Agile Testing in a Plan-Driven Process

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Master Thesis

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in International Information Systems

at Friedrich-Alexander-Universität Erlangen-Nürnberg

Submitted by:	Surabhi Vohra
Advisors:	Prof. Dr. Dirk Riehle MSc. Ramon Anger Dipl. Inf.. Stefan Eberlein
Submitted on:	30th September, 2013

Friedrich-Alexander-University Erlangen-Nuremberg
Department of Computer Science
Professorship for Open Source Software

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Nürnberg, den 30. Sept 2013

License

This work is licensed under the Creative Commons Attribute 3.0 Unported license (CC-BY 3.0 Unported), see http://creativecommons.org/licenses/by/3.0/deed.en_US

Nürnberg, den 30. Sept 2013

Acknowledgements

I would like to acknowledge Prof. Dr. Dirk Riehle for his guidance in the research work while writing this thesis. I would also like to thank him for introducing me to Capgemini.

In addition, I would like to express my gratitude towards Capgemini employees for their valuable time and inputs for case studies. In particular, I wish to express my sincere thanks to Mr. Ramon Anger and Mr. Stefan Eberlein for their continuous guidance and feedback. Without their supervision and expert opinions, this thesis wouldn’t have been possible. I would also like to extend my gratitude towards Mr. Krisztian Edöcs for his suggestions and comments during presentations at Capgemini.

Abstract

Software firms are aiming to reduce software development costs and improve software quality at the same time. Customer satisfaction is very important in today's competitive market. Different software development approaches guide companies in software development but none can independently provide the ideal solution.

This thesis, written in collaboration with Capgemini Germany, attempts to combine traditional software development approaches with agile methods. Agile testing practices have been suggested to be incorporated inside traditional software development approaches with the focus on V-model. Extended V-models with agile testing methods embedded in them have been proposed and analyzed for their pros and cons. Four hypotheses emphasizing benefits of agile testing methods inside V-model have been formulated. Extended V-models and hypotheses are validated against three case studies. Three case studies i.e. three projects using agile testing methods inside V-model have been presented and compared for weaknesses and strengths.

The results of research and case studies' analysis indicate that Scrum can be used as a framework inside V-model, within which other agile testing methods like Feature Driven Development (FDD), Test Driven Development (TDD), Acceptance Test Driven Development (ATDD) or Specification by example etc. can be embedded. FDD was found to be least agile and best candidate for adoption inside V-model, followed by ATDD and TDD. The hybrid approach combining agile testing and traditional methods provides a balance between agility and stability.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ATDD	Acceptance Test Driven Development
BRS	Business Requirements Specification
CRACK	Collaborative, Representative, Authorized, Committed and Knowledgeable
DSDM	Dynamic Systems Development Method
ESB	Enterprise Service Bus
FDD	Feature Driven Development
ISTQB	International Software Testing Qualifications Board
JIT	Just-In-Time
LSD	Lean Software Development
NSIT	National Institute of Standards and Technology
OMG	Object Management Group
OOPSLA	Object-Oriented Programming, Systems, Languages & Applications
QA	Quality Assurance
ROI	Return On Investment
SDLC	Software Development Lifecycle
SOA	Service Oriented Architecture
SRS	Software Requirements Specification
SWEBOK	Software Engineering Body of Knowledge
TDD	Test Driven Development
TINS	This Ist Not Scrum
XP	Extreme Programming
YAGNI	You Aren’t Gonna Need It
4-DAT	Four Dimensional Framework

1 Introduction

Software Testing can be defined as an activity which ensures that the product meets the user specified requirements. It helps demonstrate quality of the product to the stakeholders. Testing has become an integral part of Software Development lifecycle. It is done with an intention to find bugs and thus helps improve quality of the product. Developer's perspective of requirements might be different from user's perspective. Testing ensures that the product meets users' specified requirements.

Approximately 40-50% of total effort and cost is spent on testing which indicates the importance of testing in software development (Brykczynski, Meeson, & Wheeler, 1994). It is very important that software testing is conducted at the right time and in the right way.

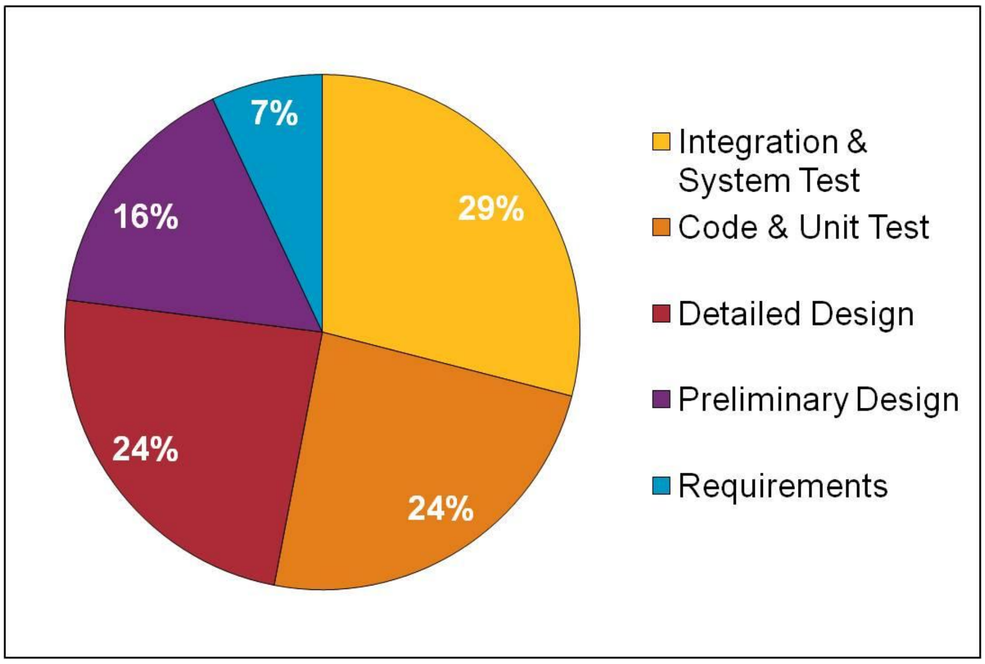


Figure 1-1. Cost of Software Development
(Adapted from: Brykczynski, Meeson, & Wheeler, 1994)

Testing can be performed at any stage of software development. However, it is mostly performed after the development or coding phase in traditional plan-driven software development. In waterfall model, testing occurs at the last stage when all the requirements have been specified and product has been developed. In V-model, which is basically an extension of waterfall model, each development activity is accompanied by a corresponding test activity. In more recent agile methods, testing is usually carried out along with the development. Waterfall model, V-model and Agile methods are explained in more detail in *chapter 2-Plan Driven Software Development Processes* and *chapter 3-Agile Software Development Processes.*

1.1 Structure of Thesis

In *chapter 1,* overall motivation for adoption of agile testing methods and practices has been provided.

Chapter 2 briefly describes traditional plan driven software development approaches. *Chapter 3* describes agile methods. It also provides the selection criteria for agile testing methods.

Chapter 4 starts with the introduction of agile testing. It then explains in detail Agile testing methods selected based upon the selection criteria presented in chapter 3.

In *chapter 5,* based upon literature review, four hypotheses have been formulated.

Chapter 6 compares agile and plan-driven approaches. It provides detailed motivation for use of hybrid approaches and also discusses challenges faced by organizations implementing Agile and hybrid approaches.

In *chapter 7,* extended V-models combining agile testing methods inside V-model have been proposed.

Chapter 8 compares extended V-models on various parameters and conclusions are drawn regarding best fit and least fit candidate for adoption inside V-model.

Chapter 9 introduces case study research and the adopted methodology for analyzing case studies.

Chapters 10, 11 and 12 present three case studies i.e. three software industry projects implementing agile testing methods inside V-model. Strengths and weaknesses of case studies have been discussed and hypotheses proposed in chapter 5 have been validated against case studies.

In *chapter 13,* three case studies are compared to each other based upon the methodology described in chapter 9.

Finally, in *chapter 14*, overall conclusions from case studies' analysis and extended V-models are drawn.

1.2 Motivation

Numerous surveys on agile methods have indicated increased adoption rate of Agile methodologies in software companies. More and more companies have started to implement agile methodologies either in pure form or in combination with plan-driven methods.

Ambysoft (February, 2008) conducted an Agile adoption rate survey in which 642 professionals were interviewed, 54.8% were developers and 29.4% were from management. 69% of the respondents indicated that their organization has adopted one or more agile technique. 31% indicated that they are not following any agile methodology but 15% indicated that they hoped to do so the same year. The results of the survey are indicated in Figure 1-2. (Ambysoft, February, 2008)

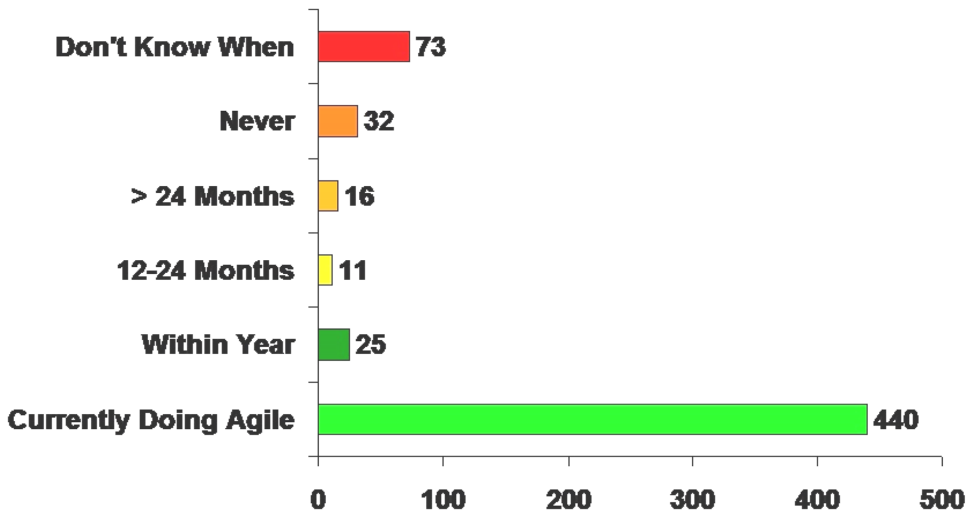


Figure 1-2. Agile Techniques Adoption Rate
(Source: Ambler, February, 2008)

Companies practicing agile methods have indicated various benefits such as reduced time to market, software development costs, planning efforts, increased customer satisfaction and so on. Software professionals have also reported higher project success rates of agile projects as compared to projects managed by traditional approaches. Another survey conducted by Ambysoft in December, 2008 indicated 70% higher project success rates of agile projects (Figure 1-3).

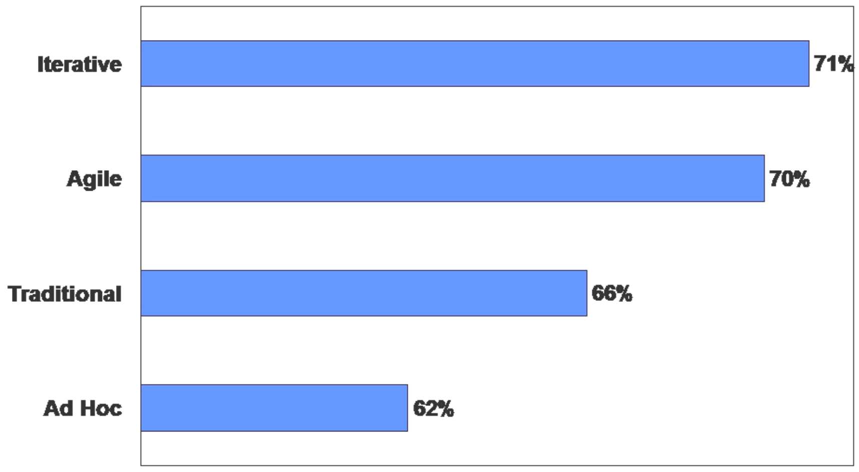


Figure 1-3. Project Success Rates by Software Development Methodology
(Source: Ambysoft, December, 2008)

The same survey indicated higher quality, better stakeholder satisfaction and improved Return on Investment (ROI) as some of the success factors of agile projects (Figure 1-4). (Ambysoft, December, 2008)

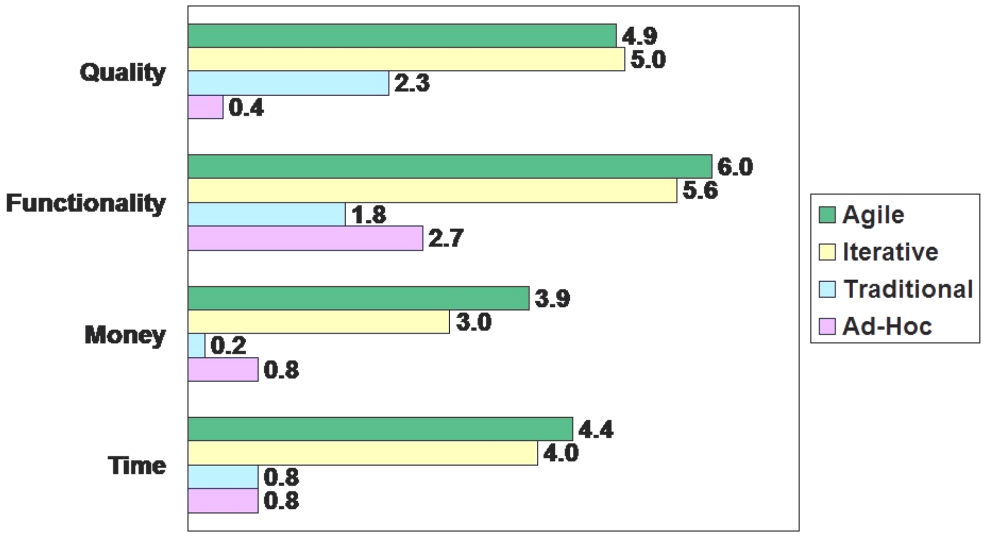


Figure 1-4. Software Development Approaches' Effectiveness
(Source: Ambysoft, December, 2008)

VersionOne Inc. also conducts annual surveys on the usage of agile methods. The results of 2012 survey indicated 90% improvement in ability to respond to changing priorities, 81% improvement in software quality and 85% increase in productivity. The survey included 4048 software professionals coming from companies of size ranging from 100 to

500 employees. The results of the survey are summarized in Figure 1-5. (VersionOne, 2012)

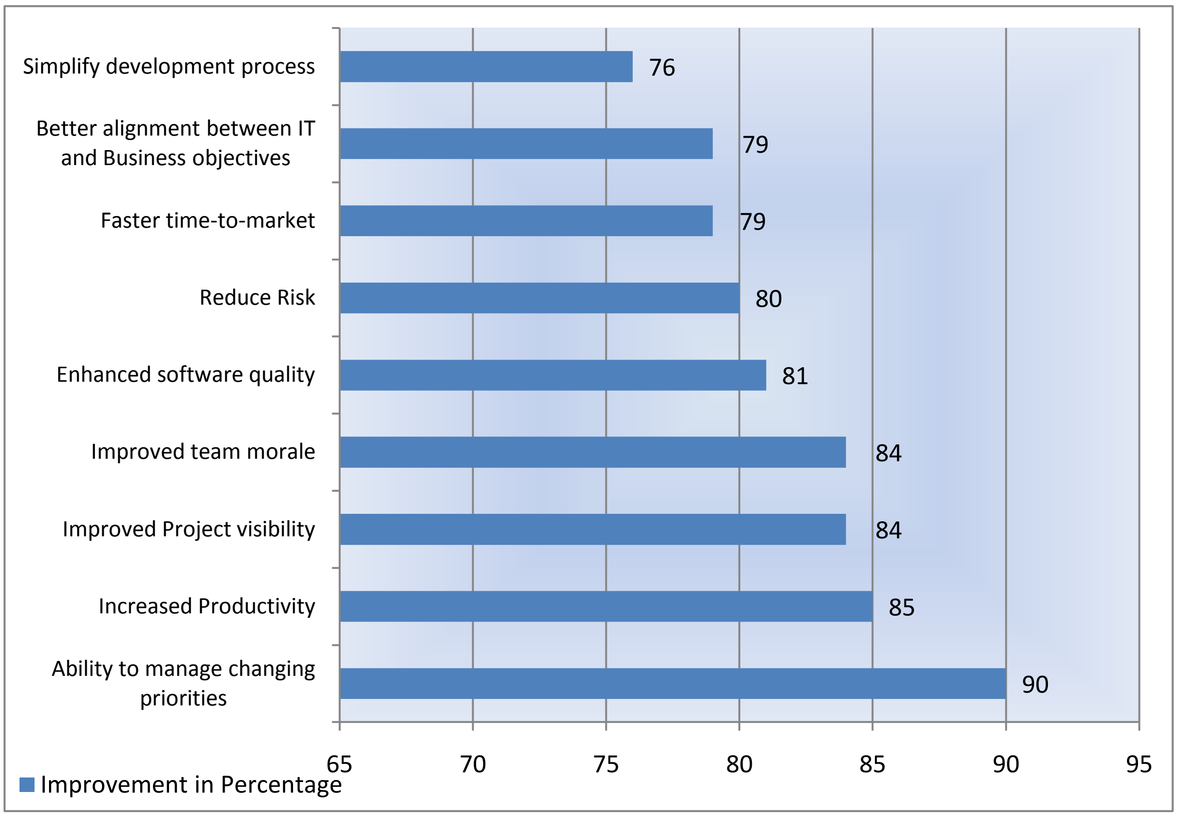


Figure 1-5. Benefits Obtained from Implementing Agile Approaches
(Source: VersionOne, 2012)

The results of all these surveys indicate increased use of agile methods and various benefits offered by them as compared to traditional plan-driven approaches.

However, companies operating in the below mentioned situations face huge challenges while attempting to use agile methods:

- **Public Sector**: German companies in public sector are restricted to use agile methodologies by the government. Companies operating in public sector are accustomed to traditional methods for 20 years or so, hence it is very difficult to bring any change. Mindset of people and organization culture are other opposing factors.
- **Hierarchical Organization**: An organization which is strictly hierarchical with pre-defined roles and responsibilities faces lots of challenges while implementing flexible, non-hierarchical agile methods. People in such organizations are accus-

tomed to defined processes and formal methods; hence they find it hard to accept informal techniques of agile methods.

- **Complex and larger projects**: Agile methods are known to work well within smaller teams and projects. Complex and larger projects need comprehensive and formal communication methods, hence agile methods are hard to implement in such situations.

This thesis attempts to provide solutions to implement agile methods in above situations where it is not possible to immediately shift to pure agile methodologies or organizations want to reap benefits of both agile as well as plan-driven methods. A hybrid approach with an extension of V-model with agile testing methods has been suggested in such situations.

2 Plan Driven Software Development Processes

Failure of many software projects in 1960s and 70s led to evolution of software development processes. Many software projects either ran out of budget or were scraped due to delays or because of final product not meeting user expectations.

The main purpose of software development processes and models is to guide the software development process. They define the order of execution of various phases and transition criteria from one phase to another phase (Boehm, 1988).

Plan driven software development models, also known as classical software development models are described in this chapter.

2.1 Waterfall Model

Waterfall model, developed by W. Royce, is the oldest model for software development. It consists of a set of phases, where one phase must be finished before starting the next phase. The progress of the software is seen as flowing downward like a waterfall and hence the name. Original waterfall model can be seen in Figure 2-1.

