2013 46th Hawaii International Conference on System Sciences

A Comparative Analysis of Traditional Software Engineering and Agile Software Development

Ashley Aitken
School of Information Systems
Curtin University – Perth, Australia
A.Aitken@Curtin.Edu.Au

Vishnu Ilango
School of Information Systems
Curtin University – Perth, Australia
VishnuIlango@gmail.com

Abstract

Over the last decade (or two) the pendulum of developer mindshare has swung decidedly towards agile software development from a more traditional engineering approach to software development. To ascertain the essential differences and any possible incompatibilities between these two software development paradigms this research investigates a number of traditional and agile methodologies, methods, and techniques. The essential differences between traditional software engineering and agile software development are found not to be (as one may first suspect from a cursory consideration) related to iteration length or project management, but rather more related to other attributes like the variety of models employed, the purpose of the models, and the approach to modeling. In the end though the two approaches are not seen to be incompatible, leading to the future possibility of an Agile Software Engineering (ASE).

1. Introduction

Software development has gone through a significant change in the last decade or two. The arrival late in the 20th Century of Extreme Programming (XP) represented the emergence (but not the birth) of a new approach that would become Agile Software Development (ASD). The history of software development is, of course, much longer and probably even pre-dates the first electronic digital computers that arrived in the 1940s. After an initial period when it was considered to be a craft, software development was commonly thought of as a kind of engineering. ASD seems to provide a new alternative (somewhat craft-like but more developed). However, there are still those who think of ASD as a form of software engineering.

This seems somewhat problematic. Either there has been a significant change in approach and ASD is not software engineering or the notion of what is currently understood to be software engineering is different to what was previously understood to be software engineering. To make this clear this paper shall use the term Traditional Software Engineering (TSE) to specifically refer to that form of software development that took an engineering approach to software development as it was understood prior to the arrival of ASD.

It is thus interesting to consider (without value judgment) what has changed during this paradigm shift. The question that this research addresses is what are the essential similarities and differences between ASD and TSE and do these make them incompatible. It addresses this question by undertaking an investigation and review of a number of software development methodologies, methods, and techniques within the TSE and ASD paradigms. It considers a set of attributes of each of these to ascertain what, if any, are the essential differences between TSE and ASD. The research involves a comparative analysis of commonly known software development methodologies, methods, and techniques as presented in the literature and used in industry. It also considers if there is room for a common ground between traditional software engineering and agile software development. It does not aim to be complete in its coverage or analysis but aims to focus on popular methodologies, methods, and techniques and the essential similarities and essential differences.

The contribution of this paper is two fold: 1) it provides a brief overview and introduction to a range of software development methodologies, methods, and techniques, and 2) it provides a discussion of these methodologies, methods, and techniques in relation to TSE and ASD. Other research has performed comparative studies of different agile software development approaches [1, 2] but none other seems to have compared TSE and ASD with the aim to ascertain the essential similarities and differences between the two approaches. The research is limited by the somewhat ad-hoc nature of the data gathered and the fact that it is not complete or comprehensive (e.g. it does not consider testing or other verification and validation tasks).

The next (second) section of this paper discusses software development in general, including phases of development, software development lifecycles, and approaches to software development. In the following section it provides an introduction to a number of TSE and ASD methodologies, methods, and techniques. In the fourth and penultimate section the paper provides an analysis and discussion of various aspects of the methodologies, methods, and techniques. The final section provides a summary of the findings and some conclusions. The results may not be definitive or comprehensive but they do provide a starting point for further discussion and research into the relationship between ASD and TSE.

1530-1605/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2013.31

2. Software Development

Software development has always involved problem solving. This section briefly investigates the nature of problem solving, the software development lifecycle and, finally, various paradigms in software development.

2.1 Software Development Phases

No matter how software development is performed or what approach is taken the essential task involved is, as mentioned above, problem solving. The way developers solve problems is generally the same no matter what the problem is or the approach taken. Problem solving involves four essential activities: requirements – gathering and documenting details about the problem; analysis – understanding the problem in enough detail to ensure a correct solution; design – finding and specifying an optimal solution to the problem; and implementation (if needed) – implementing the solution in whatever form it takes.

The essential problem in software development is how to implement, using certain technologies and within certain constraints, a particular information processing system.[1] Although there are associated problems of understanding the domain these are generally non-software related. It can be argued that no matter what paradigm or approach is taken to software development each of the problem solving activities has to be undertaken to some extent. In essence, every developer goes through the requirements, analysis, design, and implementation cycle, be it over an extended period, a week, a day, an hour or minutes, and whether or not they document the results, discuss them with others on a whiteboard, or just consider them informally within their head. There is no escaping these activities.

2.2 Software Development Lifecycles

A software development lifecycle (SDLC) gives a high-level perspective of how the different problem-solving activities may be worked through in phases by an individual or team doing software development. McConnell [3] provided a rather exhaustive list of SDLCs. The popular SDLCs in some form of contemporary use are: 1) Waterfall, 2) Iterative, 3) Iterative and Incremental, 4) Evolutionary Prototyping, and 5) Ad-hoc or Code-And-Fix SDLC.

2.2.1 Waterfall. The Waterfall SDLC was presented by Royce [4] as a method for teaching software development. It involves sequentially completing each phase in full and then moving on to the next phase. It was never meant to be a practically useful approach to software development. The idea that requirements can be totally completed, and then analysis totally completed, and then design totally completed and then implementation totally completed, is practically impossible [5]. It does not recognize that learning occurs during development resulting in the need to modify earlier deliverables. In reality, some projects aim to approximate the Waterfall SDLC, with as little rework or correction as possible (e.g. Contractual Waterfall).

