything improves.”	“We strive for improvements in a few key areas and do an methodical effort to gain the benefits.”
Small emphasis on tools	Varying emphasis on tools	Large emphasis on tools
Cheap	Expensive	Cheap

Table 15: Key differences in typical approaches and IPIDDDT to process benefits

The IPIDDDT model is very strong theoretically when comparing the basic models of software development in Table 15 for the very small company, and the questions and answers in Section 6.2 are real and acute in the world of small scale software development. Hopefully this contribution can help create solutions to small companies that enable them to create better quality with a smaller effort, and enable, for instance, agile development benefits for companies with projects that last less then one iteration.

Bibliography

Abrahamsson et al. 2002: P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta, Agile software development methods, Review and analysis, 2002, VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O. Box 2000, FIN-02044 VTT, Finland

Ambler, 2010: S. W. Ambler, Agile/Lean Documentation: Strategies for Agile Software Development, 2010, http://www.agilemodeling.com/essays/agileDocumentation.htm, 8/15/2010

Agile Manifesto, 2001: K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, et al., Manifesto for Agile Software Development, 2001, http://agilemanifesto.org/, 9/15/2010

Barnett and Schwaber, 2004: L. Barnett, C. Schwaber, Agile Development Teams Need Tools, Too, 2004, Forrester Research, Inc., 400 Technology Square, Cambridge, MA 02139 USA

Brugge and Dutoit, 2004: B. Brugge, A. Dutoit, Object-Oriented Software Engineering Using UML, Patterns,and Java, second edition., 2004, Prentice Hall, NY, USA.

Byrnes and Phillips, 1996: P. Byrnes, M. Phillips, Software Capability Evaluation, Version 3.0, Method Description, 1996, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania 15213

Cockburn, 2004: A. Cockburn, What the Agile Toolbox Contains, Cross Talk, November, 2004, http://alistair.cockburn.us/What+the+agile+toolbox+contains, 8/15/2010

Coleman Dangle et al., 2005: K. Coleman Dangle, P. Larsen, M. Shaw, M. V. Zelkowitz, Software Process Improvement in Small Organizations: A Case Study, IEEE Software, November/December, p. 68-75, 2005

Feiler and Humphrey, 1993: P. H. Feiler, W. S. Humphrey, Software Process Development and Enactment: Concepts and Definitions, 1993, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania 15213

Firth et al., 1987: R. Firth, V. Mosley, R. Pethia, L. Roberts, W. Wood, A Guide to the Classification and Assessment of Software Engineering Tools, 1987, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania 15213

Fowler, 2006 (2): M. Fowler, Continuous Integration, 2006, martinfowler.com/articles/continuousIntegration.html, 8/22/2010

IEEE std. 1074-1997: IEEE, IEEE Standard for DevelopingSoftware Life Cycle Processes, 1997, The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017-2394, USA

Larman and Vodde, 2009: C. Larman, V. Vodde, Lean primer, Version 1.5, 2009, www.leanprimer.com

Royce, 1970: W. Royce, Managing the Development of Large Software Systems, Proceedings, IEEE WESCON, August, p. 1-9, 1970, The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017-2394, USA

Rozum, 1993: J. A. Rozum, Concepts on Measuring the Benefits of Software Process Improvements, 1993, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania 15213

Schwaber, 2006: C. Schwaber, The Changing Face Of Application Life-Cycle Management, 2006, Forrester Research, Inc., 400 Technology Square, Cambridge, MA 02139 USA

Schwaber, 2007: C. Schwaber, The Truth About Agile Processes, 2007, Forrester Research, Inc., 400 Technology Square, Cambridge, MA 02139 USA

Shaikh et al. 2009: A. Shaikh, A. Ahmed, N. Memon, M. Memon, Strengths and Weaknesses of Maturity Driven Process Improvement Effort, International Conference on Complex, Intelligent and Software Intensive Systems, 481 - 486, 2009

Simons, 2002: M. Simons, How to Succeed at Offshore Agile Development, 2002, http://www.informit.com/articles/article.aspx?p=25929&seqNum=5, 5/15/2010

SPEM 2.0: Object Management Group, Software & Systems Process Engineering Meta-Model Specification version 2.0, 2008, http://www.omg.org/spec/SPEM/2.0/PDF

Zhu et al., 2006: H. Zhu, M. Zhou, P. Seguin, Supporting Software Development With Roles, IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans, Vol. 36, November, 2006