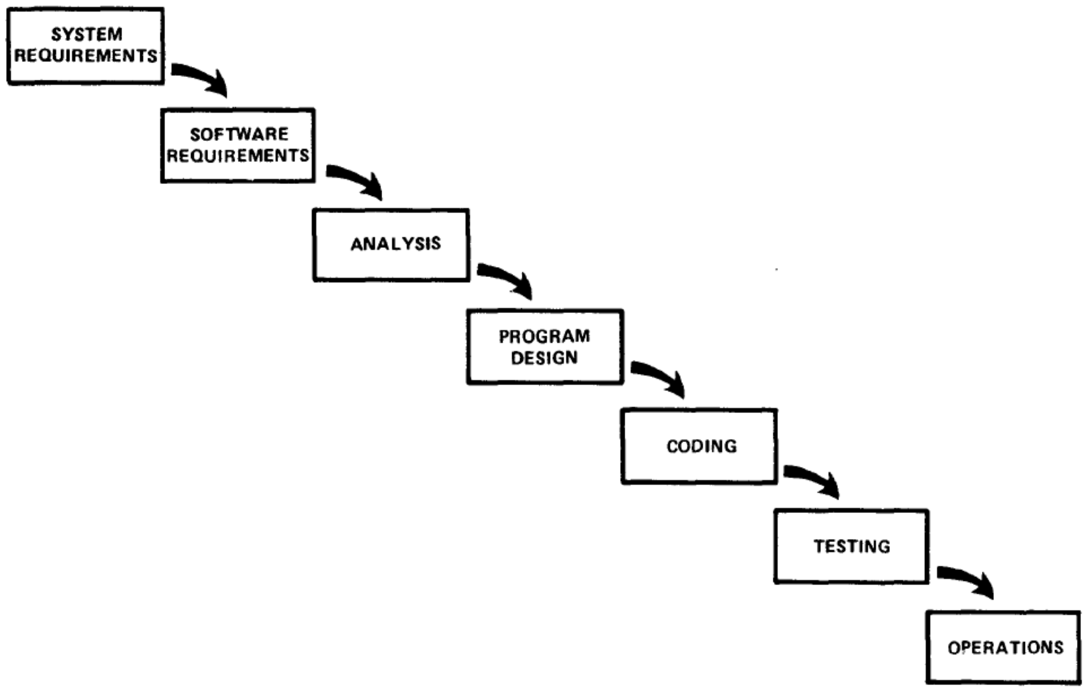


Figure 2-1. Waterfall Model for Software Development
(Source: Royce, 1970)

The various phases of waterfall model are: Requirements specification, Analysis, Design, Coding, Testing, Operation and Maintenance.

Features of waterfall model:

- Software is developed sequentially
- Each phase must be finished before starting next phase
- Requirements are frozen at the beginning of the project
- Huge emphasis on documentation during the testing as well as operational phase. Verbal record is considered intangible and inadequate (Royce, 1970)
- Testing is considered an important phase
- Formal customer involvement is considered important
- Simple and easier to understand

One of the assumptions in waterfall model is requirements don’t change. However, this rarely happens. User requirements evolve during product development and hence they keep on changing.

Waterfall Model also suffers from ‘Late Design Breakage’. In waterfall model, software development progresses fine until the integration phase when design issues are uncovered and flaws are discovered (Royce, 1970). This results in unplanned rework and stress on budget and resources. Figure 2-2 shows comparison between waterfall model and iterative software development with respect to late design breakage phenomena.

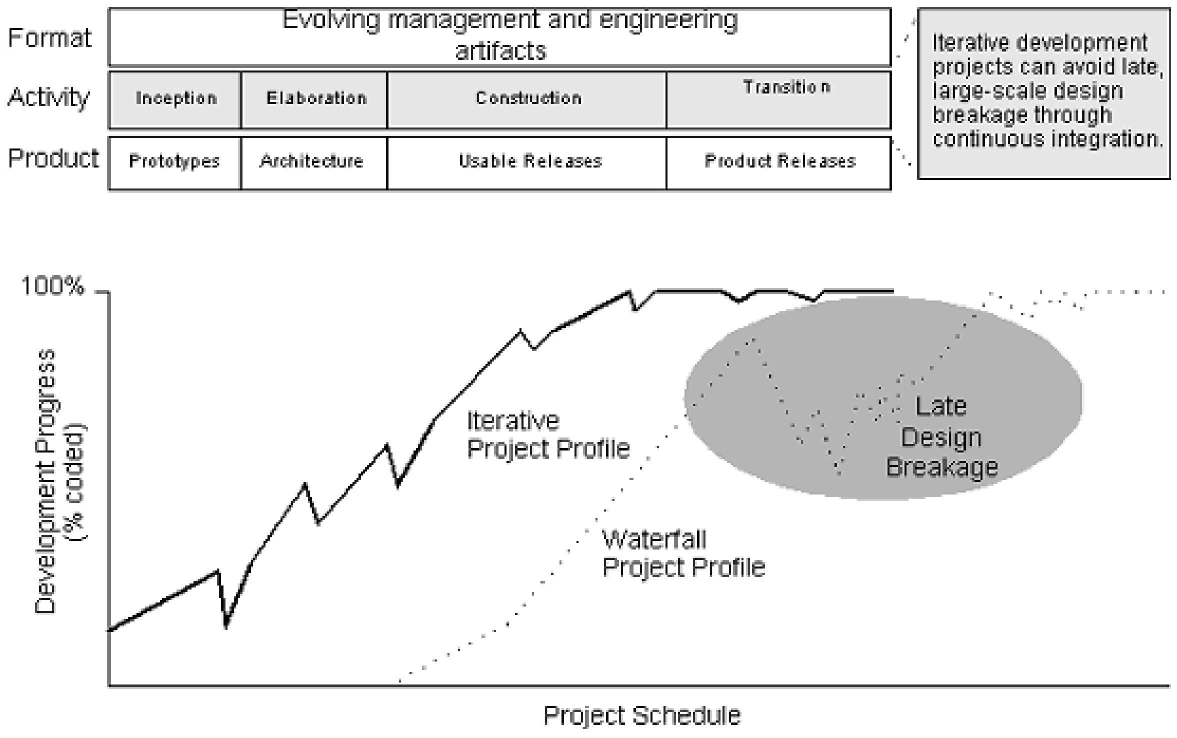


Figure 2-2. Comparison of Waterfall and Iterative Development Processes (Royce, August, 1970)

Pitfalls of waterfall model:

- Unsuitable for changing customer requirements
- Errors are detected quite late in the development cycle (Late Design Breakage)
- Huge effort spent on documentation
- Not suitable for long running projects because of possibility of change in requirements and technology

Waterfall model is most suitable for stable projects where requirements are clearly defined and change is not very likely.

Various modified versions of waterfall model have evolved over time but they all suffer from above mentioned drawbacks. This led to the development of V-model for software development.

2.2 V-Model

V-Model which is an extension of waterfall model was proposed by Paul E. Brook in 1986. It means verification and validation. It is similar to waterfall model but with special focus on testing phases. In V-model, each phase in development has an associated testing phase.

Testing is planned in parallel with development. Test documentation starts as early as requirement analysis phase. V-model can be represented in Figure 4.

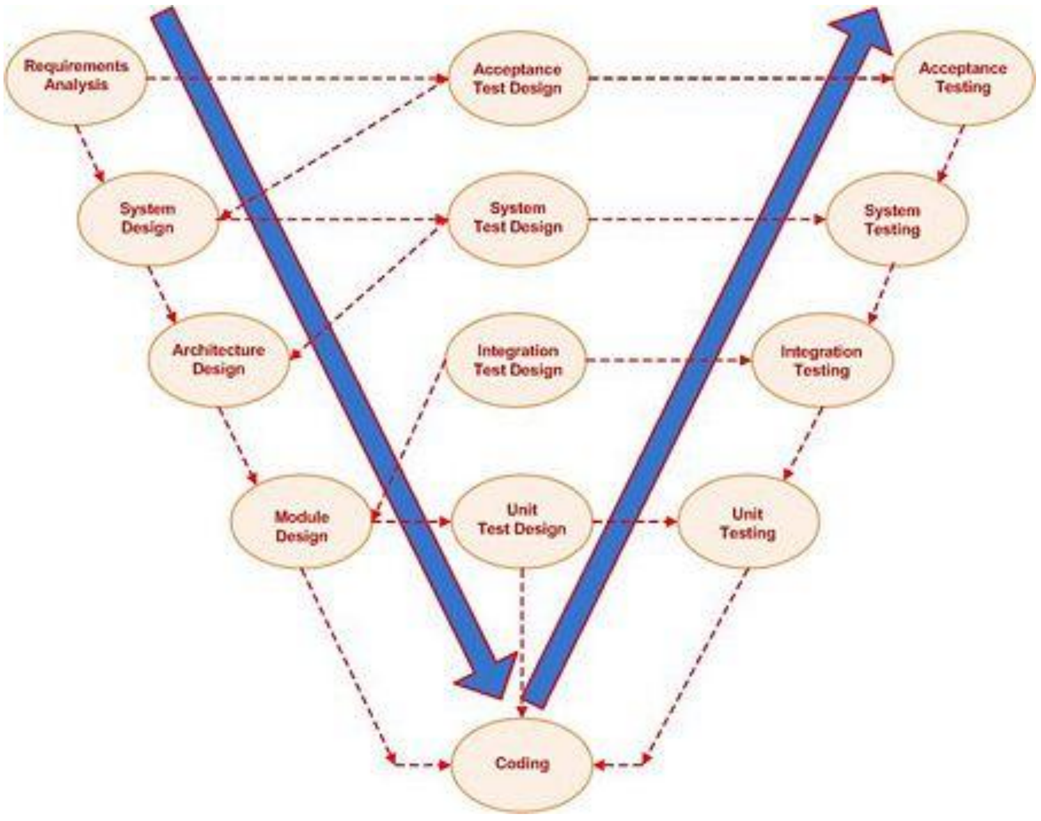


Figure 2-3. V-Model for Software Development
(Source: http://en.wikipedia.org/wiki/V-Model_(software_development))

Right hand side of V-model represents verification phases and left hand side represents corresponding validation phases. International Software Testing Qualifications Board (ISTQB) defines verification and validation as:

Verification: "Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled. [ISO 9000]" (ISTQB, October, 2012).

Validation: "Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled. [ISO 9000]" (ISTQB, October, 2012).

In short, verification means ‘Are we building the product right?’ It involves reviews of documents and plans. On the other hand, validation means ‘Are we building the right product?’ It involves real testing of software to check if it meets the requirements.

V-Model encompasses both verification and validation activities and hence provides a useful framework for software development.

<u>Verification Phases of V-Model</u>:

Various verification phases of V Model are:

<u>Requirement Analysis</u>: In this phase, requirements are gathered from users and this stage specifies what the software is intended to do and not how it is intended to do it. Users are interviewed and Business Requirements Specification (BRS) is prepared.

Roles involved: Business Analysts, Users

Associated Test Planning: User Acceptance Tests

<u>System Design</u>: In this phase, software design is created. System engineers go through BRS and design software according to the requirements. Software Requirements Specification (SRS) and entity diagrams are usually the deliverables of this phase.

Roles involved: System engineers

Associated Test Planning: System Tests

<u>Architecture Design</u>: Architecture Design, also known as High Level Design defines architecture of system. It provides list of modules and their relationships. System hardware, technology, database systems are other considerations of this phase.

Roles involved: Software Architect

Associated Test Planning: Integration Test Designs

<u>Module Design</u>: Module Design, also known as Low Level Design specifies detailed functionality of all sub-modules. Class diagrams, detailed interface diagrams, database tables are the deliverables of this phase. Information from this phase is used by developers while developing software.

Roles Involved: Software Architect, Developers

Associated Test Planning: Unit Test Design

Coding: In this phase, actual development of software takes place. Developers take their respective module's design and convert it into code.

After coding phase, validation phases of V-model start where the test documents created during verification phase are used by testing team to test the software.

Validation Phases of V-Model:

In validation phases, the real testing of software takes place. The various validation phases of V-Model are:

Unit Testing: In Unit testing, each smallest unit or method or function is tested to make sure the source code works fine and there are no errors in the code. Unit test design and plans created earlier are used in this phase. This testing is normally carried out by programmers and white box testers.

Integration Testing: In this testing, interaction between different modules is tested. This is also known as interface testing. The main purpose of this phase is to ensure that different components of system work together and there are no issues in interfaces. The integration test design developed earlier during verification phase is used as a reference for this phase.

System Testing: In system testing, the system or software on the whole is tested to make sure that it meets the system requirements or specifications.

Acceptance Testing: In this phase of testing, software is checked to see if it meets the user requirements specified at the beginning of verification phase. Testing is done from the user perspective and in real situations. It can either be carried out by testers along with the customer or it can also be carried out independently by the customer. There are two types of user acceptance testing.

Alpha Testing: Alpha testing is carried out at the customer site by the testers.

Beta Testing: Beta testing, also known as Field testing is carried out by end users or customers at their own location.

Features of V-Model:

- Sequential and disciplined software development
- Unlike waterfall model, test activities start in parallel with development
- Defines discrete testing phases

- Simpler and easier to understand and provides useful framework for software development
- Clearly defined roles and deliverables
- Documentation driven

Cons of V-Model:

- Inflexible for changing customer requirements
- Like waterfall model, huge emphasis on documentation
- Limited customer involvement
- Testing is again conducted after the whole software has been developed. Hence bugs are detected at a later stage after they have already been implemented
- Huge resources requirement
- Not suitable for smaller projects

V-Model is mostly suitable for large organizations and large projects. Also, the requirements need to be clearly defined, since subsequent change in requirements is a costly affair.

3 Agile Software Development Processes

The term ‘Agile’ means ‘able to move quickly and easily’ (Merriam Webster Online). The history of agile methods dates back to 1930s with formal evolution of methods in 1990s. In 1986, Hirotaka Takeuchi and Ikujiro Nonaka published an article called ‘New New Product Development Game in the Harvard Business Review’, which described flexible strategy for product development (Layton, 2012). In 1995, at the request of Object Management Group (OMG), Ken Schwaber and Jeff Sutherland worked together to summarize their experiences and co-presented Scrum at the Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) conference (Schwaber & Sutherland, 2013).

Agile Manifesto & Agile Principles

In February, 2001, a group of 17 software developers having experience in Extreme Programming (XP), Scrum, Dynamic Systems Development Method (DSDM), Feature Driven Development (FDD), met at a ski resort in Utah to discuss alternatives to heavyweight methodologies of software development. The result of discussion was creation of ‘Manifesto for Agile Software Development’.

Agile Manifesto as described by its authors is shown below: (agilemanifesto.org)

> We are uncovering better ways of developing software by doing it and helping others do it.
>
> Through this work we have come to value:
>
> **Individuals and interactions** over processes and tools
>
> **Working software** over comprehensive documentation
>
> **Customer collaboration** over contract negotiation
>
> **Responding to change** over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

All agile methods follow Agile Manifesto for guidelines. The four values of Agile Manifesto are based upon twelve principles. The four values and the principles behind them can be summarized below:

Value 1 - Individuals and interactions over processes and tools: Agile methods emphasize face to face communication among team members. They consider direct communication between stakeholders and developers more important as compared to following a process such as sending emails. Therefore, they stress upon co-location of all team members and onsite customer presence. Self-organizing teams are considered key to best architectures and designs (agilemanifesto.org). Layton (2012) lists some of the advantages of face to face communication as clear and effective communication, strong teamwork, more chances for innovation and better job satisfaction.

Value 2 - Working software over comprehensive documentation: Agile methods value working software as compared to documentation. Boehm and Turner (2005) describe agile processes motto as YAGNI (You aren't gonna need it) which means limiting design and documentation to just required level. The highest priority of agile processes is to satisfy customer and this is achieved by iterative and incremental development (agilemanifesto.org). Product is developed iteratively and delivered to the customer. Hence after each iteration, customer has the chance to see working software. Feedback and lessons learned from previous iteration are accommodated in next iteration.

Value 3 - Customer collaboration over contract negotiation: Close interaction with customer results in desired product being delivered to the customer. It helps to improve quality of product and reduces defect rate.

Value 4 - Responding to change over following a plan: Agile methods are adaptive rather than predictive in nature. They embrace change during any phase of software development and are capable of responding to changing customer requirements. In today's dynamic world, business requirements change very rapidly, hence it is very important to adapt according to change rather than sticking to a pre-defined plan.

Agile model lifecycle based upon four values of Agile Manifesto can be depicted in Figure 3-1.

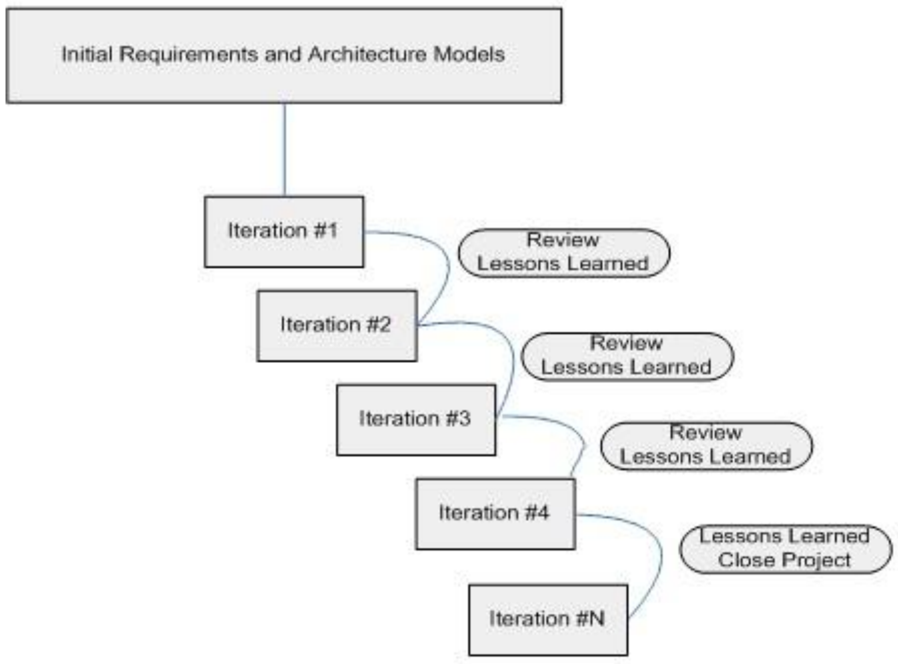


Figure 3-1. Agile Model Lifecycle
(Balaji & Murugaiyan, 2012)

3.1 Inspection of Agile Methods for V-Model Suitability

There are numerous agile methods and agile testing approaches available. The focus of this thesis is the selection of agile testing methods which can incorporate well into V-model. Agile method chosen should not disrupt the existing model completely and should impose minimum challenges for its implementation. Incorporation of a completely different Agile testing approach can lead to failure of a project. For this purpose, agile methods satisfying below criteria have been considered.

1) **Quality Assurance activities**: The foremost condition for selection of agile method is that it should have sufficient quality assurance activities associated with it. There are some agile methods which clearly provide some guidelines on testing (like XP) and there are some which do not provide specific guidelines on testing practices (like Scrum). This doesn’t mean that such agile methods are excluded from the list. Agile methods which do not talk anything about testing practices but having some

associated activities which impact quality of a product directly or indirectly are also considered.

2) **Roles**: Agile methods which have some predefined roles and responsibilities are preferred over the ones which have none. This criterion allows easy mapping of existing roles when adopting an agile testing approach inside V-model. It will also need minimum organizational change during implementation.
3) **Degree of Scalability**: Agile methods are known to work well in smaller projects as compared to traditional methods which can be applied in larger projects. Agile method chosen should be scalable to larger projects and team size. The implementation of an agile method should start from a smaller level but later on it should be possible to scale it to larger level.
4) **Degree of Agility**: Another criterion for selection of agile methods is degree of agility of an agile method. Degree of agility can be measured on the basis of level of processes, planning, documentation, iteration length etc. Qumer and Henderson (2008) have compared agility of six agile methods using a four dimensional framework (4-DAT). Results from their study and other factors have been used to compare agile methods on the scale of light weight to heavy weight methods in *chapter 8.1-Comparison of Agile Testing Methods*. Methods having less degree of agility should be easier to incorporate inside V-model which is a heavy weight process.
5) **Smallest unit of work**: Agile methods which are based on some sort of smallest unit of work are considered e.g. Feature Driven Development (FDD) based on features, Acceptance Test Driven Development (ATDD) based on acceptance tests are chosen. While adopting such a method inside V-model, it will be easier to map requirements to this smallest unit of work.
6) **Communication mechanism**: Agile methods having some sort of specified communication mechanism and techniques within the team and with the customer have been selected. These techniques could be formal in the form of documentation or could be informal like daily stand-ups.
7) **Criticality**: Boehm & Turner (2003) have presented a Home Ground Polar Chart in their book entitled ‘Balancing Agility and Discipline: A Guide for the Perplexed’ for the selection of appropriate software development approach (See Figure 3-2). Criticality is one of the factors listed for determination of relative suitability of

agile or plan-driven methods in a project (Boehm & Turner, 2003). Agile method chosen for adoption inside V-model should be tailorable to critical projects.