2.2.2 Iterative. The Iterative SDLC involves sequential attention to each phase in problem solving but allows for developers to go back and repeat the sequence again, to further the results of each activity. The Iterative SDLC matches more closely the real world in that it recognizes the need to go back and modify the results of earlier phases, and it recognizes that software developers can never really completely finish any activity. In this pure form, the Iterative SDLC sees developers trying to refine a complete solution to the problem through each cycle. Since the focus is on producing a complete solution, usually many iterations would be needed before that is achieved, and when to stop iterating is problematic with any scope creep.

2.2.3 Iterative and Incremental. The Iterative and Incremental SDLC is a special case of the Iterative SDLC in which the aim is not iterate on the complete scope of the problem but rather on a small subset of the scope. Iterations will usually continue until this subset is adequately addressed. Like the basic Iterative SDLC, the Iterative and Incremental SDLC, has nothing to say specifically about the duration of the iterations or the time spent in each activity – just that some requirements, some analysis, some design, and some implementation are repeated in that general order. The Iterative and Incremental SDLC is the current best practice for software development.

2.2.4 Evolutionary Prototyping. The Evolutionary Prototyping SDLC involves iterative and incremental development but in the case where the final destination for the development is not well defined. It is for cases where the developers and the stakeholders are exploring possible solutions. The developers use an iterative and incremental approach to build a prototype and then seek feedback to undertake further development in a possibly different direction. The emphasis is on building, sometimes quick and dirty implementations of an idea, to test it out and determine if it is worth further investigation. The notion of a prototype should mean that when the final destination is found the prototype is discarded and a similar system is built with more of a focus on quality. However, this is often difficult to do when the customer has seen (and is willing to use and pay for) working software.

2.2.5 Ad-hoc (or Code-And-Fix). Whereas the group of iterative SDLCs (2-4 above) generally involves iterating through the different problem solving activities, the Ad-Hoc SDLC allows any combination and variation of these. It is meant to describe the SDLC for an individual or group of software developers working in an ad-hoc fashion, i.e.

[1] This is not always understood by software developers or managers who sometimes perceive (at least part of) the task of software developers is to solve the domain problem.

not following any prescribed approach to the problem solving activities [6]. In such a case, a software developer may on one day spend a few hours in each activity, or on another instance spend a few minutes in each activity, or, generally speaking, any variation in between. The activities besides coding may be done informally (with pencil and paper, on a white board, even just inside ones head) or more formally (using various notations), and may or not be persisted as a part of the project. The typical "hacker" could, for example, be said to be following the Ad-hoc Iterative and Incremental SDLC.

2.3 Software Development Paradigms

A software development paradigm is a general approach to undertaking software development, a way of thinking about software development, and a metaphor for software development. The three main paradigms for software development considered here are: 1) software development as a craft, 2) software development as engineering, and 3) software development as an agile (and lean) practice.[2]

2.3.1 Software Development. When software development started out there were those who became skilled at it through experience and craftsmanship. As a result, software development was seen as a craft and those who could do it effectively as craftsmen (and craftswomen). There was an emphasis on techniques as opposed to predefined method or methodologies (perhaps because no-one had spent much time thinking about them as yet). The craft was passed on to newcomers through an apprenticeship where they would learn from the master(s) and eventually become skilled and experienced in the craft. This way of learning a craft is not the most effective or efficient way of transferring knowledge and skills – a craftsman can only mentor so many new software developers and the process of learning is slow.

2.3.2 Software Engineering. As a result of a significant increase in the capability of computers and the arrival of high-level programming languages, some software developers found the craft no longer able to keep up with increased scope and demands of software development. There was a demand for more complex software and much larger software systems that could not be developed by individual craftsmen. These software projects were often delivered late (if at all), over budget (often significantly), and did not meet all the requirements (and often had significant bugs) [8]. This situation was generally referred to as the "software development crisis."

As a result a new paradigm for software development gained favor at the NATO conference in 1968 [9]. It considered software development as a form of engineering (e.g. software engineering was to computer science as civil engineering was to physics). The hope was (and, for many, still is) is that if software development can develop as an engineering discipline then it would be possible to develop complex and large software development projects on-time, on-budget, with fewer bugs, and to meet most (if not all) of the requirements. Engineering is generally thought of as processes for using knowledge to achieve objectives, usually in building (or at least designing) complex systems or structures. Engineers put a great deal of effort into predicting and planning their work and generally work with fixed requirements, i.e. they know what they are supposed to build at the commencement of the project (e.g. a bridge or a road). Engineers also use many models as they work.

There are many who disagree with this perspective of software development as a form of engineering [10] [11]. They claim that software development is not a form of engineering because it is more complex, more malleable, and that designing software is much more of a creative task. Whilst it is true there is more complexity in software it is important to separate the domain complexity from the mapping of the domain onto the chosen development technologies to meet the non-functional requirements (NFRs). Similarly those who say every piece of software is quite unique should remember that software development is not a domain problem but a translation problem. It is also argued that software development is not engineering because the compiler essentially performs the build step in software development. This is confusing the bricklaying with the design of the building. The essence of software development is design, and design is making this code meet the non-functional requirements, not finding a solution to a domain problem. Software developers, even those who just code, are not just laying bricks, they are investigating the requirements, analyzing the problem, and designing a solution (often at the same time in their head).