8) **Personnel**: This is another factor described by Boehm & Turner (2003) for selection between agile vs. plan driven approach. Agile approaches need more number of Level 2 and Level 3 people (highly experienced and skilled) as compared to plan-driven approaches which can work with mixture of Level 2 and 3 and Level 1B (trainable) resources (Boehm & Turner, 2003) (See Figure 3-2). Agile method chosen for inclusion inside V-model should not be too much dependent upon highly skilled and experienced people. It should be possible to work with less experienced people which could be trained later on.

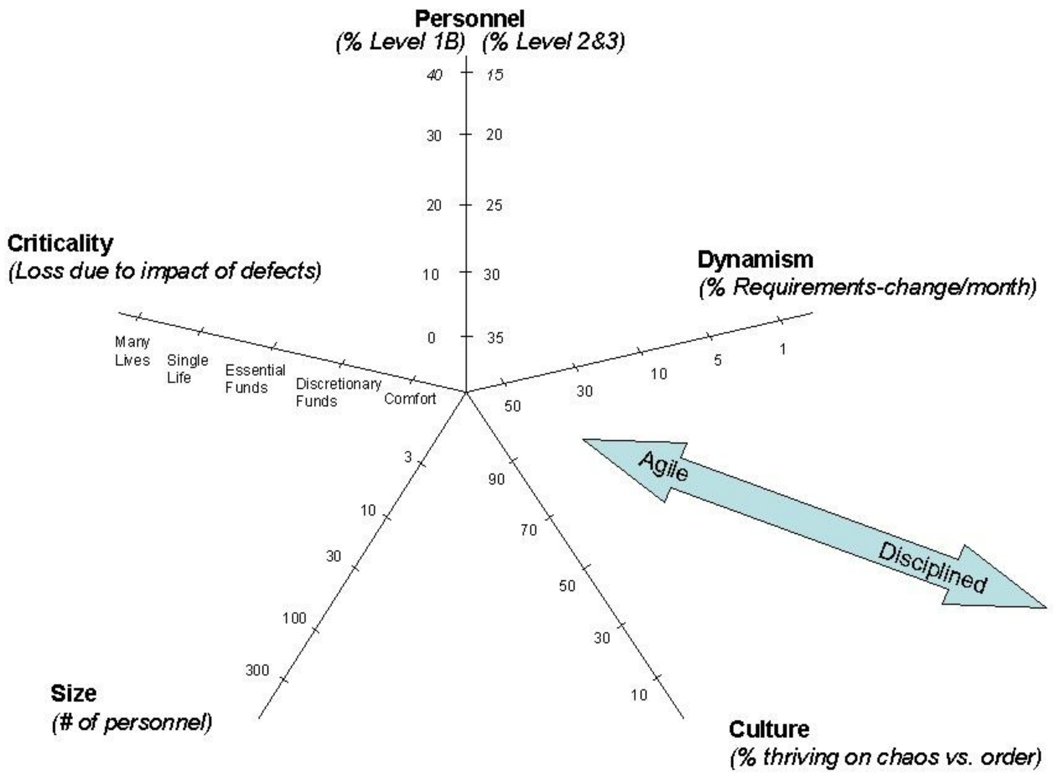


Figure 3-2. Home Ground Polar Chart
(Source: Boehm & Turner, 2003)

Agile software development processes and agile testing methods satisfying above criteria are explained in detail in following sub chapters and in *chapter 4-Agile Testing*.

3.2 Scrum

Scrum is the most popular agile method. A survey conducted by VersionOne in 2012 indicates increased popularity of Scrum among organizations. 54% of respondents said that they use Scrum and 18% said that they use Scrum variants (Scrumban, Scrum/XP hybrid etc.) (VersionOne, 2012).

Schwaber and Sutherland (2013), who are the creators of Scrum, describe it as

> A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value
>
> Scrum is:
>
> - Lightweight
> - Simple to understand
> - Difficult to master. (p.3)

Figure 3-3 (Schwaber & Sutherland, 2012) shows complexity of software projects as dependent upon three dimensions – requirements, technology and people. Software projects are complex since requirements are known with less certainty and technology employed is complex in nature.

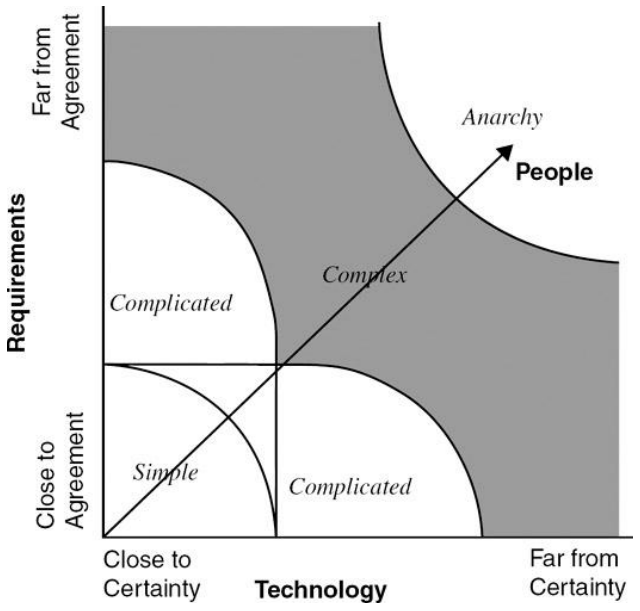


Figure 3-3. The Stacey Graph.
(Source: Schwaber & Sutherland, 2012)

Scrum tackles complexity of software projects by using empirical process control. Empirical process is based upon experience and three pillars of transparency, inspection and adaptation. Transparency means common understanding of process and definition of 'Done' is shared by all the participants. Scrum artifacts are periodically inspected to detect any undesirable variations. Adjustments are made to the process as soon as deviations are detected. (Schwaber & Sutherland, 2013)

Scrum is based upon iterative incremental approach to deliver potentially releasable product increment. Scrum process can be depicted in Figure 3-4. Product is developed in short iterations called sprints where each sprint usually lasts for 30 days.

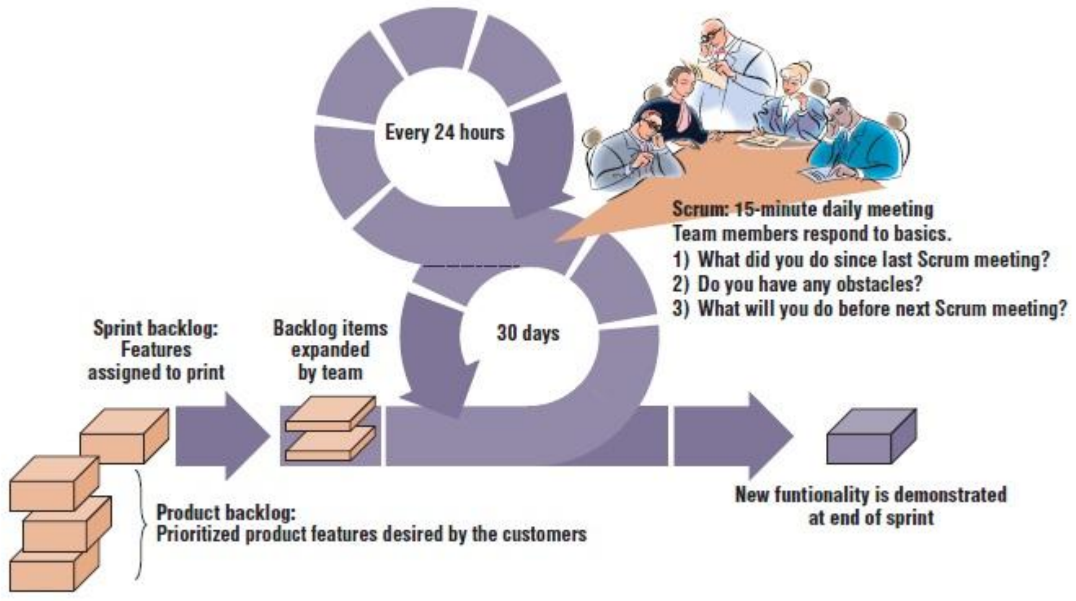

Every 24 hours
Scrum: 15-minute daily meeting
Team members respond to basics.
1) What did you do since last Scrum meeting?
2) Do you have any obstacles?
3) What will you do before next Scrum meeting?
Sprint backlog: Features assigned to print
Backlog items expanded by team
30 days
Product backlog: Prioritized product features desired by the customers
New funtionality is demonstrated at end of sprint

Figure 3-4. Scrum Process

(Source: Boehm & Turner, 2005)

Scrum framework is composed of Scrum team, events, deliverables and artifacts.

3.2.1 Scrum Roles

Scrum team consists of three roles – the Product Owner, Development Team and the Scrum Master. Schwaber (2004) has compared these roles to chickens and pigs, where people who are committed to the project are referred to as pigs and those who are just involved are referred to as chickens (Please see Chapter 1Appendix B for the related joke on chickens and pigs). In Scrum - product owner, development team and Scrum master are pig roles (Schwaber, 2004). They are the ones who are directly responsible for the product whereas customer and management are chicken roles (Schwaber, 2004). They are not committed to the project at all the times and are just interested in project progress. Scrum team is self organizing and cross functional team. Team plans and works on its own (self-organizing team) without much interference of management.

Product Owner is responsible for maintaining the list of requirements in a product back-log. He describes requirements in terms of user stories which explain the intended functio-nality. A simple user story consists of the pattern Title….As a…I want to…so that…It also contains acceptance criteria in the form When I do this, this happens. (Layton, 2012)

Additional responsibilities of product owner include:

- Listing all requirements in product backlog
- Prioritizing items in product backlog on regular basis
- Ensuring that development team builds the required functionality and meets the acceptance criteria
- Explaining all the requirements by means of user stories which are clear and un-derstandable by development team
- Maintaining project’s Return on Investment (ROI)
- Tracking project progress and providing project status to stakeholders

Product owner has the final say on requirements implementation and prioritization. If cus-tomer wants some requirement to be developed on priority, he needs to contact and coordi-nate with product owner.

Development Team is responsible for development of product. It is a cross functional team with people having required skills, eliminating team’s dependency on external re-sources. Development team is responsible for design, development and testing of product, delivering potentially releasable product at the end of each iteration. It is also responsible for estimation and implementation of the user stories it has committed for the current sprint.

Scrum Master ensures that proper Scrum process is followed by the team. His responsibil-ities include:

- Coaching team for Scrum process
- Removing any impediments to the team's progress (Schwaber & Sutherland, 2013)
- Maximizing business value the team delivers within a sprint
- Facilitating Scrum meetings

A.Pham and P.V. Pham (2012) have summarized responsibilities and collaboration between Scrum Team members as shown in Figure 3-5.

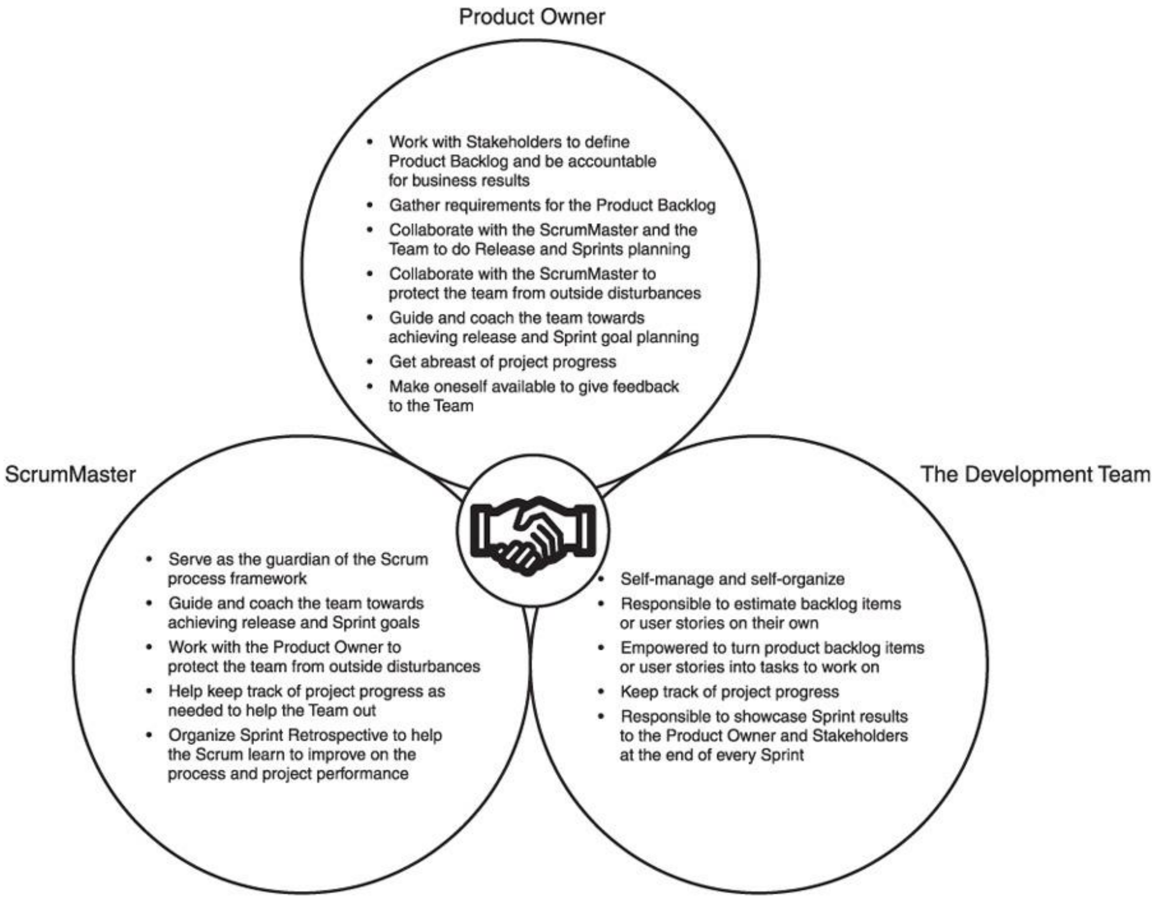

Product Owner
Work with Stakeholders to define Product Backlog and be accountable for business results
Gather requirements for the Product Backlog
Collaborate with the ScrumMaster and the Team to do Release and Sprints planning
Collaborate with the ScrumMaster to protect the team from outside disturbances
Guide and coach the team towards achieving release and Sprint goal planning
Get abreast of project progress
Make oneself available to give feedback to the Team
ScrumMaster
Serve as the guardian of the Scrum process framework
Guide and coach the team towards achieving release and Sprint goals
Work with the Product Owner to protect the team from outside disturbances
Help keep track of project progress as needed to help the Team out
Organize Sprint Retrospective to help the Scrum learn to improve on the process and project performance
The Development Team
Self-manage and self-organize
Responsible to estimate backlog items or user stories on their own
Empowered to turn product backlog items or user stories into tasks to work on
Keep track of project progress
Responsible to showcase Sprint results to the Product Owner and Stakeholders at the end of every Sprint

Figure 3-5. Responsibilities of Scrum team members.
(Source: A. Pham & P.V. Pham, 2012)

3.2.2 Scrum Events

Scrum meetings or so called events facilitate transparency, inspection and adaptability which are necessary for empirical process control. All Scrum events are time boxed which means they are held for specified time. Scrum events are organized on four levels – sprint planning meeting, daily scrum meeting, sprint review and sprint retrospective.

In **Sprint Planning Meeting**, planning for the sprint takes place. Sprint planning meeting is usually held for eight hours for a four week sprint. It consists of two parts which answer questions of ‘What’ and ‘How’ (A. Pham & P.V. Pham, 2012). During the first part of planning meeting, the product owner presents product backlog and works with the development team to decide which user stories will be part of next sprint. Development team has the autonomy to decide on its own how many user stories it wants to work upon. It also provides estimation for the user stories. During the second part of the sprint planning meeting, development team works together to analyze user stories selected during the first part. The team divides each user story in its tasks. The tasks are placed on the task board and each team member decides himself which task he wants to handle first.

Daily Scrum Meeting lasts for 15 minutes and is held daily at the same time and place. During the daily Scrum meeting, team members discuss three questions (Schwaber & Sutherland, 2013):

- What was done since the last Scrum meeting?
- What will be done before the next Scrum meeting?
- What are the obstacles being faced by the team?

Daily Scrum meetings help timely achievement of sprint deliverables. Any impediments faced by the team are identified and resolved on time, avoiding any delays. Scrum master ensures that daily Scrum meeting happens but his presence is not mandatory.

Sprint Review Meeting takes place at the end of sprint and is attended by product owner and development team. Scrum master organizes sprint review meeting. Development team demonstrates what was done during the sprint to product owner and product owner can accept or reject the implementation. Based upon what was achieved and what was not achieved, product backlog is adjusted and stories might be identified for the next sprint.

Sprint Retrospective Meeting is held after the sprint review meeting. The main purpose of retrospective meeting is to inspect last sprint and get team’s feedback on what went well and what did not go well. Sprint retrospective meeting provides the opportunity to identify weaknesses of last sprint and to include the improvements in next sprint.

3.2.3 Scrum Artifacts

Scrum artifacts include product backlog, sprint backlog, burn-down charts, burn-up charts and task boards. These artifacts help to track project progress and ensure shared understanding of all stakeholders.

Product Backlog: It lists all the requirements ordered from high priority to low priority. The product owner is responsible for maintaining product backlog. Product backlog is the single source for all the requirements and features for the stakeholders. It keeps on evolving with time. Task estimations, sprint numbers, user stories are some of the items included in product backlog.

Sprint Backlog: The sprint backlog is the subset of product backlog which contains the requirements to be worked on by the development team in a specific sprint. The sprint backlog contains task estimations, responsible person for task and status. Development team updates sprint backlog during a sprint. Any item not covered in a sprint is moved back to the product backlog.

Burn-down charts: In order to track progress of sprint, burn-down charts are used. Sprint burn-down chart indicates how much needs to be done to complete a sprint. Similarly project burn-down chart shows work progress for whole of project (See Figure 3-6)

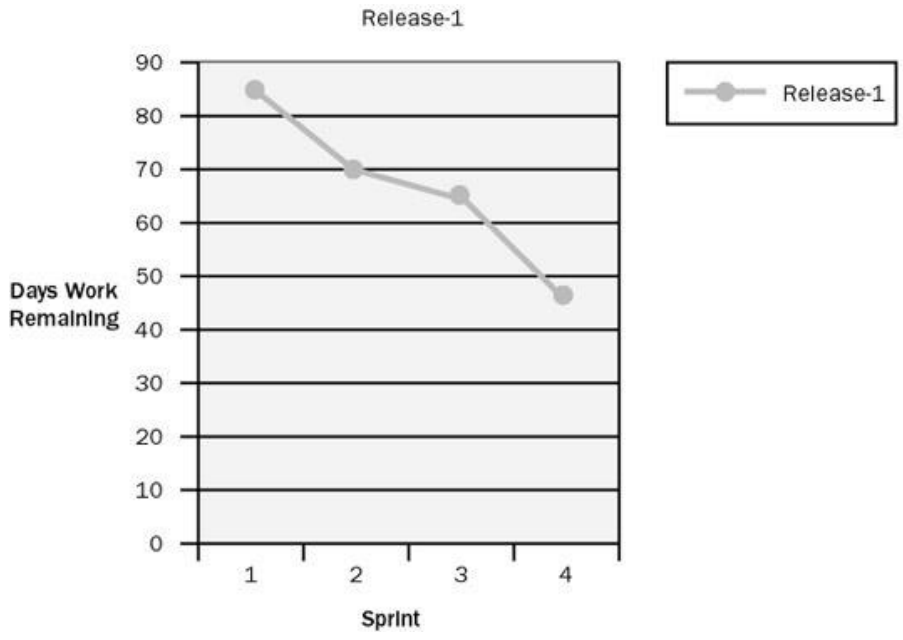


Figure 3-6. Sample Project Burn-down chart
(Source: Schwaber, 2004)

Burn-up charts: Like burn-down charts, burn-up charts can be used in agile projects managed by Scrum methodology. Burn-up charts plot work completed in a sprint against time, hence they slope upward as compared to burn-down charts that slope downward.