2.3.3 Agile Software Development. The agile software development approach [12] [13] [14] (that also incorporates many aspects of lean software development [15] [16] [17]) is characterized by taking an adaptive approach to change within software development. That is, software development is undertaken in a way that makes responding to change perhaps less difficult and less expensive that would be in an engineering approach. For example, agile software development keeps the development process and artifacts as light (and minimal) as possible, so not as to require rework. Agile is also about having a people (developer) focus, removing waste (the lean contribution), and collaboration rather than contractual negotiation between developers and client. Rather that seeing software development as a form of engineering, Cockburn [18] sees it as a co-operative game.

ASD is a "broad church" and can mean many things to different people. It is broadly defined by the "Agile Manifesto" – a manifesto and set of values and principles declared by a group of leading software developers. The values proposed in the Agile Manifesto [19] are:

[2] Another is "software development as gardening" [7] Hunt, A. and Thomas, D. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

ASD is also thought of as an approach to software development that focuses on incrementally and iteratively adding business value, with the process of how the software is developed being be left up to the team who are responsible for its development. It also adopts the lean focus on reducing waste and only doing activities or creating artifacts that directly add value. ASD has become very popular with 76% of organizations reporting in 2009 that they have adopted agile techniques [20].

3. Methodologies, Methods, and Techniques

For this research a number of different methodologies, methods, and techniques for developing software were reviewed and characterized according to a number of different attributes (e.g. history, process, artifacts, iteration length, project management, scope within SDLC and applicability in different contexts). Methodologies, methods, and techniques were distinguished as follows:

Techniques – a way of doing some aspect or part of software development [18]. They may relate to technical (e.g. coding or testing) or non-technical (e.g. management) aspects of software development.

Methods – a process, however general or specific, and a notation (set of deliverables or artifacts, including documentation and source code) [21]. A method usually includes a number of different techniques as well.

Methodologies – a collection of methods or a framework for determining a method [18]. Usually a methodology cannot be used as is but needs to be customized to fit the project and other constraints.

3.1 Methodologies

This section will consider only two methodologies for software development – the Unified Process Methodology and the Crystal Family Methodology. The former is more commonly associated (but not exclusively) with TSE and the latter with ASD.

3.1.1 Unified Process Methodology. The Unified Process (UP) is a generic name for the IBM Rational Unified Process [22, 23], which is a commercial process developed originally by Rational Corporation and then acquired by IBM (when they purchased Rational). The UP was developed as a process to accompany the Unified Modeling Language (UML), a set of mostly graphic models covering all phases of software development. The UP is large and extensive (like the UML) containing many processes and techniques to help with the development of many different types of software. Overall, the UP is described as an iterative and incremental, architecture-centric and use-case driven methodology. If anything it is this focus on textual and graphic models and upfront architecture that differentiates UP from more agile methods.

3.1.2 Crystal Family Methodology. The Crystal Family (CF) of methods was developed by Alistair Cockburn in 1992 [12]. The methods within the family are chosen based on several factors such as team size, system criticality and project priorities. It aims to be human-powered, ultra light, and stretch-to-fit. Methods within the Crystal Family include Crystal Clear, Crystal Yellow, Crystal Orange and Crystal Red with larger projects requiring more coordination and heavier methodologies than smaller ones. The general process focuses on iterative and incremental development with delivery on a regular basis, a focus on software deliveries as milestones (rather than written documents), direct user involvement, automated regression teaching, two viewing per release, and workshops for product and method turning at the beginning and in the middle of each increment. Key tenets include teamwork, communication, and simplicity as well as frequent reflection for process improvement. The Crystal Family promote frequent delivery of working software, high user involvement, adaptability and the removal of bureaucracy or distractions, but have a relatively restricted communication structure that suits a single co-located team.

3.2 Methods

This section provides an overview of a range of different (but predominantly ASD) methods.

3.2.1 Extreme Programming Method. Extreme Programming (XP) [24] is a programming-focused software development method that intends to improve software quality and responsiveness to changing customer requirements through simplest coding solutions. Kent Beck, Ward Cunningham, and Ron Jeffries introduced XP in the latter parts of the 1990s. XP is highly iterative and incremental and includes six stages (with embedded implicit phases). It uses story cards for requirements and then primarily code-based models (for testing and implementation). Other models (e.g. analysis and design models) are primarily for communication and generally are not persisted. Iteration lengths are generally 1-4 weeks and there is no specific project management approach (although Scrum is usually used). XP is most suitable for small to medium sized project and small co-located teams. It has a number of practices, including planning game, TDD, customer involvement, refactoring, pair programming, collective ownership of code, continuous integration, sustainable pace and coding standards. It is a coder and code-focused method that appeals to coders but relies on highly skilled developer. It provides working software in short cycles but it is more difficult to scale to large projects or to sustain long-term maintenance.

3.2.2 Scrum Method. Scrum (S) is a mostly project management method that concentrates on how team

members should function in order to develop software flexibly in a constantly changing environment [1]. Scrum was introduced by Takeuchi and Nanoka in 1986 but the contemporary movement was introduced by Ken Schwaber and Jeff Sutherland in the mid 1990s [25] [26]. Scrum includes three stages pre-game, development, and post-game, with the pre-game being divided into two sub-stages called planning and architecture, and the development stage consists of a number of sprints – iterative cycles where functionality is developed or enhanced to produce new increments. Scrum does not specify the way features are specified, just that there is a list of such features (called the backlog). Iterations can last from around one week to around one month with three to eight sprints before a release. Scrum also requires customer involvement as the "product owner." Scrum is ideally suited for project with rapidly changing requirements. It helps to improve engineering practices and its flexibility allows it to adapt to changing contexts. Some notions such as "self-organizing" teams may lead to uncertainty and risks and it may not always be possible to "make the sprint."