Task Boards: A task board is used by development team members to track their work. A task board lists tasks on sticky notes along with their status and names of developers responsible for their completion. A task board is usually divided into columns – To Do, In progress, Accept and Done (Layton, 2012).

3.3 Extreme Programming (XP)

Extreme programming (XP) is another popular agile method. It was created by Kent Beck at around 1996 at Chrysler. In XP, each good practice is done in an extreme way and hence the name e.g. XP suggests that if testing is a good practice then do it at all the times and if pair programming is good, then team should always follow pair programming practice (Slinger & Broderick, 2008). Kent Beck challenged the traditional cost of change curve and argued that cost of changing software can be kept uniform over time by following some practices (See Figure 3-7). Upfront requirements gathering can be avoided and requirements can be defined with the progress of project.

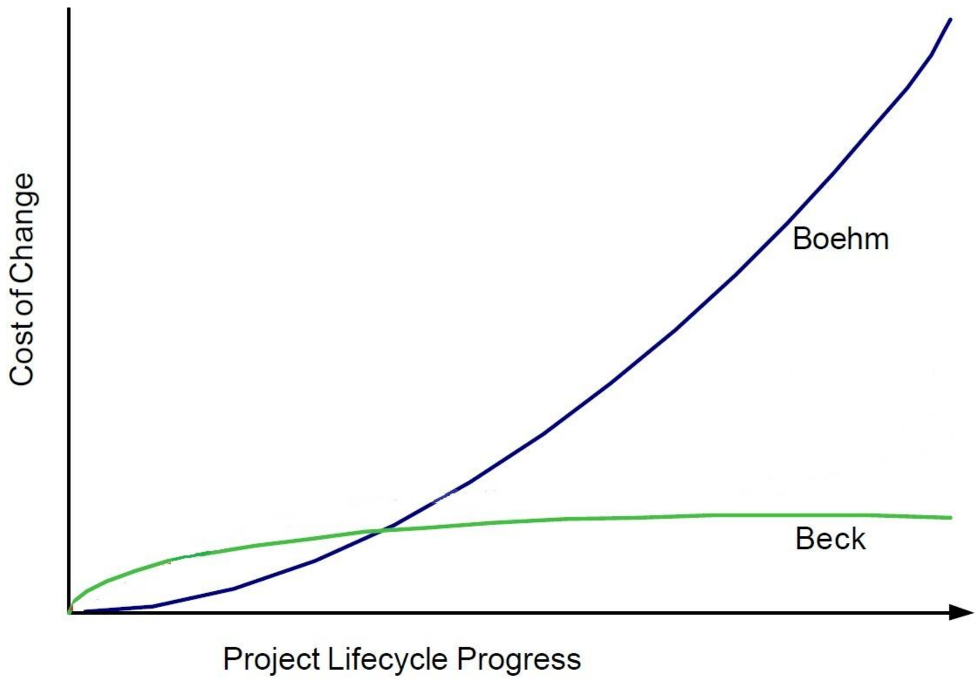


Figure 3-7. Cost of Change Curves
(Source: Adapted from Szalvay, 2004)

Kent defined 4 values and 12 engineering practices in order to achieve flat cost of change curve. 12 practices later on evolved into 28 practices.

3.3.1 XP Values

XP is based upon four values:

1) Communication
2) Simplicity
3) Feedback
4) Courage

Communication: XP also emphasizes upon first value mentioned in agile manifesto which considers face to face communication important over written one. Developers and customers are encouraged to be co-located to avoid any misunderstandings and to resolve issues faster.

Simplicity: XP emphasizes upon simple design and code. Designing is done only for the current iteration and complexity is kept at minimum.

Feedback: XP considers feedback an important part of the project. Customer is involved at regular intervals. He defines requirements for each iteration and steers the project by providing frequent feedback.

Courage: It takes a lot of courage for the developer to throw away and rewrite the code he wrote if it doesn’t meet customer expectations any longer. XP encourages developers to throw away code that is not performing well and is breaking here and there.

3.3.2 XP Practices

XP defines some practices for software development. These practices can be followed in any project independent of software methodology employed. Some of the useful XP practices are defined here.

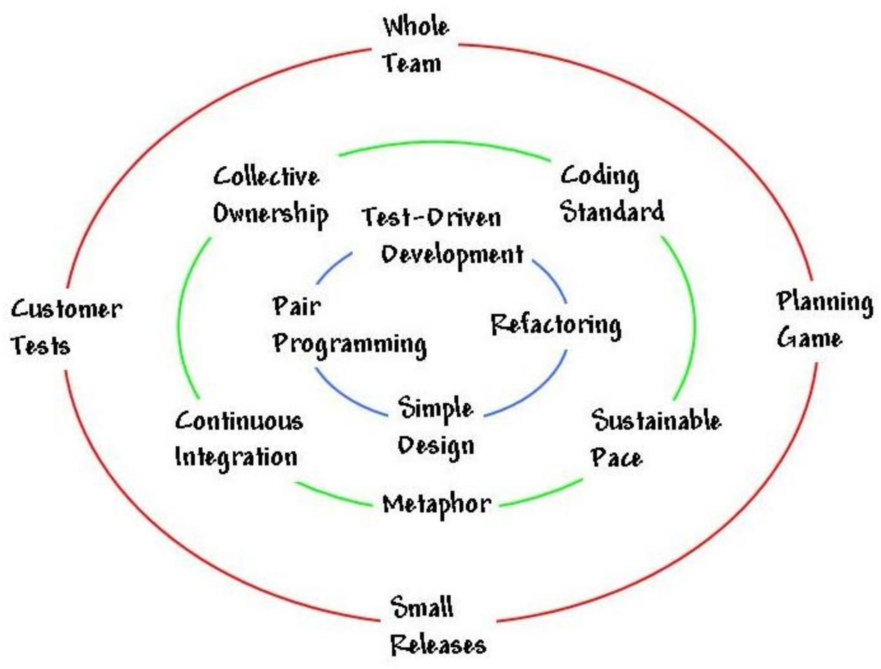


Figure 3-8. XP Practices
(Source: XProgramming.com)

Whole Team: Whole team is responsible for the project. XP recommends on-site customer which means customer or a representative from customer is available at all the times to answer queries related to the product. Customer is suggested to be physically present at the same location as the developers.

Planning Game: XP utilizes planning at two levels – Release planning and Iteration planning. During release planning, planning for the whole project takes place. Customer provides a set of requirements and decision is made which requirement will go into which release. Scope and cost estimations are done on high level. Release planning is usually rough and changes with the progress of project. Release is usually broken down into 1 to 3 weeks long iterations. Iteration planning is usually more concrete as compared to release planning. Customer decides the priority and order of features in the form of user stories. Developers break down stories into tasks and provide their estimates. Planning game is important for steering the project in the right direction.

Small Releases: In XP, product is developed in small iterations and after each iteration, working software having business value can potentially be delivered to the customer. Small releases ensure regular customer feedback and reduce risk.

Customer Tests: Customer writes acceptance tests for each feature in XP. These tests are usually automated and ran every time new feature or change is checked-in or code is refactored to make sure no existing functionality is broken upon addition of new one.

Collective Ownership: Code ownership is considered the responsibility of whole team rather than individual team members. Every team member is allowed to view and edit other team member's code (Ambler & Lines, 2012). Every team member can check-in any change or fix anytime, he doesn't need to take permission or wait for anybody.

Coding Standard guidelines: XP guides developers to use uniform coding standard. This makes the code easier to read and understandable by everyone in the team. This also leads to reduction in defect rate and cleaner code.

Sustainable Pace: XP discourages overtime and suggests working on a sustainable velocity. Standard 40 hour week work is suggested for the team.

Metaphor: A metaphor is the set of terms and common language used to describe functionality of system (Hightower et al., 2004). Use of metaphors avoids any misinterpretations and makes sure everyone uses the same language for the system.

Continuous Integration: Developers should check-in their code as frequently as possible. This avoids 'Late design breakage' and unnecessary delays in project. Developers are encouraged to check-in their code on daily basis. Automated builds are the suggestion to avoid manual work.

Test Driven Development (TDD): In XP, developers practice test first approach which means developers write tests before writing code. Each time a developer checks-in his changes, whole set of tests are run to make sure the system doesn't exhibit any undesired behavior. TDD is explained in more detail in *chapter 4.3.1-Test Driven Development (TDD).*

Refactoring: XP recommends refactoring which means restricting and improving the code design without changing its behavior (Dooley, 2011). Refactoring removes redundant, duplicate, unnecessary, bad smelling code and leads to cleaner, more understandable and better smelling code (Stephens & Rosenberg, 2003).

Simple Design: XP suggests keeping design as simple as possible while fulfilling all the requirements. Refactoring is one technique for keeping design simple.

Pair Programming: Pair programming means two developers work on the same piece of code on one computer. Pair programming helps to reduce defect rate and facilitates knowledge sharing. XP recommends switching pairs on regular basis.

3.4 Lean Software Development (LSD)

Lean thinking or lean manufacturing can be dated back to 1940s when Taiichi Ohno, father of Toyota's production system created a way to deal with Toyota's manufacturing problem. Toyota needed to produce cars at a cheap rate but they couldn't use mass production because of small number of customers. Ohno, then invented lean manufacturing which is based upon principle of waste elimination. (Russo, Scotto, Sillitti & Succi, 2010)

The Toyota Production System was based upon Just-in-Time (JIT) concept which means produce only what is necessary. The upstream process doesn't produce until the downstream process needs it.

3.4.1 LSD Principles

LSD is based upon lean manufacturing principles. M. Poppendieck & T. Poppendieck (2003) applied lean principles to software development and explained them in their book 'Lean Software Development: An Agile Toolkit'. The seven lean principles adapted to software development can be explained here.

1. **Eliminate waste**: Any activity which does not create value to the customer is a waste. Lean principles guide to identify such activities and eliminate them. M. Poppendieck & T. Poppendieck (2003) have explained seven sources of waste in software development which include partially done work, extra processes, extra features, task switching, waiting, motion and defects.
2. **Amplify learning**: Software development is an empirical and learning process. In LSD, learning is achieved by iterative development. After each iteration, prototype is shown to customer and his opinion is asked. Feedback from customer provides a way of improving product and minimizing defects.

3. **Decide as late as possible**: It is not important to start with full set of requirements upfront. M. Poppendieck & T. Poppendieck (2003) recommend delaying decisions until the last responsible moment until the feature is clear and complete information is available. This is particularly important in software industry where there is high uncertainty.
4. **Deliver as fast as possible**: Delivering high quality product frequently is the key to customer satisfaction and success. Design, develop and deployment cycle times should be reduced and small iterations should be incorporated.
5. **Empower team**: In lean organization, developers can take decisions on their own without management interference. Team is given more power and responsibilities which boosts their confidence and trust.
6. **Build integrity in**: M. Poppendieck & T. Poppendieck (2003) define two types of integrity – perceived and conceptual. The software needs to have both types of integrities. Perceived integrity delights the customers by delivering them what they want. For perceived integrity, regular flow of information between customers and developers is important. Conceptual integrity means all pieces of software work together properly and for this flow of information between developers is important. (Russo et. al, 2010)
7. **See the whole**: Lean thinking recommends seeing the software as whole and optimizing processes accordingly rather than looking at sub systems and concentrating too much on them. Interaction between sub systems is more important than individual systems.

3.4.2 Kanban

The meaning of Japanese word Kanban is card/ticket. Kanban, which is the basis of lean manufacturing and JIT principles, first originated in Toyota Production system. In the context of manufacturing, it is a physical card used by downstream processes for signaling upstream processes when replenishment is required. It is based upon ‘Pull’ concept wherein downstream processes pull work from upstream processes as and when required. This leads to reduction in waste and limits the work in progress and inventory. Kanban concept as applied to production processes can be seen in Figure 3-9.

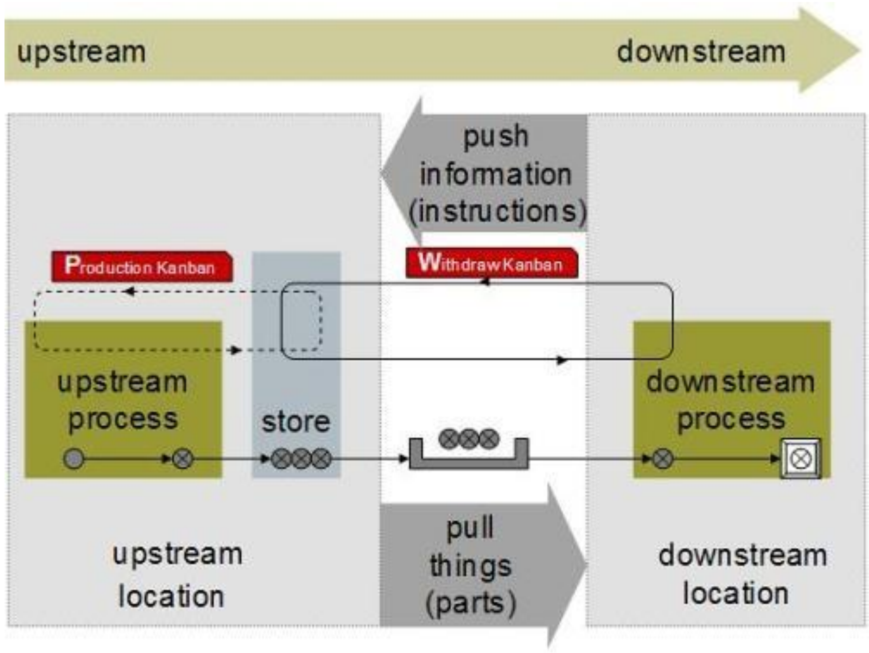


Figure 3-9. Kanban in Production

(Source: http://www.infoq.com/articles/hiranabe-lean-agile-kanban)

3.4.2.1 Kanban Software Development

David J. Anderson applied principles of Kanban to software development and identified five principles.

Kanban Core Principles:

1. **Visualize workflow**: In order to get clear understanding of a process, it is important to visualize it first. In Kanban software development, teams use Kanban board which displays all the work items in columns representing their status and phase of software development. Status of work item is represented in three columns – To Do, Doing and Done (names may vary with organizations). Team members have responsibility of updating Kanban board thus reducing management overhead.
2. **Limit work in progress**: In Kanban method, work in progress is limited at each phase. This avoids work overload for team members. Pull concept is applied to move work from one phase to another. When 'To Do' queue in downstream phase becomes empty, work is pulled from 'Done' queue of upstream phase.
3. **Manage flow**: Flow of work through each phase of software development should be measured and managed. Any bottlenecks should be identified and resolved.

4. **Make policies explicit**: Without making process policies explicit across organization and team, it is very hard to hold discussions about improvement. Hence process policies should be made explicit and understandable by everyone in the team. (Anderson, 2010)
5. **Improve collaboratively (using models and the scientific method)**: Kanban method suggests use of models and theories and scientific approach about work and workflow. Team should have a shared understanding of these theories and models to bring about improvement. David J. Anderson suggests three models in his book titled 'Kanban - Successful Evolutionary Change for your Technology Business' – The Theory of Constraints, The Theory of Profound Knowledge and the Lean Economic Model. (Anderson, 2010)

Sample Kanban board can be seen in Figure 3-10.

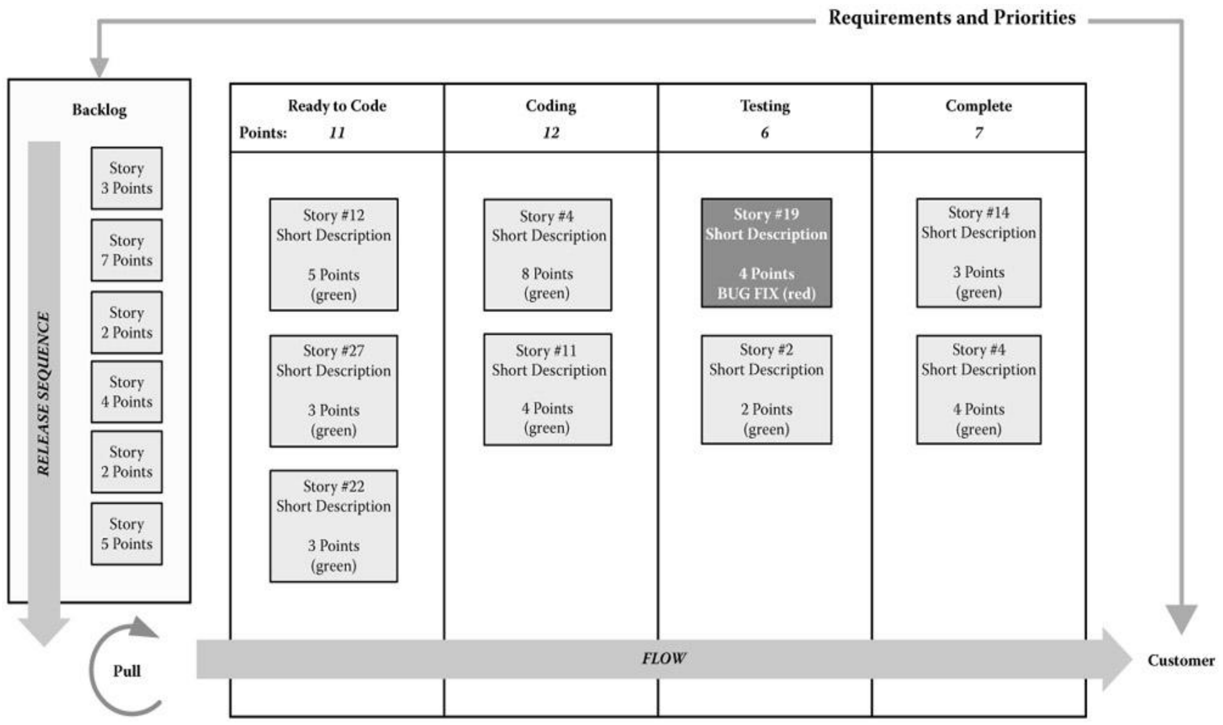

Requirements and Priorities
Backlog
Story 3 Points
Story 7 Points
Story 2 Points
Story 4 Points
Story 2 Points
Story 5 Points
RELEASE SEQUENCE
Pull
Ready to Code
Points: 11
Coding 12
Testing 6
Complete 7
Story #12 Short Description 5 Points (green)
Story #27 Short Description 3 Points (green)
Story #22 Short Description 3 Points (green)
Story #4 Short Description 8 Points (green)
Story #11 Short Description 4 Points (green)
Story #19 Short Description 4 Points BUG FIX (red)
Story #2 Short Description 2 Points (green)
Story #14 Short Description 3 Points (green)
Story #4 Short Description 4 Points (green)
FLOW
Customer

Figure 3-10. Kanban Board
(Source: Bell & Orzen, 2011)

One of the distinct features of Kanban method is that it respects existing roles and responsibilities and applies five core principles stated above to the existing process. Hence Kanban method can be applied irrespective of the software development method in use. The fundamental principles as stated by the originator of Kanban, David J. Anderson are:

Start with what you do now

Agree to pursue incremental, evolutionary change

Respect the current process, roles, responsibilities & titles   (Anderson, 2010).