3.2.3 Feature Driven Development. Feature Driven Development (FDD) [27] [28] is an iterative and incremental software development method that is model-centric, business requirements-driven and emphasizes quality through the process with a timely, accurate and meaningful status reporting and progress tracking for all levels of leadership. Jeff Luca, Peter Coad and Stephan Palmer developed it in 1997 as an evolution of the Coad Method. It consists of five stages: 1) develop an overall model – high-level description of the system, including requirements and domain models; 2) build a feature list based on client-valued functionality; 3) plan by feature – sequence features according to their dependencies; and then 4) design by feature and 5) build by feature, are executed iteratively for small groups of features selected from the feature list. FDD uses models for specification (as well as communication) and encourages frequent progress reporting. Iterations are usually around 2 weeks in length. According to Palmer and Felsing [29], FDD is suitable for most projects, even those requiring considerable agility. FDD offers predictability if requirements are stable but knowledge sharing may be hindered by individual class ownership within development.

3.2.4 Dynamic Systems Development Method. Dynamic Systems Development Method (DSDM) is a software development method based upon the Rapid Application Development methodology [30], the goal of which is to deliver projects on time and on budget while adapting to changing requirements [31]. This is achieved by fixing time and resources and then adjusting the functionality that can be delivered within those constraints. DSDM consists of five stages: feasibility study, business study, functional model iteration, design and build iteration and implementation, with the first two being sequential and the last three being iterative and incremental. It encourages a number of non-code-based models as well as code-based prototyping across the stages of development. Iterations are time-boxed from a few days to a few weeks. Although it works best with small teams of between two and six developers, DSDM can support many teams on a project. DSDM also requires active user involvement. It has a strong framework for design but it is difficult to see how changes can be incorporated into the non-iterative stages of development.

3.2.5 Adaptive Software Development Method. The Adaptive Software Development (ASD) method was developed by James Highsmith in 2000 [32] based on earlier methods (e.g. "RADical Software Development" co-authored with Bayer [33]). It involves iterative and incremental development with constant prototyping and encourages continuous adaptation of the process for high-speed and high-change projects. Projects iterate through three stages: speculate – a form of adaptive cycle planning, collaborate – concurrent component engineering by a team, and learn – quality review to learn from mistakes, with project initiation at commencement and final quality assurance and release at the end. It includes non-code based models (e.g. project mission and product specification outline). Iteration length and project management details are not specified. It aims to build an adaptive organizational culture for developing complex large systems.

3.2.6 Personal and Team Software Development Process. These are in fact two related methods. The Personal Software Process (PSP) [34] and the Team Software Process (TSP) [35]. Both originated from the SEI at Carnegie Mellon University. The PSP, as its name suggests, is about individual developers creating individual processes for their module-level development. Key to the PSP is each engineer refining these processes based on detailed measurements made whilst developing (e.g. time in each phase, type and number of errors, lines of code produced) to improve productivity and quality. PSP probably requires the most discipline of any methods but results can be striking [36] [37]. The TSP is a team related process that devolves responsibility to the team to plan and estimate their development work (using aggregate data from each developers PSP). The TSP shares a number of features in common with ASD but overall the PSP and TSP would be more likely considered a form of TSE.

This research also considered other methods including the Waterfall Method [4], Prototyping Method [38], Spiral Method [39], Model Driven Development [40] but do not report on them in detail here due to space restrictions.

3.3 Techniques

The research considered the following (predominantly agile) software development techniques (with many coming from [24]): Pair Programming, Behavior Driven

Development [41], Use-Case Driven Development [42], Test Driven Development [43], Acceptance Test Driven Development, Unit Test Driven Development, Planning Game, Customer Involvement, Collective Ownership, Sustainable Pace, Coding Standards, Standup Meetings, Simple Design, Refactoring, System Metaphor, Component-based Development, Time-boxed Development, Change Tolerant Development, Risk Driven Development, Customer Focus Groups, Domain Object Modeling, Developing by Feature, Empowered Teams, Frequent Delivery, Continuous Delivery, Integrated Testing, Regular Builds, Configuration Management, Continuous Integration, Continuous Delivery, Domain Drive Design [44]. The aim was to consider which of the techniques were compatible with TSE and ASD or not.

4. Comparative Analysis and Discussion

Comparing and seeking the similarities and differences between software development paradigms is fraught with problems. The first is that there is no clear or agreed upon definition of each of the paradigms. The second is that each paradigm is a broad enough to allow a wide variety of very different methodologies, methods, and techniques within their bounds. For example, the Agile Manifesto is broad enough to allow a focus on processes, tools, etc. as long as there is more of a focus on people, code, etc.

This situation makes it easy for readers to pick a part any statements claimed herein. For example, it is claimed in this paper that ASD focuses primarily on code-based models and generally does not persist non-code-based models, even the user stories. Of course, there are those who will disagree with this statement and give evidence of a particular agile method or company that does use and persist non-code-based models. Such counterclaims are, however, missing the purpose of this paper. It is attempting to seek the essential nature of traditional software engineering and agile software development and then determine the similarities and differences between them. There will always be specific variations that don't fit the essential approaches characterized here. Trying to characterize these essentials aspects is important though.

Similarly, this research tries to focus on best practices rather than actual practices (with all their variations and failings). For example, it shall explain that the best practice for TSE was a highly iterative and incremental software development lifecycle. However, many organizations seem to fallback into a pseudo waterfall or at least longer-iteration software development lifecycle. On the other side, a similar phenomenon has recently been reported in that many organizations employing ASD with Scrum seem to fallback to something called "Water-scrum-fall" [45]. In reality most companies seem to develop their own minimally prescriptive process (at an organizational level or within teams). This was the case with TSE and this is the case with ASD. There is nothing wrong with "home grown selective customizations," albeit that they are often a lot less complete and a lot less detailed than what is prescribed in either TSE or ASD methods.

4.1 Analysis of Methodologies and Methods

This research collected general data for more than a dozen different approaches, categorized them (and their components) as methodologies, methods, or just techniques, and then evaluated them according to their recommended iteration length and how much the approaches aligned with the general approaches of TSE and ASD. Two custom pictographic representations are used to summarize this data.

Figure 1 shows all of the approaches and displays iteration length, how the approach is categorized, and where it lies on a continuum between fully TSE and fully ASD. The two axes indicate the length of the iteration and the position on a spectrum between TSE and ASE. The offering of the approach is also shown by its inclusion inside the boxes for methodology, method, or techniques. Please note the position inside each box is not important, only the membership of each category is relevant. For example, some items are just techniques, some are methods with techniques and some may be methodologies with techniques.

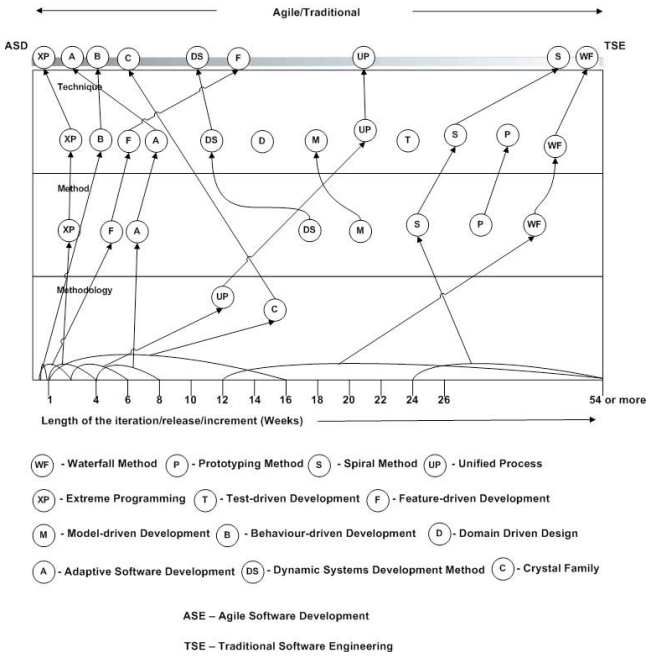

Figure 1 – Agility, Components, and Iteration Length

Figure 2 represents a summary of the information collected about all approaches with regard to the SDLC phases and which deliverables are produced within the approach (x-axis lower box), which phase are managed by the approach (x-axis upper box), and the year of introduction (y-axis). Amongst a number of things it demonstrates that whilst many of the TSE approaches were

full-lifecycle, many (but not all) of the ASD are more focused in their application.

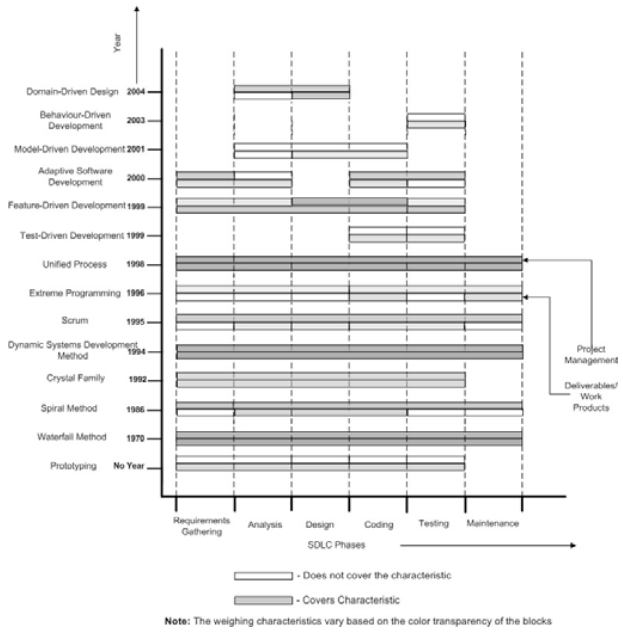

Figure 2 – SDLC Phases, Project Management, Deliverables and Year of Introduction

In the next part of this section, TSE and ASD (will be compared based on the preceding review. They will be compared according to the values and principles and characteristics of the ASD approach. Finally, the essential similarities and differences between TSE and ASD and any compatibilities (or incompatibilities) between the different paradigms will be discussed.

4.2 Comparing Values and Principles

Table 1 indicates the general way in which values and principle are perceived to differ between TSE and ASD. It is not clear, however, that this must be the case. This section will consider the values and principles of ASD to see if they are that incompatible with TSE.