4 Agile Testing

“Agile testing is a software testing practice that follows the rules of the agile manifesto, treating software development as the customer of testing” (agiletesting.com.au, 2009, September). It is normally followed in projects practicing agile methods but can be implemented in traditional projects too. Agile testing differs from traditional testing in below respects:

1) The time at which testing is carried out in the project
2) The frequency of testing
3) The extent of testing

Agile testing starts from the beginning of project with testers involved in conception or requirement analysis phase whereas traditional testing starts only after the coding phase.

Agile Testing is carried out frequently and in each iteration, whereas traditional testing is only carried once and usually with a specific start and end date. Figure 4-1 shows traditional vs. agile testing.

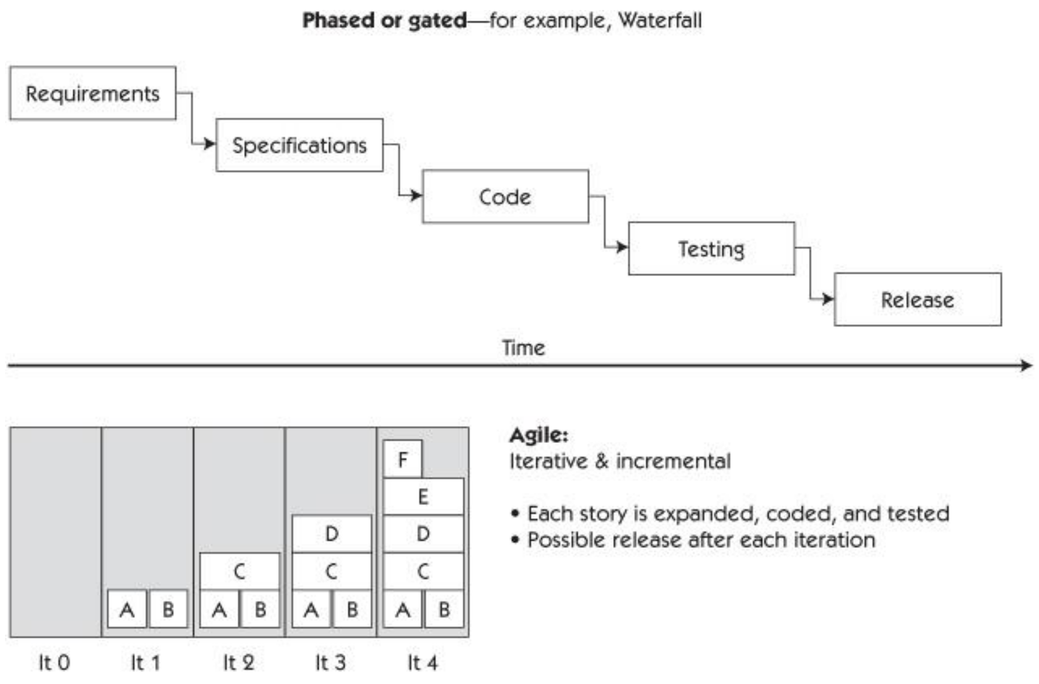

Phased or gated—for example, Waterfall
Requirements
Specifications
Code
Testing
Release
Time
F
E
D
C
A
B
It 0
It 1
It 2
It 3
It 4
Agile:
Iterative & incremental
• Each story is expanded, coded, and tested
• Possible release after each iteration

Figure 4-1. Traditional vs. Agile testing
(Source: Crispin & Gregory, 2008)

4.1 Agile Testing Characteristics

In order to understand testing in agile projects, it is important to understand its various features and how it differs from traditional testing. The following characteristics are guided by Lisa Crispin's thoughts on agile testing (Crispin & Gregory, 2008).

- **Whole team approach**: Testing is considered the responsibility of whole team. It is done collaboratively by testers, developers and users. There is overlapping of roles as whole team works collaboratively (Figure 4-2). Testers help users in writing example tests which are used by developers in coding. Apart from writing code, developers also follow other testing practices like unit testing, continuous integration, TDD etc. Testers also collaborate with developers in writing unit tests. (Crispin & Gregory, 2008)

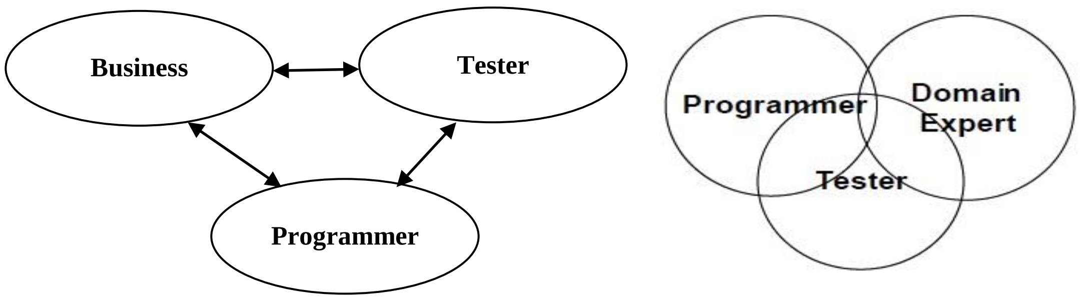


Figure 4-2. Traditional vs. Agile roles
(Source: Crispin & Gregory, 2008)

- **Testing is not a separate phase**: In agile software development unlike traditional development, testing is not considered as a separate phase instead it is merged inside development / construction phase. Tests are written for every user story, followed by its coding and testing. Agile testing is a continuous and fluid process where several stories can be in conception, development or testing phases (Carter, 2010).

  Ambler (2006) has shown agile testing as a continuous process in Software Development Lifecycle (SDLC) which starts with the first phase and goes on till the software is released to the customer (Figure 4-3).

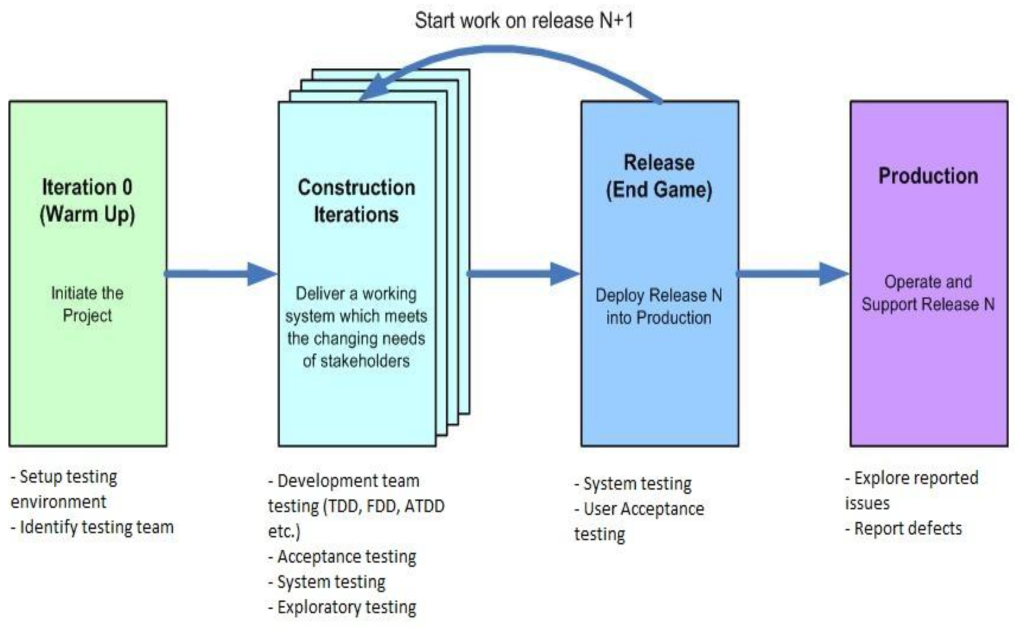

Start work on release N+1
Iteration 0 (Warm Up)
Initiate the Project
Construction Iterations
Deliver a working system which meets the changing needs of stakeholders
Release (End Game)
Deploy Release N into Production
Production
Operate and Support Release N
- Setup testing environment
- Identify testing team
- Development team testing (TDD, FDD, ATDD etc.)
- Acceptance testing
- System testing
- Exploratory testing
- System testing
- User Acceptance testing
- Explore reported issues
- Report defects

Figure 4-3. Agile Testing
(Adapted from Ambler, 2006)

- **Test often and early**: Agile testing methods occur much earlier in the SDLC and are carried out in each iteration. Since in agile methods, product is developed in iterations and after each iteration, potentially shippable increment is delivered to the customer; hence it is important to completely test increment in each iteration.
- **Feedback guides development**: In Agile testing, continuous feedback guides development process. Testers play a key role in providing feedback to developers and users. Other automated methods like continuous integration, build results also provide timely feedback and help in avoiding bottlenecks. (Crispin & Gregory, 2008)
- **Definition of 'Done'**: In agile projects, definition of 'Done' is modified. No user story is considered 'Done' until it is developed and tested too. Hence testing is considered mandatory activity for Doneness of a user story or a feature. (Crispin & Gregory, 2008)
- **Lean testing approach**: Agile testing is based upon lean principles with emphasis on limited documentation. Direct conversations with developers are considered valuable as compared to logging defects in a defect tracker and writing lengthy test reports. (Crispin & Gregory, 2008)

Agile testing is similar to traditional testing in some aspects. A testing process encompassing various testing types needs to be followed in both agile as well as traditional testing. In order to make sure that the final product meets the quality criteria in all the aspects, it needs to be properly tested and this makes the testing strategy an important consideration.

4.2 Agile Testing Quadrants

Crispin & Gregory (2008) have described agile testing Quadrants in their book ‘Agile Testing: A Practical Guide for Testers and Agile Teams’. These Quadrants help in effectively planning testing activities in an agile team. They help in communicating testing goals to the entire team and help to identify which role performs which testing and at what stage.

Figure 4-4 shows Agile Testing Quadrants.

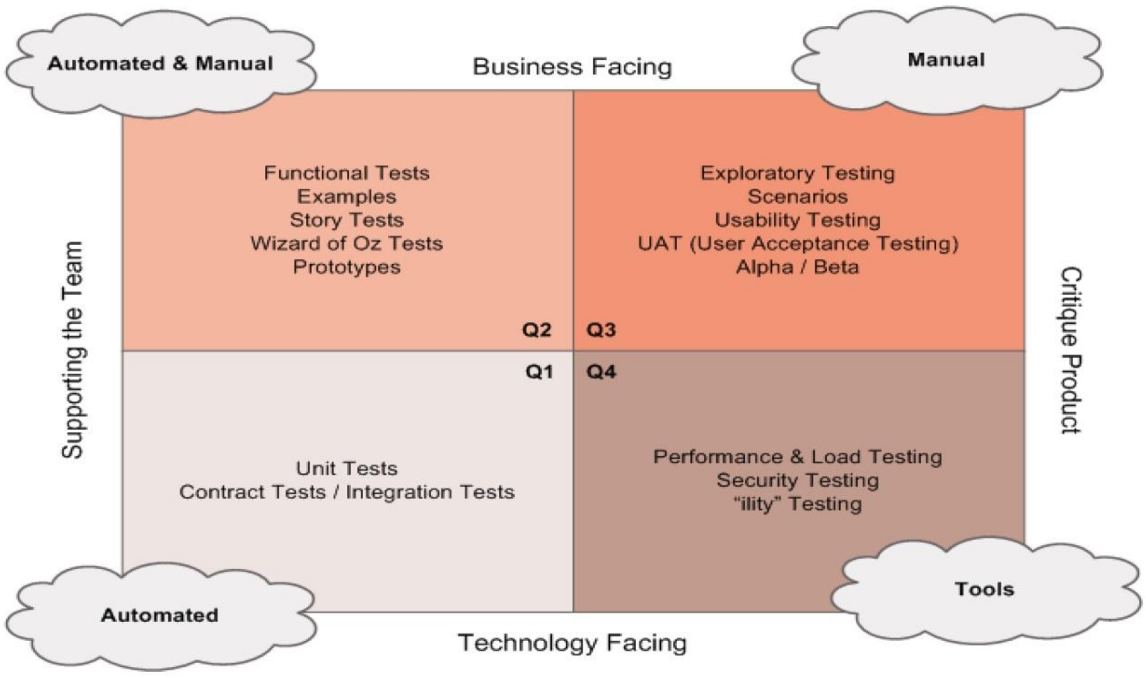


Figure 4-4. Agile Testing Quadrants
(Source: Crispin & Gregory, 2008)

A user story or a feature is considered ‘Done’ when apart from design and development; it has been tested in all four quadrants. The quadrants help ensure that the team is supported and at the same time product is also critiqued. They also help ensure that the product meets business as well as technology requirements.

All four quadrants of agile testing are explained below. Agile testing techniques such as TDD, ATDD, Feature Driven Development (FDD), Specification by Example, Exploratory testing etc. mentioned below are explained in more detail in *chapter 4.3-Agile Testing Methods*.

Q1: Technology facing tests that support the team

The tests which fall in Quadrant1 include unit tests and component tests. These tests help ensure that the quality is built inside product and provide the basis for good design.

Agile techniques such as TDD, FDD, etc. come under Q1 tests. Programmers perform these tests. However, testers can help developers in writing unit tests and in understanding requirements from customer perspective. Q1 tests provide an important base for other quadrant tests. (Crispin & Gregory, 2008)

Q2: Business facing tests that support the team

Functional tests, story tests are some of the tests included in Quadrant2. These tests drive development with customer or business facing tests. Requirements from customers are captured in the form of executable examples which help developers in coding from customer point of view. Direct examples avoid any ambiguity and confusion regarding customer needs. It also helps in clear consensus between developers and users regarding specifications. (Crispin & Gregory, 2008)

ATDD, Specification by Example, Pair Testing are some of the agile testing methods followed in Q2. Programmers in collaboration with testers and customers carry out Q2 tests.

Q3: Business facing tests that critique the product

Exploratory tests, User acceptance tests (Alpha and Beta) which evaluate product from customer needs fall under Quadrant3 tests. These tests are carried out under realistic conditions and provide immediate feedback. Test execution fosters learning and new tests design (exploratory testing). (Crispin & Gregory, 2008)

Q3 tests are carried out normally by end users. Testers and developers provide support to the customers. It is recommended to carry out these tests manually instead of automating them.

Q4: Technology facing tests that critique the product

Quadrant4 includes technical tests that analyze and evaluate the product. ‘ility’ or non-functional tests such as scalability, stability, reliability, maintainability, interoperability, compatibility, installability and so on fall under this category. Other tests include performance tests, load and stress tests, security and recovery tests. The importance of these tests depends upon the customer and software to be developed and may be carried out at any stage. Special tools and skills are required to run these tests. (Crispin & Gregory, 2008)

Mapping of agile testing techniques to agile testing quadrants ensures that product is tested from all the aspects and provides guidelines on whether the tests should be run manually or should be automated.

4.3 Agile Testing Methods

This section explains agile testing methods and practices as being followed by agile teams. These methods look upon agile manifesto for guidelines and follow all four agile values as explained in agile manifesto.

4.3.1 Test Driven Development (TDD)

Test Driven Development (TDD) is one of the core practices of XP. Its origin can be attributed to Kent Beck, the father of XP. Even though TDD was first practiced in XP, it can be implemented in any software development methodology.

TDD is based upon test first programming concepts of XP. In traditional way of software development, developers write tests after writing code. This traditional way of development leads to unnecessary code being written since we do not have direct mapping between requirements and code. It also sometimes leads to few of the requirements not being implemented inside code. Writing test cases is an afterthought and it may lead to skipping of some of the test cases. Defects are already built into the application and detected late. TDD provides a solution to this problem by turning the process upside down and suggests developers to write tests before writing code (Bender & McWherter, 2011). As the name suggests, in TDD, tests drive development. TDD forces developers to think about requirements before writing code. This needs paradigm shift since most of the developers are accustomed to first writing code, followed by writing tests to verify that code. Hence developers at first find it painful to start with tests but once they get used to this, TDD benefits can be clearly realized.

Some people argue TDD to be a design practice and some argue it to be a testing practice, however in agile methods, since design, development and testing are closely integrated, hence it doesn't matter if we consider TDD a design approach or a testing approach (Collino, 2009). TDD is basically a design technique but it creates automated tests as a by-product which can be used for automated unit level regression testing, hence it can be considered a testing technique too (Reid, 2009).

4.3.1.1 TDD Cycle

TDD is based upon three phases of Red, Green and Refactor and can be represented in Figure 4-5.

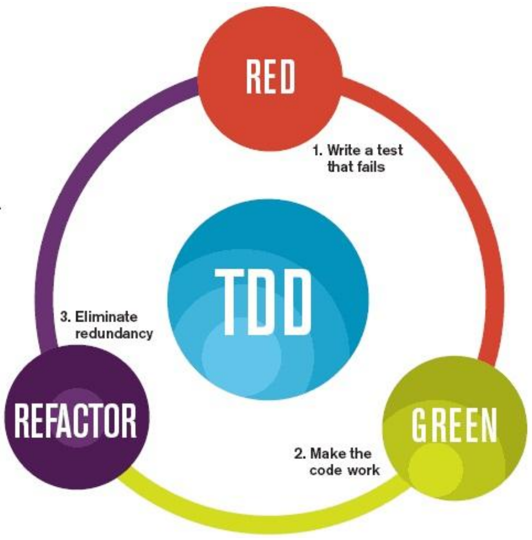


Figure 4-5. TDD Cycle

(Source: http://diogoosorio.com/blog/entry/test-driven-development-tdd-using-phpunit)

TDD starts with developer writing an initial test and then he tries to compile and run that test which obviously fails (*Red Phase*) since there is no code written yet. The developer then writes just minimum amount of code to pass this test (*Green Phase*). During the Green Phase, the developer needs to make sure that none of the existing tests fail. The next step in TDD is *code refactoring* which means developer tries to remove redundant and duplicate code without changing its behavior. Refactoring is a normal practice followed by developers but TDD makes it mandatory after writing each line of code. The whole cycle of *Red -> Green -> Refactor* is repeated again until all the requirements have been covered by tests and the corresponding code.

In TDD, unit tests are derived from specifications and requirements, hence by means of running tests, we have in fact executable requirements. Tests provide a means to ensure that intended business functionality is implemented in code. (Bender & McWherter, 2011)

TDD is based upon the rule that no functionality is added without test. A feature which does not have an associated test is either not added or its test is written first. For a bug fix again a test is added first which makes sure that bug is properly tested and avoids unnecessary rounds between development team and testers.

4.3.1.2 Benefits of TDD

Practicing TDD needs a mindset change and once it is done, it provides various advantages in software development. Some of them are listed below:

Iterative development: TDD leads to development of small increments of code. The initial increment might not be that useful to the user but later on as more and more functionality is added, working software can be delivered to customer at regular intervals.

Simpler high quality code: One of the main benefits of TDD is that it avoids unnecessary complicated code being written. Simpler code is written just to pass the required test. Code which doesn’t correspond to any test and requirement is not written at all.

Direct mapping between code and requirements: TDD provides a means of direct mapping between code and requirements. Specifications are represented in terms of tests and every line of code corresponds to tests. No code is written unless we have a corresponding test for it. Passing of all tests help build confidence in developers since code matches requirements.

Built in regression testing: TDD results in a set of automated unit tests as a by-product. These unit tests provide a means of regression testing and can be run every time a new functionality is added or a bug is fixed. This set of unit tests can also be run after code refactoring and hence ensure that external behavior of code is not changed after refactoring.

Tests not an afterthought: Unlike in traditional software development, tests are not an afterthought in TDD. Testability of code is given primary importance and developers are forced to think in terms of user perspective. TDD ensures maximum test coverage for all the code and builds quality in.

4.3.2 Acceptance Test Driven Development (ATDD)

Like TDD, Acceptance Test Driven Development (ATDD) also known as Executable Acceptance Test Driven Development (EATDD) is based upon test-first programming concepts of XP. In ATDD also, developers write tests before writing code. However there are some differences between ATDD and TDD. The difference lies in the nature of the tests. In TDD, unit tests drive development, whereas in ATDD, acceptance tests or requirements drive development. Another difference lies in the creator of these tests. In TDD, development team writes unit tests and they may take help of customer or domain experts in this process, in ATDD, acceptance tests are written by the customer in collaboration with domain experts or business analysts and testers (Park & Maurer, 2008). TDD mainly works in the design phase and is practiced by the developers; ATDD is mainly a practice in the requirements and specification phase and is practiced by both developers and domain experts in order to have shared understanding of the specifications (Koudelia, 2011).

In ATDD, requirements are written in the form of executable acceptance tests (Park & Maurer, 2008) and these tests are used to drive coding. Fit, FitNesse, Selenium are some of the tools that support ATDD.

4.3.2.1 ATDD Cycle

Hendrickson (2008) in her article on ATDD has described ATDD cycle as composed of Discuss, Distill, Develop and Demo phases with TDD at the heart of it.

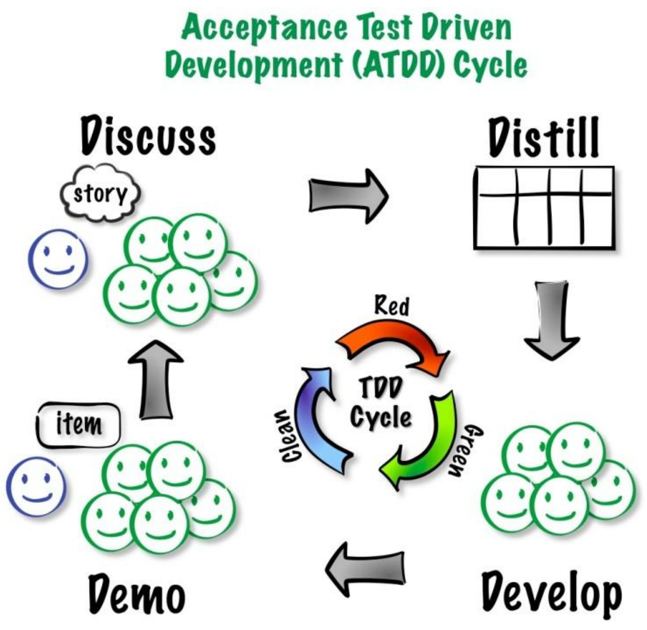


Figure 4-6. ATDD Cycle
(Source: Hendrickson, 2008)

Discuss: ATDD starts with discussion of requirements among customers and business analysts or domain experts. Once the specifications become clear, the user stories are represented in terms of acceptance tests with clear acceptance criteria. Testers are also involved in this stage. This process leads to clear understanding of requirements among various stakeholders.

Distill: In the next stage, acceptance tests are distilled in a framework friendly format (Hendrickson, 2008). There are table-based (FitNesse), text-based and scripting language-based acceptance testing framework tools available (Koudelia, 2011).

Develop: In the next step, TDD cycle of Red -> Green -> Refactor is followed and acceptance tests are used to drive software development. Developers start with executing acceptance test which initially fails, then they go on writing minimum amount of code to pass that test. Once the test passes, code is refactored to remove bad smelling code. Like TDD, this whole process is followed iteratively till all the tests pass.

Demo: Once the tests pass (Green Phase), exploratory testing (explained in Chapter 4.3.4-Exploratory Testing) is conducted and demo of working software is given to the customer.

4.3.2.2 ATDD Benefits

Apart from the benefits offered by TDD, ATDD based upon executable acceptance tests offers some additional benefits.

Shared understanding: Following ATDD practice leads to clear and common understanding between customers, domain experts, testers and developers. During the ‘Discuss’ phase, a clear acceptance criteria for an acceptance test is agreed upon and this eliminates any ambiguity and confusion related to requirements.

Living specifications: In ATDD, business analysts don’t write lengthy requirements documents, instead they write executable requirements in the form of acceptance tests (Park & Maurer, 2008). These tests and hence requirements evolve constantly along with the software development. Developers are forced to update the acceptance tests in case they change anything in the code; otherwise the acceptance tests will fail.

Evaluation of development progress: In ATDD acceptance tests provide a means of tracking development progress. The number of tests passed indicates how much the system has been developed according to user requirements.

4.3.3 Feature Driven Development (FDD)

Feature Driven Development (FDD) is a “client-centric, architecture-centric, and pragmatic software process” (Ambler, n.d.). FDD’s origins can be dated back to 1997 when Jeff De Luca and Peter Codd applied concept of features to a project of large Singapore Bank. The 50 person project was implemented in 15 months (Ambler, n.d.). Later on, in 1999, Jeff De Luca and Peter Codd published description of FDD in their book ‘Java Modeling in Color with UML’. Stefan Palmer and Mac Felsing also published a book ‘A Practical Guide to Feature Driven Development’ in 2001.

FDD is based on object oriented concepts and emphasizes upon creation of object model prior to design and development phases. FDD is considered less agile since it stresses upon modeling and design activities. It also defines prescribed roles and assigns responsibilities.

Features are the core pieces in FDD as user stories are in XP and Scrum. Features can be defined as small functions which have value for the client and are expressed in the form <action><result><object> (Ambler, n.d.). Each feature represents client requirements and is represented in a domain object model using Unified Modeling Language (UML) (Palmer & Felsing, 2001).

4.3.3.1 FDD Cycle

FDD process consists of five activities with last two activities performed iteratively:

1. Develop an Overall Model
2. Build a Features List
3. Plan by Feature
4. Design by Feature
5. Build by Feature

The first three phases involve modeling and planning activities, while the last two phases involve design and development activities.

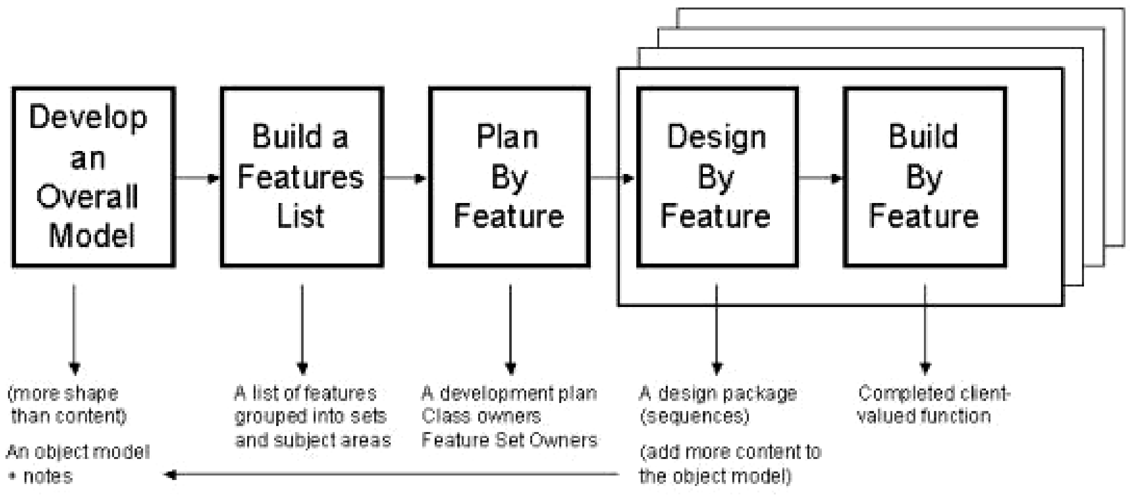

Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
An object model + notes
A list of features grouped into sets and subject areas
A development plan Class owners Feature Set Owners
A design package (sequences)
(add more content to the object model)
Completed client-valued function

Figure 4-7. FDD Cycle
(Source: Palmer & Felsing, 2001)

There are six main roles in FDD - Project Manager, Chief Architect, Development Manager, Chief Programmer, Class Owner, and Domain Expert with additional roles of Domain Manager, Release Manager, Build Engineer, System Administrator, Tester, Deployer etc. (Palmer & Felsing, 2001).

The five activities of FDD can be described below in detail.

Develop an Overall Model: The first step in FDD is creation of an overall model for the project by domain experts and development team members. This is achieved by first defining the scope of the project and by conducting domain walkthroughs by domain experts. Small teams consisting of domain experts and chief programmers for each domain are formed and these teams create models for each domain. Chief Architect may suggest alternatives and guide in refinement of models. The overall object model is then prepared from individual domain models. (Palmer & Felsing, 2001)

Roles involved: Domain experts[1], Chief Programmers[2] and Chief Architect[3].

The results of first activity are creation of overall object model, class diagrams, sequence diagrams and model notes.

Build a Features List: In the second activity, a features list is prepared by Chief Programmers. For this, domains are decomposed into subject areas which are in turn decomposed into business activities and business activities steps. Features are granular functions such that a completion of feature should not take more than two weeks. In case a feature takes more than two weeks, it is broken down into smaller features. (Palmer & Felsing, 2001)

Roles involved: Chief Programmers

Plan by Feature: During the next FDD activity, a development plan is created. The order in which features need to be developed is planned. Chief Programmers are assigned business activities and completion date of business activities is calculated based upon complexity of features, dependencies of features and load of developers. After assignment of business activities, classes are assigned to developers. (Palmer & Felsing, 2001)

Roles involved: Project Manager[4], Development Manager[5] and Chief Programmers.

[1] Domain experts are users, clients, sponsors, business analysts, or any mix of these (Palmer & Felsing, 2001).

[2] Chief Programmers are experienced developers who have been through the entire software development lifecycle a few times (Palmer & Felsing, 2001).

[3] Chief Architect is responsible for the overall design of the system. The Chief Architect resolves disputes over design that the Chief Programmers cannot resolve themselves. (Palmer & Felsing, 2001)

Design by Feature: During the design by feature activity, Chief Programmers select features for development from the features assigned to them. Feature team comprising developers who own the classes belonging to features to be developed is then formed. The developers of feature team create sequence diagrams for the features. Chief Programmers then refine the object models to add additional classes, methods and attributes. Finally a design inspection is held. (Palmer & Felsing, 2001)

Roles involved: Chief Programmers, Developers

Build by Feature: In the last activity of FDD, features are developed by feature team. Class owners implement their respective features followed by code inspection, unit testing and build process. (Palmer & Felsing, 2001)

Roles involved: Chief Programmers, Developers

The 'Design by Feature' and 'Build by Feature' activities are performed iteratively over features with tasks like designing of features, coding, code inspection, unit testing, integration testing, system testing etc.

4.3.3.2 Benefits of FDD

FDD offers some distinct advantages as compared to other agile approaches. Some of them are listed below:

Scalable: FDD is more scalable as compared to other agile approaches like XP and hence it is more suitable for use in large and complex projects.

Tracks project progress: FDD provides ways of tracking project's progress by means of features. It uses percentage of completeness of each feature to predict project completion date (Palmer & Felsing, 2001).

[4] Project Manager is the administrative lead of the project responsible for reporting progress; managing budgets; fighting for headcount: and managing equipment. space, and resources, etc. (Palmer & Felsing, 2001)

[5] Development Manager is responsible for leading the day-to-day development activities. He is responsible for resolving everyday conflicts for resources when the Chief Programmers cannot do it between themselves. (Palmer & Felsing, 2001)

Helps transition towards agility: FDD, being more planned agile approach, is good for organizations transitioning from traditional approaches to agile approaches (McDonald, n.d.).

4.3.4 Exploratory Testing

The term Exploratory Testing was coined by Cem Kaner in 1983 in his book ‘Testing Computer Software’. Before early 1990s, exploratory testing was also known as ad-hoc testing.

Kaner (April, 2008) defines exploratory testing as ‘the style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the value of her work by treating test-related learning, test design, test execution and test result interpretation as mutually supportive activities that run in parallel throughout the project’.

Bach (2003), author of ‘Rapid Software Testing’ gives widely accepted definition of exploratory testing as ‘simultaneous learning, test design and test execution. Exploratory testing is any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests.’

Bach & Bolton (2012) have demonstrated testing on a continuum where on one hand we have pure scripted testing, which is planned well in advance and includes execution of predefined scripts and on the other hand we have freestyle exploratory testing (pure exploratory testing). Testing performed in projects can be anywhere on this continuum with varying degrees of scripted and exploratory testing.

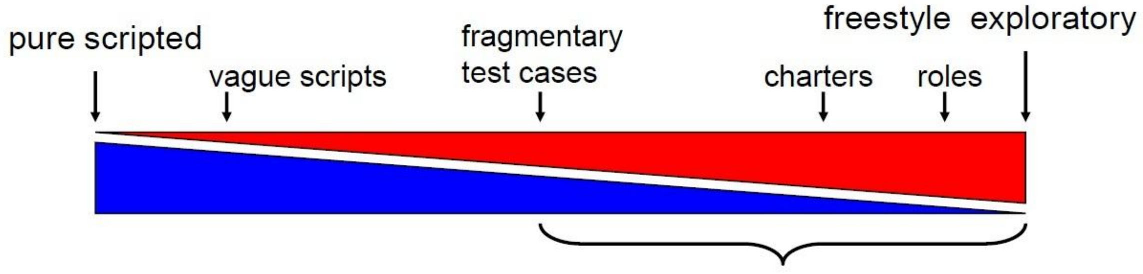


Figure 4-8. Exploratory Testing on a Continuum
(Source: Bach & Bolton, 2012)

Tests are not designed in advance in case of exploratory testing and testers do not follow step by step instructions unlike scripted testing. As the tester executes tests, he uses the feedback gained from last test in order to write and execute new tests. Learning is the key factor in exploratory testing. In scripted testing, scripts hardly change during the execution, on the other hand in exploratory testing; new tests are always optimized based upon the execution of old tests (Bach, 2003).

4.3.4.1 Exploratory Testing Methodology

Bach (2003) has mentioned five basic elements of exploratory testing namely – time, tester, product, mission and reporting. Exploratory tester interacts with a product over a period of time to fulfill a testing mission and reports results. A charter defines the testing mission, it may be detailed or it may just state the overall requirements. A tester then works towards achieving this test mission. (Bach, 2003)

There are two ways to manage exploratory testing – delegation and participation. In management by delegation, a tester is assigned charter by test lead. The tester then executes and writes tests in order to fulfill mission specified in the charter. A tester has the freedom to execute tests on his own. He then reports the results back to the test lead. In management by participation, test lead also participates in testing. He sits along with testers and directs test strategy. (Bach, 2003)

Documentation in exploratory testing varies from documenting bugs to capturing all actions of testers by an automated tool which also serves as a way of regression testing later on. Due to lack of documentation in terms of test cases and scripts, one of the drawbacks of exploratory testing is difficulty in tracking tester's efforts and overall testing progress.

Session-based Test Management: In order to overcome challenge posed by exploratory testing, Bach (2000) has devised tool-supported approach called session-based test management which helps to track testing effort on the basis of 'session'. 'A session is an uninterrupted block of reviewable, chartered test effort' – Bach (2000). A session usually lasts for 90 minutes with three sessions in a day. Time spent in meetings, emails, or conversations is excluded from a session. Only the time spent in testing the mission specified in charter is included in a session. Testers provide session report at the end of a session or at the end of multiple sessions which is reviewed by the test lead or test manager. A session report usually contains information about charter (test mission and areas to be tested), tester name, task breakdown metrics, issues, bugs, notes etc. Task breakdown metrics contain information about time spent on various tasks in a session such as test design and execution, bug investigation and reporting. Testers also have the freedom to test outside the mission specified in charter. Session reports are scanned by a tool which provides metrics in the form of tables. (Bach, 2000)

Session based test management is considerably a new approach and puts a lot of burden on test manager.

4.3.4.2 Exploratory Tester Skills

Skills of tester are very important in exploratory testing. Software Engineering Body of Knowledge (SWEBOK) indicates the same in its definition of exploratory testing '*The effectiveness of exploratory testing relies on the software engineer's knowledge, which can be derived from various sources: observed product behavior during testing, familiarity with the application, the platform….*' (SWEBOK, 2004). A skilled exploratory tester can begin testing with minimal documentation and can find bugs in a small time.

Bach (2003) describes some of the distinguished skills of exploratory tester as:

- Skilled exploratory tester is a test designer who is able to design such tests that are able to explore product from all perspectives such as risk, value, functionality etc.
- Experienced tester is a careful observer. He is able to observe bugs over and above what is mentioned in scripts.
- Exploratory testers are critics of their own. They are able to critique their own results and findings.
- Skilled experienced testers give better ideas about testing a product than a novice tester. They are also able to utilize wide variety of tools and resources as and when required.

4.3.4.3 Benefits of Exploratory Testing

There are various benefits of exploratory testing but experts suggest it using along with scripted testing in order to reap its maximum benefits. The following benefits have been discussed in the existing literature (Bach, 2003; Itkonen & Rautiainen, November 2005; Itkonen, Mantyla & Lassenius, September 2007):

Rapid feedback: Exploratory testing provides rapid feedback between developers and testers; hence any changes can be quickly accommodated in the system and tested.

Diversify testing: Exploratory testing provides a way to diversify testing. It appends scripted testing and confirms the testing performed by test case based testing.

Effective defect detection: Studies indicate that exploratory testing leads to more efficient testing by detecting effective and more number of bugs. A study conducted by Itkonen et al. (2007) indicates that test case based testing produces more false defect reports as compared to exploratory testing.

Minimum preparation: Exploratory testing needs minimum preparation for execution since it doesn't need any scripts to start with. This is particularly helpful in situations where testing needs to be started soon and requirements are unclear or change rapidly.

4.3.5 Specification by Example

Adzic (2011) has defined Specification by example as 'a set of process patterns that helps teams build the right software product by writing just enough documentation to facilitate change effectively in short iterations or in flow-based development'. The components of this definition are:

Set of process patterns: Adzic (2011) has described a set of seven patterns for successful implementation of specification by example. These patterns are described in detail in *sub-chapter 4.3.5.1-Key Process Patterns*.

Right software: There is a difference between building right software and building software right. Building right software (Validation) focuses on if the software meets the user requirements and expectations. Building software right (Verification) focuses on if the software meets the quality criteria. Specification by example helps to build right software and also complements practices that ensure software is built right (Adzic, 2011).

Just enough documentation: Specification by example is based upon just enough documentation for software development. It avoids lengthy and bulky documentation as in traditional software development and utilizes documentation that is maintainable, evolving and testable (Adzic, 2011).

Short iterations: Like other agile methods, specification by example emphasizes product development in short iterations. It can be used in conjunction with Scrum or XP or with flow based processes like Kanban (Adzic, 2011).

The various terms used interchangeably with specification by example are Acceptance Test Driven Development (ATDD), agile acceptance testing, Behavior-Driven Development (BDD) etc. (Adzic, 2011). ATDD has already been described in *chapter 4.3.2-Acceptance Test Driven Development (ATDD)*. The preceding sub-chapters discuss in detail the various concepts of specification by example. While comparing various agile testing methods and extended V-models (see *chapter 8-Comparison of Extended V-Models*), the two terms - specification by example and acceptance test driven development have been used interchangeably.

4.3.5.1 Key Process Patterns

Adzic (2011) has described in his book 'Specification by Example: How successful teams deliver the right software' a set of key process patterns which has been followed by teams implementing specification by example. Figure 4-9 shows the key process patterns of specification by example.