Table 1 Comparing Values and Principles of TSE & ASD
Source: (Adapted from [19])

Traditional Software Engineering (TSE)	Agile Software Development (ASD)
Process and tools driven.	People and collaboration driven.
Document driven. Every activity is measured by intensive documentation or deliverables.	Code or programming driven. Working software is used as a measure for progress. Emphasizes more importance to the design of the code.
Does not involve customers continuously.	Involves customers continuously.
Does not welcome changing requirements during the projects progress. Difficult to change direction once contracts are signed.	Welcomes changing requirements during the projects progress. Easy to change direction as the requirement changes.
Does not encourage the delivery of working software to customers continuously.	Delivers working software continuously to customers
Does not generally encourage a delivery within two weeks to two months. Iterations are longer.	Encourages software delivery or release cycles every two weeks (or less) to two months (maximum). Shorter iterations.
Does not generally encourage self-organized teams.	Software can be delivered at its best from co-located teams.
Follows the phases of SDLC formally. For example, encourages formal analysis and planning.	Does not follow the phases formally. For example, does not have a formal analysis or planning phase.
All the requirements decided in the requirements gathering phase should be delivered by the final software built.	Short software deliveries for individual features or stories. The features or stories can evolve frequently.
Does not emphasize constant pace among developers and customers or users, as the users are not continuously involved throughout the project – In traditional approaches customers are involved only during pre and post delivery of the software.	Developers and customers or users should be able to maintain a constant pace indefinitely to promote sustainable development.

Consider the values from the Agile Manifesto (AM):

4.2.1 Individuals and interactions over processes and tools. Early in ASD there was a major rejection of processes and tools (generally from a revolt against the UP). However, with the ever-growing number of processes and tools associated with ASD it is not as clear now that they are always discounted compared to individuals and interactions. On the other hand teamwork and interactions have always been a part of TSE. If individuals and their interactions have not always worked so well, it is probably a fault of the project management rather than the TSE approach itself. It is not clear then that TSE is incompatible with individuals and interactions being more important than processes and tools. One could also say that TSE has always seen processes and tools as aids for individuals and their interactions (e.g. consider the swim-lanes in the UP).

4.2.2 Working software over comprehensive documentation. This value seems to use documentation as a derogative term, in comparison to code (that becomes working software). This is inappropriate because code is just as much a form of documentation as any requirements, analysis, or design document. In reality they are all models, code just happens to be the model that is generally closest to the final executable software. It is also disingenuous to suggest that models other than the code are less important or just "documentation for documentation sake." Just as the code is developed to produce the required working software, so too are the other models in TSE. TSE builds requirements, analysis and design models because it is believes these models enable this approach to build better software and to build it faster (e.g. errors found in models in earlier phases of the SDLC could be corrected with less effort and time than those found later in the SDLC).

4.2.3 Customer collaboration over contract negotiation. This value seems a little disingenuous as well,

i.e. to assume that TSE would prefer contract negotiation to collaboration with the customer. It is true that contract negotiation is common under TSE but it would be fair to say that it is common under ASD as well. Most of the business world is not ready for "time and materials" software development. If anything the ASD approach has just included two items (a third will be discussed next) into the contract. These being: 1) that they cannot guarantee completion of all requirements, but 2) that they guarantee to add value almost immediately and to continue doing that (less the contract be terminated). In reality, many TSE projects have been contracted in this manner as well, with staged delivery (of high priority requirements firstly), and out-clauses if stages were not completed satisfactorily. On the other hand TSE employed contract negotiation to protect developers from scope creep, whereas ASD manages that problem by never agreeing to deliver everything, only to continually deliver.

4.2.4 Responding to change over following a plan. It is true that TSE was generally built around planning for the entire development project. Much of this was a result of the contract negotiation discussed above. Firstly though, any plan of any real worth is a living document that changes as the project progresses. TSE may have planned an entire project but no plan survives its first encounter with the actual work and plans are almost guaranteed to change after each iteration or release. Secondly, there is no reason to believe that TSE could not work within an environment that only planned each iteration (even if they were very short iterations). Even ASD working within a negotiated contract has plans for the overall project, whether they are more lightweight and less detailed plans or not. Within the right agreement and with the right tools it should be clear that TSE could respond to change just as easy as ASD.

4.3 Comparing Methodologies, Methods, and Techniques

Comparing methodologies is not easy because by their very nature they need to be customized before they can be used, and there is often room for a large degree of variation in all aspects of the custom method. So, for example, the Unified Process (UP) methodology that is often held up to be the archetype for TSE can be customized to provide lean and agile methods, e.g. the dX method [46, 47], the Agile Unified Process [48, 49], and the Essential Unified Process [50]. So rather than compare methodologies, or even methods, directly the paper will discuss a number of different characteristics of methods. These are:

4.3.1 Project Management. There are two aspects of project management that are relevant to this comparison: 1) who manages the project, i.e. who is responsible for the success of the project, and 2) what management techniques are used. TSE is often portrayed as having top-down project management, e.g. with a dedicated Project Manager role, whereas ASD is portrayed as devolving responsibility to the team (perhaps with the help of coaches and Masters.) The latter form of project management is definitely not restricted to ASD though. The TSP was strong on making the team responsible for managing the development, from estimating schedules to getting the job done. Many methods (ASD and TSE) now empower teams to take responsibility for planning iterations (or releases) and delivering on those plans.

4.3.2 Iteration Length. Ostensibly a highly iterative and incremental software development lifecycle seems to be one of the defining characteristics of ASD. This is also probably due to the fact that ASD is often compared with a Waterfall SDLC (often attributed to TSE). Unfortunately, this is somewhat of a straw-man comparison. Most (if not all) contemporary software development methods employ iterative and incremental SDLCs. That said, just as TSE development often cannot meet best practice and falls back to pseudo waterfall, so too can ASD fall back to "Water-Scrum-fall" as mentioned earlier. So the notion that ASD differs from TSE in that it is highly iterative (i.e. iteration lengths are from a few weeks to a few months) whereas TSE is predominantly waterfall is incorrect. Even the poster-child for TSE methods, the Unified Process, suggests iteration lengths of a few weeks to a month or so.

4.3.3 Functional Increments. TSE has allowed any form of increment within development, e.g. components or horizontal slices of functionality in layers of the software architecture. ASD has been emphatic about the need for vertical increments of functionality that complete a user story. The reason for this is two fold: 1) to deliver directly usable value to the customer, and 2) to ensure that no waste is produced in over-engineering or poorly engineering any part of the solution. The only code that is developed in each layer is that necessary to fulfill the current user story. This may be a more lean approach but it is not clear that it leads to the most effective design of each layer, at least not in the most direct fashion. Clearly, TSE is not incompatible with the development of vertical slices if desired. TSE, however, can proceed with adding value in components or horizontal layers even if they are not visible directly to the user.

4.3.4 Models (aka Documentation). TSE has been committed to developing models within all phases of software development. Whilst these have traditionally been in the form of documentation (including figures and text), this is not the only way to build models. As mentioned earlier, TSE believes that constructing these models before implementing the code can find errors earlier and correct them with less effort. ASD (as evidenced in a number of the methods, e.g. DSDM) are not against additional models, just that the most popular methods prefer code to text and figures and prefer not to persist anything but code. Also in ASD temporary and fragmented models are generally used to aid communication between individuals and within the

team rather than to specify the system in an alternative way (or within a different phase) as is the case in TSE.

4.3.5 Architecture. TSE processes almost always involve upfront design of the architecture. This is often characterized by ASD as "big upfront architecture (or design)" and made to sound synonymous to waste and even poor design. However, it is not necessary for the design of the whole software application to be completed up front. The claim by ASD that architecture emerges is also hard to comprehend. Big decisions like the approach to persistence or a multi-tiered or multi-layered architecture need to be made some time early in the software development project, since they are central to the structure of the solution. ASD claims that such architectural decisions are made just in time in ASD whereas often too early in TSE.

4.3.6 Techniques. Comparing the applicability of ASD techniques to TSE is also interesting. For this research a list of techniques (described in the previous section) that are predominantly associated with agile software development were investigated. For each technique it was considered whether the technique was only useful with ASD or TSE or could successfully be used within both paradigm. It was found that nearly every practice that is associated with ASD could be applied in the context of TSE. This is to say that these practices are not really defining differences between ASD and TSE. The only standout technique is component-based development (CBD), which is not really compatible with ASD's focus on vertical feature development, i.e. development of a particular feature across all layers in an application, as opposed to focusing on a particular component or subsystem within one layer.

5. Summary and Future Work

This research has reviewed agile software development (ASD) methodologies, methods and technique against a backdrop of traditional software engineering (TSE). The goal of this research was to try to ascertain if these approaches to software development were compatible by considering their similarities and differences. It has found that in areas where one would, perhaps, initially assume there are differences (e.g. in iteration length because it is so often mentioned) TSE and ASD are very similar. Where they do seem to differ is subtler (e.g. in the use of models).

TSE is not incompatible with individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, or responding to change over following a plan. TSE encompasses the belief that creating (and persisting) requirements, analysis, and design models will improve all project outcomes (like time, budget, functionality, and quality). Similarly, ASD does not seem to be against further modeling or non-code models, even persisting those models, as long as they add value. It embraces other models, particularly for example various testing activities (although these are still often in code). ASD has revolutionized the management of software development, although a number of the techniques have been used previously in TSE (e.g. in TSP).

TSE and ASD as best practiced are similar in their pursuit of short iteration lengths. They both have to employ all the phases of software development, even if more informally and without "documentation" within ASD. Both seek significant customer involvement, although where ASD demands it, TSE has generally just desired it (and both often struggle to achieve it). Both develop models (aka documentation) for communication between individuals (or within teams). And, finally, both aim to deliver value all the time though the form of this value may differ.

The differences start with the fact that ASD requires that value delivered is directly visible to the user, whereas TSE is generally happy with value that is not immediately visible to the user (e.g. in components or hidden layers). ASD is better at only modeling to a level of detail that is needed, whereas TSE generally tries to build complete models (iteratively). TSE likes to make as much knowledge about the problem and solution explicit within the models, whereas ASD, generally speaking, is happy for that knowledge to live within developer's heads (and indirectly in the code). ASD supports and encourages emergent and evolving architecture whereas TSE encourage upfront architecture (with justification and evaluation of options). Most models that persist in ASD are often code-based, whereas TSE encourages many models for specification.

In the end it seems that there is nothing really incompatible with applying all the principles and values of ASD, along with most (if not all) of the practices, to TSE. The only reason it would seem that ASD would be unhappy with multiple models for specification (not just communication) is if they cannot be shown to improve the efficiency and effectiveness of the overall software development. So, it seems there is room and the possibility for a merger of these two approaches, perhaps an Agile Software Engineering (ASE). Future research will be needed to consider what would be required to make this a reality.