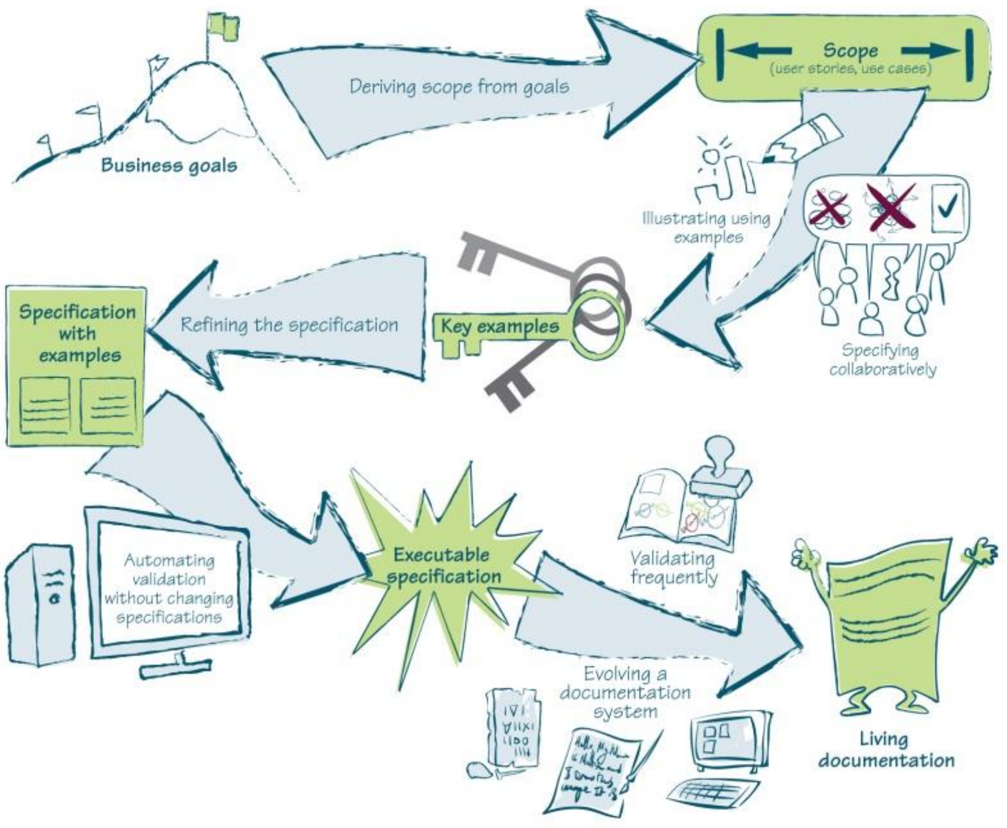

Business goals
Deriving scope from goals
Scope
(user stories, use cases)
Illustrating using examples
Specifying collaboratively
Key examples
Refining the specification
Specification with examples
Automating validation without changing specifications
Executable specification
Validating frequently
Evolving a documentation system
Living documentation

Figure 4-9. Key Process Patterns of Specification by Example
(Source: Adzic, 2011)

1) **Just-in-time approach**: The key process patterns are not followed in one shot or in one phase. They are followed at different stages of software development and in iterations. (Adzic, 2011)
2) **Deriving scope from goals**: In most of the cases, business users, customers, analysts and product owners specify scope of project in the form of user stories, use cases or product backlog items. Development team is not involved in the process and it simply implements those user stories and features. By providing scope, business users indirectly provide a design of solution for which they don't have expertise. In specification by example, development team derives scope from goals. Business users and customers provide business goals. The implementation team works along with users to design a high level solution in the form of user stories and use cases to achieve those business goals. One of the ways to derive scope from goals is to understand why some application is needed and who the users of that application are. Understanding these two parts can lead to better solutions. Another

way to design better solutions is to understand where the value comes from. This helps development team to prioritize business goals and hence scope (in the form of user stories, use cases etc.). (Adzic, 2011)

3) **Specifying collaboratively**: Specification by example suggests collaboration and participation from entire team in requirements specification. Testers and developers are suggested to work along with business analysts in specifying requirements. This practice leads to shared understanding among team members. With developers and testers taking part in specification process, any misinterpretation of requirements at a later stage during development is avoided. Skills and expertise of team members is exploited and it leads to specification of requirements from business as well as technical perspective. (Adzic, 2011)

   Collaboration models: Adzic (2011) has suggested different models for collaboration of entire team. The choice of collaboration model depends upon product's maturity, availability of business users and level of domain knowledge in the team.

   All-team workshops: Teams starting with specification by example are suggested to begin with all team workshops. These are big workshops involving all team members. These type of workshops lead to knowledge sharing among team members. All-team workshops are hard to organize since they involve lot of participants. Once the team has sufficient domain knowledge, other types of collaboration models can be used. (Adzic, 2011)

   Three Amigos: Another type of collaboration model is smaller workshops. These workshops involve one developer, one tester and one business analyst and are called Three Amigos. These type of workshops are easier to organize and are suggested when developers and testers need frequent clarifications regarding requirements and domain. (Adzic, 2011)

   Pair-writing: In case of mature products, where developers have sufficient domain knowledge, it is suggested to write tests in pair. Developers can form pairs either with testers or with business analysts. Developers-analysts pairs make sure that tests are written from business as well as technical perspective. Developers-testers pairs help testers to get an overview of executable specifications and help them to plan automation in advance. (Adzic, 2011)

Frequent review of tests by developers: This form of collaboration is suggested in projects where analysts write tests. Analysts write acceptance tests and get them reviewed from developers. This saves developers' time and allows them to spend more time on implementation. (Adzic, 2011)

Informal conversations: Informal conversations are suggested in situations where business users and development team sit close to each other. These conversations happen on demand and between anyone who is involved in the project. (Adzic, 2011)

Table 4-1 summarizes the various collaboration models discussed.

Collaboration model	Participants	Applicable situation	Product maturity	Domain knowledge of team members	Availability of business users
All-team workshops	Entire team	Starting with specification by example	Immature product	Less	Less
Three Amigos	Developer, tester, business analyst	Frequent clarifications required	Immature product	Less	More
Pair-writing	Developer-analyst or Developer-tester	Mature product	Mature product	More	More
Review of tests by developers	Analysts and developers	Analysts write tests	Mature product	More	More
Informal conversations	Anyone having stake in project	Business users readily available	Mature product	More	More

Table 4-1. Collaboration models in specification by example
(Source: Adapted from Adzic, 2011)

4) **Illustrating using examples**: Requirements written in natural language can lead to misinterpretation by various stakeholders and team members. Specification by example suggests representation of requirements in the form of examples. These examples illustrate the expected functionality of product. Illustration of requirements in the form of examples is useful in case of complex domains where specification of requirements in natural language is difficult. Choice of key examples is an important point to be considered. Developers and testers help business users and analysts in finding corner cases and representation of those cases in the form of examples. Examples should be precise, realistic and easy to understand by all stakeholders. (Adzic, 2011)
5) **Refining specification**: It is important to refine the specification and remove extra details from it. A specification should be precise and it should describe what the application does rather than how it does it. Business users often create examples of user interface of application. These details should be removed from examples. Specifications in the form of examples can serve as acceptance tests and criteria for the entire team. (Adzic, 2011)

   Adzic (2011) has suggested some characteristics of an ideal specification. It should be:

   - precise
   - testable
   - not a script
   - about business functionality and not about software design
   - not tightly coupled with the code
   - self explanatory
   - focused
   - in domain language
6) **Automating validation without changing specifications**: Specifications represented by examples are used for validating system during development to make sure the developed product meets user needs. This process of validation needs to be faster in case of short iterations in order to provide quick feedback. Hence it is suggested to automate the validation process. The automation should not change the specifications. If technical automation like in development and testing is used to

automate specifications, there are chances of loss of information. Also, technically automated specifications cannot be understood by business users. There are some tools like Concordion, FitNesse etc. which help in automating specifications. These automated specifications are called executable specifications and these can be executed and utilized by any stakeholder.

7) **Validating frequently**: System should be validated against executable specifications as often as possible. Executable specifications should be in sync with the system functionality at all the times. Any changes required in the specifications can be made after discussion with the business users. In case of large teams spread across multiple sites, the validation process might take long hours. Also, if there is any error, it is hard to detect. In such situations, it is suggested to set up multistage validation. Each site/team can have an isolated validation environment. Any changes made should be checked in this isolated environment first. If all tests pass in this environment, changes can be checked in central validation environment. (Adzic, 2011)
8) **Evolving a documentation system**: Executable specifications should provide a living documentation system. The specifications should be easy to read, understand, find, consistent and access. They should be continuously updated, always representing current requirements. Any stakeholder and team member should be able to access executable specifications and understand them.

4.3.5.2 Benefits of Specification by example

Specification by example offers various benefits in software development.

Living documentation: Living documentation system is achieved by executable specifications. This system is understandable by everyone in the team. Any changes made in the specifications are reflected immediately by this system and communicated across all team members.

Shared understanding: Specification by example leads to shared understanding among team members. Collaboration, being the main pillar of specification by example leads to knowledge sharing and avoids miscommunications or misinterpretations.

Continuous validation: By means of executable specifications, system is continuously validated against requirements.

Business-IT alignment: Specification by example leads to business and IT alignment. Requirements specified collaboratively by both technical as well business representatives enable this alignment.

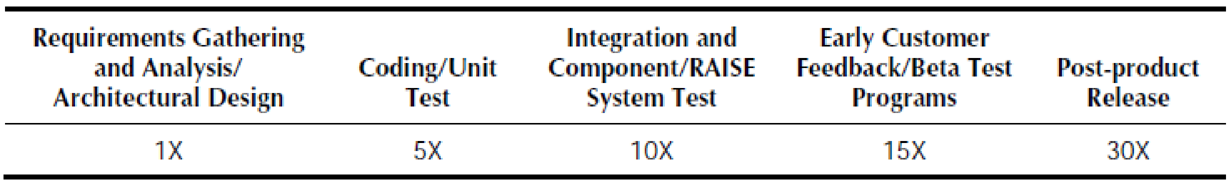
5 Hypothesis Formulation

In this chapter, hypotheses will be formulated and discussed. These hypotheses are based upon results of surveys presented in *Chapter 1.2-Motivation* and various benefits of agile testing methods described in literature and presented in preceding chapters. First detailed motivation behind use of agile testing methods is discussed and then hypothesis based upon that motivation is formulated.

These hypotheses discuss various perceived benefits of inclusion of agile testing methods inside V-model.

Motivation 1: Early and Often Testing

A study conducted by National Institute of Standards and Technology (NSIT) (2002) indicates that relative cost of fixing bugs in the beta testing phase is 15 times as compared to fixing it in the requirement analysis phase.



Requirements Gathering and Analysis/ Architectural Design	Coding/Unit Test	Integration and Component/RAISE System Test	Early Customer Feedback/Beta Test Programs	Post-product Release
1X	5X	10X	15X	30X

Table 5-1. Relative Cost of Fixing Defects
(Source: NSIT, 2002)

Table 5-1 shows that defects detected at design and coding phases are easier and less expensive to fix as compared to ones found during testing and production phases.

Reduction of cost of development and improvement in software quality are the major objectives of software development firms. One of the possible ways to achieve these objectives is early detection and fixing of bugs. This can be in turn achieved by conducting testing as early as possible in the software development lifecycle. Since agile methods encourage testing along with development, hence agile testing practices are the recommended solution.

Hypothesis 1: “***Agile testing methods when embedded inside V-model will lead to early and often testing and hence bugs being detected earlier in the software development life-cycle”***

<u>Motivation 2</u>: Regular Customer Feedback

‘Paradox of Expertise’ in psychology states that an expert sometimes thinks to know things that he actually doesn’t really know. The ‘You know it all’ conception of experts prevents them from further learning (Eikenberry, 2009). Sometimes experts think that they know everything and this prevents them from listening to others having lesser experience (Eikenberry, 2009).

Another case when expertise paradox occurs is when experts cannot describe their actions in detail. When an expert solves so many problems in a domain, their solving becomes hardwired in brain and it becomes automatic. They can solve more problems, but cannot describe in detail step by step process of solving. ("The Expertise Paradox", 2011)

Sometimes end users of software cannot express their implicit knowledge in a way that developers can understand. A user cannot decide and express his needs until and unless he has a look at the working software. A stakeholder after looking at the working software may realize that this was not exactly what he really wanted. Also, it is very rare that requirements remain fixed during course of software development. Customer requirements keep on changing because of technology changes or market changes.

Hence it is really important to involve customer in the software development and get regular customer feedback.

Traditional software development approaches engage customer only at the beginning and at the end of project which results in delayed feedback and misinterpretation of requirements. On the other hand, iterative software development approaches and agile methods engage customer at regular intervals. Product is developed iteratively and after every iteration, working prototype is shown and delivered to the customer.

Hypothesis 2: “*Agile testing methods when embedded inside V-model will lead to regular customer feedback and hence will be able to react quickly to changing customer requirements”*

<u>Motivation 3</u>: Reduced Development Time

Time-to-market is an important factor in today's competitive industry. The success stories of companies like Apple and Samsung, companies in the automotive sector can be attributed to this factor. Companies struggle hard and try to get their innovative product as quickly as possible in the market in order to gain early and maximum market share.

Agile methods lead to reduction in development cycle times and hence time-to-market. Frequent feedback, testing, and shorter releases are some of the techniques employed by agile methods which reduce development times.

This view is supported by results of surveys conducted by VersionOne and Ambysoft which are presented in Chapter 1.2-Motivation. This leads to third hypothesis related to benefits of adoption of Agile testing inside V-model.

Hypothesis 3: "*Agile testing methods when embedded inside V-model will lead to reduction in development cycle times*"

<u>Motivation 4</u>: Better Product Quality

In today's world, customers are very demanding and expect excellent quality product.

Agile testing methods build quality in by techniques like continuous integration, pair programming, exploratory testing, test driven development, refactoring, on-site customer etc. Testing is considered with priority instead of last step in the development cycle.

Hypothesis 4: *"Agile testing methods when embedded inside V-model build better quality product"*

The hypotheses formulated above will be proven true or false with the help of real world case studies. These case studies making use of agile testing methods inside traditional approaches will be analyzed in order to examine if the merits discussed in literature regarding adoption of agile methods are also practically true.

6 Towards Hybrid Approaches for Software Development

In this chapter, hybrid approaches combining traditional and agile methodologies for software development have been discussed. The need for such hybrid approach has been explained and extended V-models with embedded agile testing methods have been presented.

6.1 Agile vs. Plan-Driven Approaches

Boehm & Turner (2003) have identified home grounds in four areas namely Application, Management, Technical and Personnel for agile and traditional approaches where they work best.

Application: Agile methods work best in smaller projects of size up to 40 members, however there have been cases where agile methods have been able to scale up. Traditional methods on the other hand work well in larger and complex projects. In terms of environment, agile methods are most appropriate for turbulent, high change environment where requirements keep on changing. Traditional methods are best suited for projects with stable requirements with change rate of only 1 percent per month. (Boehm & Turner, 2003)

Management: Agile and plan-driven methods differ in the way they are managed. Agile methods depend upon close customer interaction. They stress upon regular customer involvement and feedback. Product delivery is considered more important than planning. Team's tacit knowledge is the base for planning rather than documentation. Traditional methods on the other hand are based upon formal contract between customer and developers. Extensive planning mechanism and documentation is used upfront in order to avoid any risk. (Boehm & Turner, 2003)

Technical: In agile methods, development takes place in short iterative cycles delivering potential usable product at the end of each iteration. Simplicity is the basis of agile methods like XP where simple architecture and design is stressed upon. In techniques like TDD and ATDD, tests are written before code and tests drive development. Plan-driven methods depend upon upfront requirements gathering and analysis. Most of the requirements are explained in the beginning of the project and agreed upon. In traditional methods, testing occurs as the last phase of software development and hence they suffer from expensive ‘late design breakage’ which has already been explained in Figure 2-2.

Personnel: Agile methods’ success depends upon Collaborative, Representative, Authorized, Committed and Knowledgeable (CRACK) customers. On-site customer presence is considered very important. Plan-driven methods also benefit from the presence of CRACK customers but it is not the requirement. Agile methods have the dependency on highly experienced and skilled developers – 30 percent full-time level 2 and level 3 resources (See Chapter 1Appendix C for description of personnel levels). Traditional methods can work with mixture of low skilled and high skilled resources. In terms of culture, agile approaches work in a chaotic environment, whereas plan-driven approaches work in an environment with clearly defined roles and responsibilities. (Boehm & Turner, 2003)

6.2 Need for Hybrid Approaches

While agile and plan-driven methods work best in specific project environments, question arises why do we need hybrid approaches combining agile and plan-driven methods. Boehm (2002) argues the importance of hybrid approaches outside the home grounds of agile and plan-driven methodologies. There is a need for a balanced approach which reaps the benefits of two and compensates their weaknesses (Boehm & Turner, 2003).

Various surveys indicate rising popularity of hybrid approaches. A report by Forrester Research indicates this trend. In one of the surveys conducted by Forrester Research in 2009, 39% of the respondents said that they mix different agile methodologies and 35% said that they mix agile and traditional methods. (West, Grant, Gerush & D’Silva, 2010)

Below are some of the reasons that demand use of hybrid approaches:

1) **Need for balance**: There is a need for balance to be maintained between agility and stability. Agility provides flexibility and space for innovation, stability on the other hand provides assurance and risk prevention (Vinekar, Slinkman & Nerur, 2006). Agility leads to chaos, stability leads to bureaucracy. ‘Neither agile nor plan-driven methods provide a methodological silver bullet’ – Boehm & Turner (2003). Successful organizations need both agility and discipline. (Boehm & Turner, 2003)
   Vinekar et al. (2006) suggest ambidexterity as the possible solution. An ambidextrous organization with agile and traditional subunits will be able to respond to

changing requirements without compromising stability. It will be able to provide explorative[6] as well as exploitative[7] abilities.

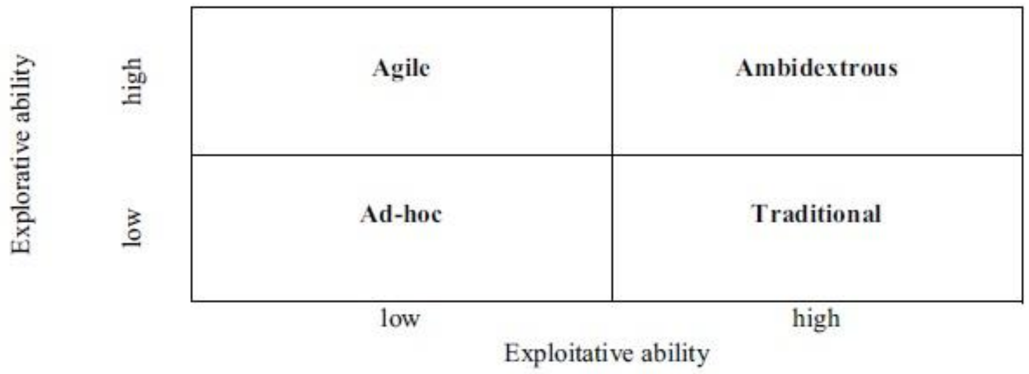


Figure 6-1. Ambidexterity as the solution
(Source: Vinekar et al., 2006)

2) **Compensation for each method's weaknesses**: Hybrid approach combining both agile and traditional software development processes in a right way can compensate for each method's weaknesses and provide maximize benefits from both. It can strike a perfect balance on below aspects:

   Planning: Agile methods employ minimum planning techniques which might work in smaller projects but in larger projects, considerable planning is required to avert any future risks. Plan-driven methods employ huge upfront planning techniques which results in huge effort being spent and delayed time-to-market. Hybrid approach combining planning techniques of agile methods with those of traditional methods can maximize benefits resulting in risk aversion and faster time-to-market.

   Documentation: Agile methods rely on tacit knowledge of the team members (Boehm, 2002). Hence if some team member leaves the team or company, there is a risk of losing this knowledge. Tacit knowledge also doesn't work well in large projects and organizations. Traditional methods on the other hand rely too much on documentation which becomes obsolete and needs to be changed every time there

[6] Explorative innovations are radical innovations, designed to meet demands of emerging customers and markets. They are based upon acquiring new knowledge and skills. A non-hierarchical structure and informal communication in an organization promotes explorative innovations. (Jansen, Van Den Bosch & Volberda, 2006)

[7] Exploitative innovations are incremental innovations, designed to meet demands of existing customers and markets. They are based upon existing knowledge and skills. A hierarchical organization with centralized decision making promotes exploitative innovations. (Jansen, Van Den Bosch & Volberda, 2006)

is change in requirements. Hybrid approach with neither too less nor too much documentation can provide a viable solution.

Risk management: Both agile and traditional methods employ different risk management strategies. Agile methods avert risk by frequent delivery, regular feedback and continuous integration. Traditional methods avert risk by documentation, planning and well defined architectures. Hybrid approaches can prove beneficial by combining risk aversion strategies of both the methods.

Customer involvement: As explained earlier, success of agile methods depends upon availability of CRACK customers. In some situations, clients might be unavailable or not willing to get too much involved into the project. In such situations, there are high chances of failure of agile projects.