6. References

[1] Pekka, A., Outi, S., Jussi, R. and Juhani, W. *Agile Software Development Methods - Review and Analysis*. VTT Publications, City, 2002.

[2] Awad, M. A. *A comparison between agile and traditional Software Development Technologies*. The University of Western Australia, 2005.

[3] McConnell, S. *Rapid development: Taming wild software schedules*. Microsoft Press Redmond, WA, USA, 1996.

[4] Royce, W. W. *Managing the development of large software systems*. City, 1970.

[5] Laplante, P. A. and Neill, C. J. The demise of the waterfall model is imminent and other urban myths. *ACM Queue*, 1, 10 2004), 10-15.

[6] McConnell, S. and Tripp, L. Guest Editor's Introduction: Professional Software Engineering - Fact or Fiction? *IEEE Software*(November/December 1999).
[7] Hunt, A. and Thomas, D. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.
[8] Gibbs, W. W. Software's chronic crisis. *Scientific American*, 271, 3 (1994), 72-81.
[9] Naur, P. and Randell, B. Software Engineering: Report of a conference sponsored by the NATO Science Committee (7–11 Oct. 1968), Garmisch, Germany. *Brussels, Scientific Affairs Division, NATO* (1969).
[10] DeMarco, T. Software Engineering: An idea whose time has come and gone. *IEEE Software*, 26, 4 (2009), 95-96.
[11] Loka, R. R. Software development: What is the problem? *Computer*, 40, 2 (2007), 112-111.
[12] Cockburn, A. and Highsmith, J. *Agile software development*. People's Posts & Telecommunications Publishing House, 2003.
[13] Cohen, D., Lindvall, M. and Costa, P. *Agile software development*. 2003.
[14] Martin, R. C. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
[15] Poppendieck, M. and Poppendieck, T. *Lean software development: An agile toolkit*. Addison-Wesley Professional, 2003.
[16] Poppendieck, M. and Poppendieck, T. *Implementing Lean Software Development: From Concept to Cash (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2006.
[17] Ebert, C., Abrahamsson, P. and Oza, N. Lean Software Development. *IEEE Software*2012), 22-25.
[18] Cockburn, A. *Agile Software Development*. City, 2000-2001.
[19] Beck, K., Beedle, M., Bennekum, A. v., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D. *Manifesto for Agile Software Development*. City, 2001.
[20] Ambler, S. W. *Agility at Scale: Become as agile as you can be*. 2009.
[21] Paige, R. F. When are methods complementary? *Information and Software Technology*, 41, 3 (1999), 157-162.
[22] RationalSoftware Rational Unified Process: Best Practices for Software Development Teams (1998).
[23] Kruchten, P. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.
[24] Beck, K. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.
[25] Schwaber, K. SCRUM Development Process (1987).
[26] Schwaber, K. and Sutherland, J. What is Scrum. *URL:* http://www.scrumalliance.org/system/resource/file/275/whatIsScrum.pdf2007).
[27] Coad, P., Lefebvre, E. and De Luca, J. *Java modeling in color with UML: Enterprise components and process*. Prentice Hall, 1999.
[28] Anderson, D. J. Feature-Driven Development (2004).
[29] Palmer, S. R. and Felsing, M. *A practical guide to feature-driven development*. Pearson Education, 2001.
[30] Martin, J. *Rapid application development*. Macmillan Publishing Co., Inc., 1991.
[31] Stapleton, J. *DSDM, dynamic systems development method: the method in practice*. Addison-Wesley Professional, 1997.
[32] Highsmith, J. A. *Adaptive software development*. Dorset House, 2000.
[33] Bayer, S. and Highsmith, J. RADical software development. *American Programmer*, 7 (1994), 35-35.
[34] Humphrey, W. *Introduction to the Personal Software Process*. Addison-Wesley, Reading, MA, 1997.
[35] Humphrey, W. S. Team Software Process (2000).
[36] Humphrey, W. S. Using a defined and measured personal software process. *Software, IEEE*, 13, 3 1996), 77-88.
[37] Humphrey, W. *A Discipline for Software Engineering*. Addison-Wesley, Sydney, 1995.
[38] Naumann, J. Prototyping: The New Paradigm for Systems Development. *Management information systems quarterly*, 6, 3 1982), 29-44.
[39] Boehm, B. A spiral model of software development and enhancement. SIGSOFT Softw. Eng. *Notes*, 11, 4 1986), 14-24.
[40] Beydeda, S., Book, M. and Gruhn, V. *Model-driven software development*. Springer Verlag, 2005.
[41] North, C. *Intoducing BDD*. Dannorth.net, City, 2006-2011.
[42] Jacobson, I. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.
[43] Janzen, D. and Saiedian, H. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38, 9 (2005), 43-50.
[44] Evans, E. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
[45] West, D., Gilpin, M., Grant, T. and Anderson, A. *Water-Scrum-Fall is the reality of agile for most organizations today*. 2011.
[46] Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J. and Houston, K. *Object-oriented analysis and design with applications*. Addison-Wesley Professional, 2007.
[47] Ambler, S. W. *Skinnier RUP*. City, 2006.
[48] Ambler, S., Nalbone, J. and Vizdos, M. *Enterprise unified process, the: extending the rational unified process*. Prentice Hall Press, 2005.
[49] Van Baelen, H. *Agile (Unified Process)*. 2011.
[50] Jacobson, I. *EssUP: The Essential Unified Process–An introduction*. City.