One of the examples of such failures is C3 project, the first XP project which failed because of the replacement of on-site customer in the middle of the project. New on-site customer was not able to define user stories as clearly as the old on-site customer which led to project being cancelled. (Boehm & Turner, 2003)

Plan-driven methods only involve customer at the beginning and at the release of software which doesn’t work in situations with changing requirements. Hybrid approach which is not too much dependent upon customer for its success and involves customer at regular intervals will be the optimal solution.

3) **Need for flexibility in larger projects**: Apart from high assurance provided by plan-driven methods, today’s fast paced market demands the need for flexibility and speed in larger and complex projects. Agile methods are flexible to changing environment. Hence hybrid approaches combining agile methods in a traditional approach can provide a solution.

4) **Requirement of appropriate mix of testing techniques**: Both agile and plan-driven methods use different testing techniques for product quality. Agile methods use constructive activities which build quality in the product (Itkonen, Rautiainen & Lassenius, 2005). Developers perform tester role too and test their own code. Traditional methods use destructive testing techniques where independent testing team tests product with the purpose of finding errors in it.

   Itkonen et al. (2005) have contrasted between heartbeat quality assurance activities of agile methods and release quality assurance activities of traditional methods. Heartbeat quality assurance activities build quality in product during development or construction phase. Agile techniques like continuous integration, TDD, Pair-programming, refactoring etc fall into this category. Release quality assurance activities test product at the end of release and are performed by an independent testing team. Traditional methodologies mostly employ this technique. (Itkonen et al., 2005)

Testing is an important aspect of software development. Both agile and traditional methods can benefit by adopting each other's testing techniques.

6.3 Challenges of Hybrid Approach Implementation

Proper care needs to be taken while combining agile and traditional approaches since the two contrast each other in various dimensions. Adoption of agile methods inside traditional models poses various challenges which need to be considered and tackled accordingly in advance. Nerur, Mahapatra & Mangalaraj (2005) have identified these challenges occurring at four levels – management and organizational, people, process and technology.

1) **Management Challenges**: Adoption of agile methodologies poses significant management challenges which can further be categorized into three levels – management style, management attitude and Reward systems

   Management style: Traditional methods employ 'command and control' style of management. Software is developed in phases with assigned roles and responsibilities for each phase. Management structure is hierarchical with top management involved in big decisions. Agile methods on the other hand employ 'leadership and collaboration' style of management. Stakeholders and developers take part in decision making process. Teams are self-organized with developers having decision autonomy on the assignment of tasks. Hence, it poses a significant management challenge to shift from command and control style of management to leadership and collaboration style of management. (Nerur et al., 2005)

   Management attitude: Project manager in traditional methods acts as a planner and controller, whereas in agile methods, he acts as a facilitator and coordinator (Nerur et al., 2005). He plays roles of protector and coach, guiding team and resolving barriers faced by it (Boehm & Turner, 2005). Adoption of agile methods needs change in project manager's role and attitude. Sometimes, managers find it difficult to give up the autonomy and authority they enjoyed earlier in traditional methods (Nerur et al., 2005).

Reward systems: Adoption and transition to agile methods needs changes in reward systems in an organization. Team-work is an important aspect of agile development. In contrast to individual contributors in traditional processes, we have team-contributors in agile processes (Boehm & Turner, 2005). Reward systems need to be changed in order to recognize and reward team-contributors.

Organizational culture: Schein (2010) has defined culture in his book 'Organizational Culture and Leadership' as "a pattern of shared basic assumptions learned by a group as it solved its problems of external adaptation and internal integration…". Schein (2010) mentions one of the sources of culture as the learning experiences of group members as their organization evolves. Hence the more established and mature the organization is, the more deeply rooted its organizational culture is.

Organizational culture affects the actions and behavior of people (Nerur et al., 2005). It has significant impact on their decision making processes, strategies, innovation practices and planning and control mechanisms (Nerur et al., 2005). Culture change is difficult and time-consuming (Schein, 2010) which makes the adoption of agile methods in existing processes difficult. It is even more harder for legacy systems.

2) **<u>People Challenges</u>**: People are the key factor in software development. Boehm & Turner (2003) have righty said "Software engineering is done of the people, by the people and for the people." People challenges are categorized into below levels according to the roles:

Developers' issues: Developers in traditional methods need to adapt to agile practices like TDD, pair-programming, continuous integration to name few. This needs change in mindset and requires time till the developers get used to using these techniques and methods.

Testers' issues: Testers in traditional approaches are used to working in an independent team. Hence they find it difficult to adapt to working in a collaborative manner with developers. Testers are used to writing test cases from lengthy requirements documents; hence they find it difficult to write test cases from user stories and cards.

Customers' issues: Agile methods rely on availability of CRACK customers. Many times it is difficult to find dedicated and knowledgeable clients who are willing to take active part in software development. Hence customer involvement needs to be optimized while adopting agile practices otherwise it poses a threat to project success.

Staff issues: Agile methods' dependency on experts poses another challenge to their adoption. It is sometimes difficult to find sufficient number of experts for software development team (Nerur et al., 2005).

3) **<u>Process & Environment Challenges</u>**: Agile and traditional methodologies differ in many aspects with respect to their processes which poses another challenge while bringing the two methods together in an organization.

Contrasting processes: Agile and traditional methods have contrasting processes in terms of release cycles, process progress metrics, development practices and tools. In contrast to traditional methods, agile methods emphasize development in short iterative cycles with frequent deliveries. The two methodologies also vary in the way requirements are defined. The shift from phased approach to development to iterative approach is difficult.

Compliance adherence: Another challenge for adoption of agile methods is requirement of organizations for standards and compliance adherence. Agile methods with emphasis on less documentation and processes, normally don't qualify for process ratings and certifications (Boehm & Turner, 2005). However CMMI, ISO and compliance to other process standards is required in case of safety critical areas and software (Boehm & Turner, 2005). Hence organizations adopting agile methods need to strike proper balance so that their processes adhere to standards if required.

Nerur et al. (2006) have used the framework for organizational change by Adler and Shenhar (1990) to explain the level of impact and time required by organizations to tackle various challenges described above.

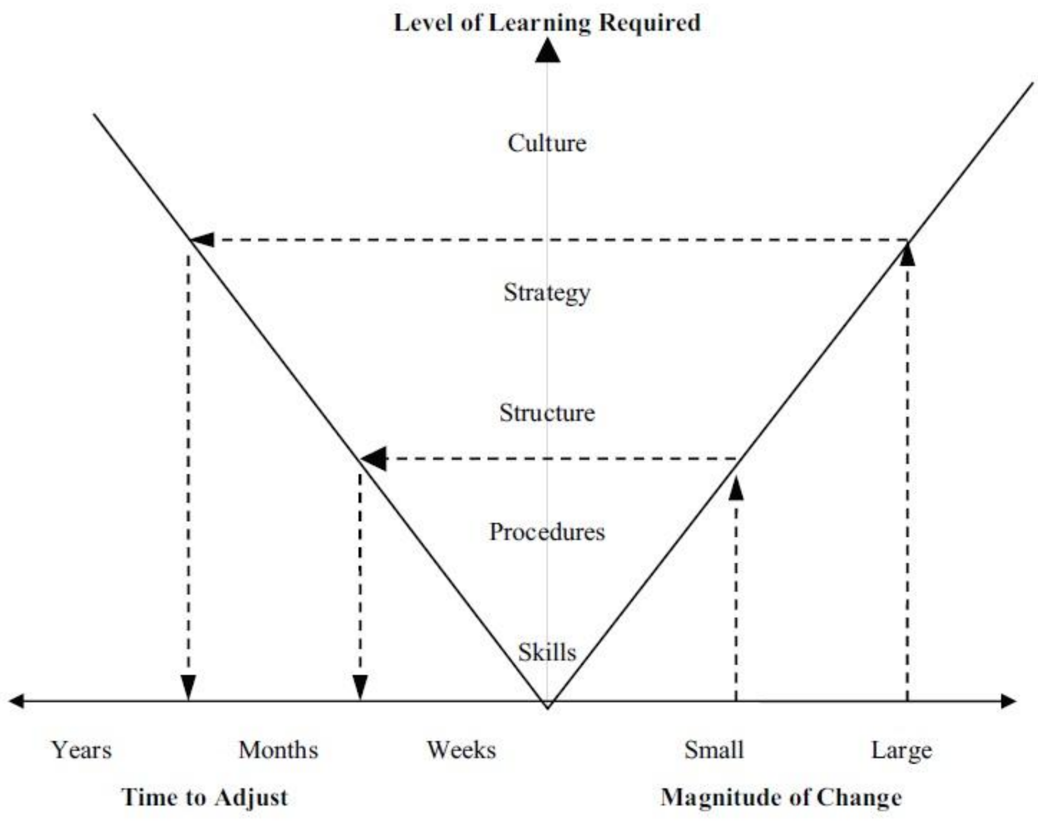


Figure 6-2. Framework for Organizational Change
(Source: Adler & Shenhar, 1990)

Of the three categories of challenges described above, process challenges occur at the skills and procedures level, hence the magnitude of change is small and the time required for adjustment is also short. However management and people challenges occur at the culture and strategy level and hence the magnitude of change is large and the time required for adjustment is also long. (Nerur et al., 2006)

7 Extended V-Models

In this chapter, hybrid approaches for adoption of agile testing methods inside traditional V-model have been suggested. V-model has been extended to include agile testing approaches. The presented extended V-models try to keep challenges discussed in *chapter 6.3-Challenges of Hybrid Approach Implementation* to a minimum. Care has been taken that these models require minimum changes to existing processes and roles and hence are easier to adopt in an organization.

7.1 SCRUM and V

This section discusses the hybrid approach of blending Scrum inside V-model. Although Scrum doesn't provide any specific testing techniques but it provides the agile process management framework (Rong, Shao & Zhang, 2010). Scrum prescribes management practices which when implemented inside projects managed by traditional approach can lead to flexibility and stability at the same time. It can provide benefits of both agile as well as traditional development. The suggested approach has been named as V-SCRUM-V inspired from Water-Scrum-Fall approach suggested by West (2011). The hybrid approach is depicted in Figure 7-1.

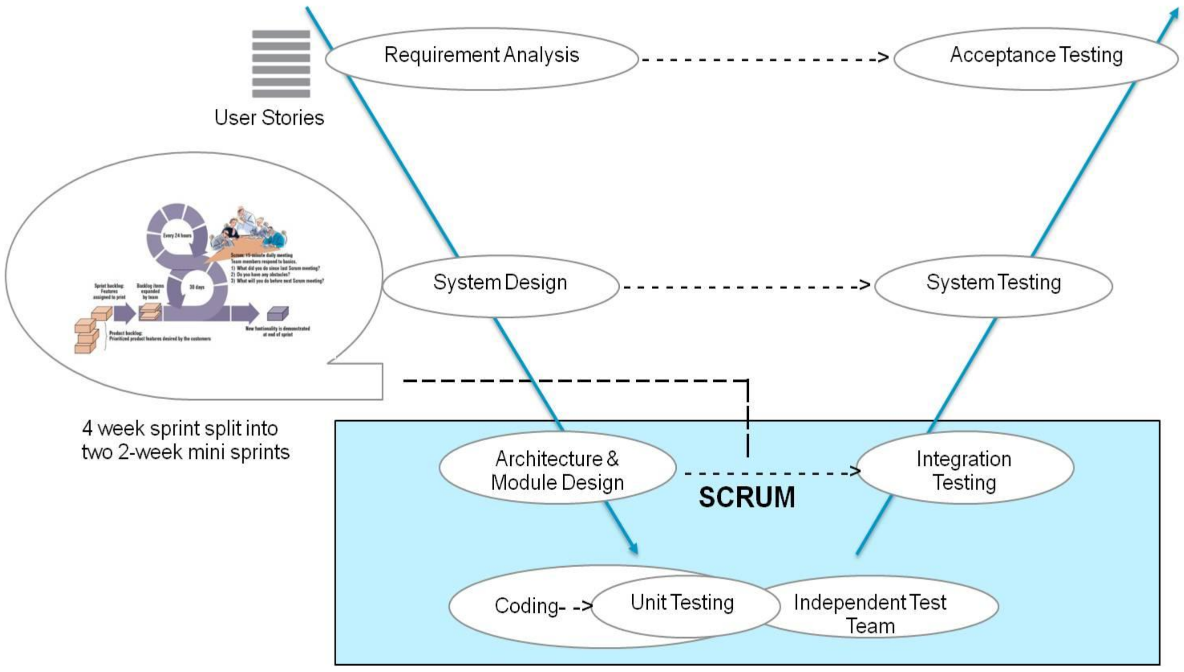


Figure 7-1. V-SCRUM-V Model

7.1.1 V-SCRUM-V Model in Detail

The hybrid approach described in V-SCRUM-V model uses V-model framework and blends Scrum inside it. The initial phases like requirements analysis and design are based upon traditional concepts. However, coding and testing phases are based upon iterative concepts of agile development. The later phases of system testing and acceptance testing are again based upon traditional development concepts. Strict V-methodology and strict Scrum is not followed, but best practices from both the methodologies have been combined in all the phases. Hayata & Han (2011) have suggested a similar model for software development which is based upon Waterfall-Up-Front and Waterfall-At-End concept. However, they haven't provided detailed explanation of the model and the possible ways of its implementation.

In order to support the development and testing phases, requirements from the users need to be presented in the form of user stories. Acceptance criteria could be included in user stories so that developers and testers are on same page (Schiffmann, 2012). Unlike traditional development methodology, 100% requirements need not be defined in the requirement analysis phase. However, critical requirements should be defined upfront. This leads to flexibility and ability to accommodate changing requirements. Overall system design specifying major components should be completed before coding. Detailed architecture design evolves during the sprints.

Scrum methodology should be followed during the coding and testing phases. User stories should be prioritized and presented in a product backlog. Unlike Scrum, independent testing team should perform product testing. The usual sprint duration of 4 weeks can be divided into two 2-week mini sprints of development and testing. The role of independent testing team and the reason for its inclusion is described in chapter 7.1.4-Independent Test Team. Apart from following Scrum inside development team, it can also be followed inside test team. The next chapter describes the use of Scrum inside test team. After development and testing, the overall system testing should be conducted. User acceptance testing in the customer environment should also be performed and after that product is ready for delivery. Incase customer is interested, potential product increments can be released to the customer at an earlier stage during Scrum sprints.

7.1.2 SCRUM inside Test Team

In V-SCRUM-V model, development and test teams follow Scrum practices and product is developed in iterations known as sprints which deliver potentially shippable product at the end of each sprint. After 2 weeks of mini sprint, development team can hand over developed user stories to the independent test team for testing which again follows Scrum and performs testing for another 2 weeks. Scrum practices based upon the principles of transparency, inspection and adaption can lead to improved management practices inside test team. The implementation of Scrum inside test team has been inspired by an article from Schiffmann (2012) in Agile Record magazine.

Test Product Backlog: After the mini sprint of 2 weeks in development team, Quality Assurance Manager or Test Manager can pick developed user stories and present them in a separate test product backlog or same product backlog utilized by the development team can also be used.

Sprint Planning meeting: Before testing in a mini testing sprint begins, the test product backlog can be presented to all the test team members by the Quality Assurance Manager or Test Manager where each member decides on its own the stories to test.

Daily Scrum test meeting: Daily Scrum meetings could be performed inside the test team. Quality Assurance Manager or Test Manager could organize such meetings. Test lead can lead these meetings. Daily Scrum meetings can help in identifying any impediments to testing and resolve them on time (Schiffmann, 2012).

Sprint Retrospective meeting: At the end of mini testing sprint, a retrospective meeting can be organized in which all test team members can provide their feedback and areas of improvement can be identified.

Test Boards and Burn-down Charts: Schiffmann (2012) suggests use of test boards similar to task boards which present all the items to be tested along with their status. Burn-down charts can also be used in the testing team (Schiffmann, 2012).

7.1.3 Mapping of Roles

The V-SCRUM-V model provides benefits of traditional development in terms of clear roles and responsibilities. Since Scrum is followed inside development and testing phases, hence care needs to be taken while defining roles. Table 7-1 suggests one way to map traditional roles with Scrum roles, avoiding any duplicity.

V-SCRUM-V Model Role	SCRUM Role
Business Analyst / Product Manager	Product Owner
Software Architect	Scrum Team
Developer	Scrum Team
Tester	Scrum Team
Development Manager	Scrum Master in development team
Quality Assurance Manager / Test Manager	Scrum Master in test team

Table 7-1. Mapping of V-SCRUM-V and SCRUM Roles

The role of Product owner in Scrum can be performed by Business Analyst or Product Manager since he is responsible for gathering customer requirements and has the clear understanding of customer needs. A representative from customer should also be involved during the development process to provide frequent feedback from user perspective.

We have cross-functional team inside Scrum where developers are also responsible for testing. Since testers are specialized in testing techniques and they test application from user point of point, hence clear roles of developers and testers are suggested in hybrid approach.

The role of Scrum Master inside development team can be performed by Development Manager after proper training. Since within an organization in a traditional setup, development manager is familiar with traditional plan-driven concepts, hence after proper Scrum training, he should be able to blend and utilize best practices from both the approaches. He should be responsible for making sure that proper Scrum process is followed within the team and should remove any impediments being faced by the team. Similarly, Quality Assurance Manager or Test Manager can serve the role of Scrum Master in test team.

7.1.4 Independent Test Team

As described earlier in chapter 6.2-Need for Hybrid Approaches, agile methods employ constructive testing techniques as compared to traditional methods which employ destructive testing techniques. V-Scrum-V model suggests use of an independent testing team. This team is independent in the sense that it is a separate team which works closely with the development team. This team should be responsible for testing user stories at the end of 2-week development mini sprint. Itkonen et al. (2005) and Nawaz & Mallik (2008) also support use of independent test team.

In psychology, ‘Theory of Cognitive Dissonance’ states that people sometimes have conflicting cognitions (knowledge, opinion, or belief about the environment, about oneself, or about one’s behavior) which leads to dissonance (feeling of discomfort) and people try to reduce this dissonance (Festinger, 1962). Applying this theory to software development, it is difficult for a developer to detect and accept bugs in his own code. An independent tester is required who will test the application with an intention of finding bugs in it (Nawaz & Mallik, 2008). A professional tester can test the application from customer perspective.

It is also important that test team should work in collaboration with the development team. Nawaz & Mallik (2008) have suggested a communication model for effective working of an independent testing team in agile development. This communication model suggests testers, developers and customers working very closely and all three actors participating in all the meetings. Pair testing is one of the recommended techniques where testers sit along with developers and test the system. (Nawaz & Mallik, 2008)

7.1.5 Benefits of V-SCRUM-V Hybrid Approach

The V-Scrum-V hybrid approach discussed above offers various benefits in software development. Traditional organizations attempting to go agile can definitely benefit from this model.

1. **Iterative development**: V-SCRUM-V approach offers a way of including iterative development inside traditional setup. This model incorporates flexibility to respond to changing customer requirements and avoids unexpected delays caused by late design breakage.

2. **Prioritized requirements**: By means of product backlog and prioritized user stories, more riskier and valued requirements can be developed first. If customer wants any functionality earlier, it can be developed first in a sprint.
3. **Improved visibility**: The hybrid model leads to improved visibility by means of task/test boards and daily sprints. This can lead to effective resource handling within the team.
4. **Easy mapping of roles**: The hybrid approach presented above offers a simple way to blend agile practices inside existing V-model. There is direct mapping of roles of two approaches without need of additional roles.
5. **Provides a base**: Since V-SCRUM-V model provides a framework for including Scrum management practices inside V-model, it can be extended to include other agile testing techniques like TDD, ATDD, FDD etc.

7.2 TDD inside V-Model

Test first programming concepts of XP can be followed inside traditional V-model. This section describes blending of TDD, one of XP practices inside V-model. It presents an extended V-model and discusses the various benefits of hybrid approach.

7.2.1 TDD-V Extended Model in Detail

TDD concepts can be adopted inside V-model. Figure 7-2 shows an extended V-model with TDD embedded in it.

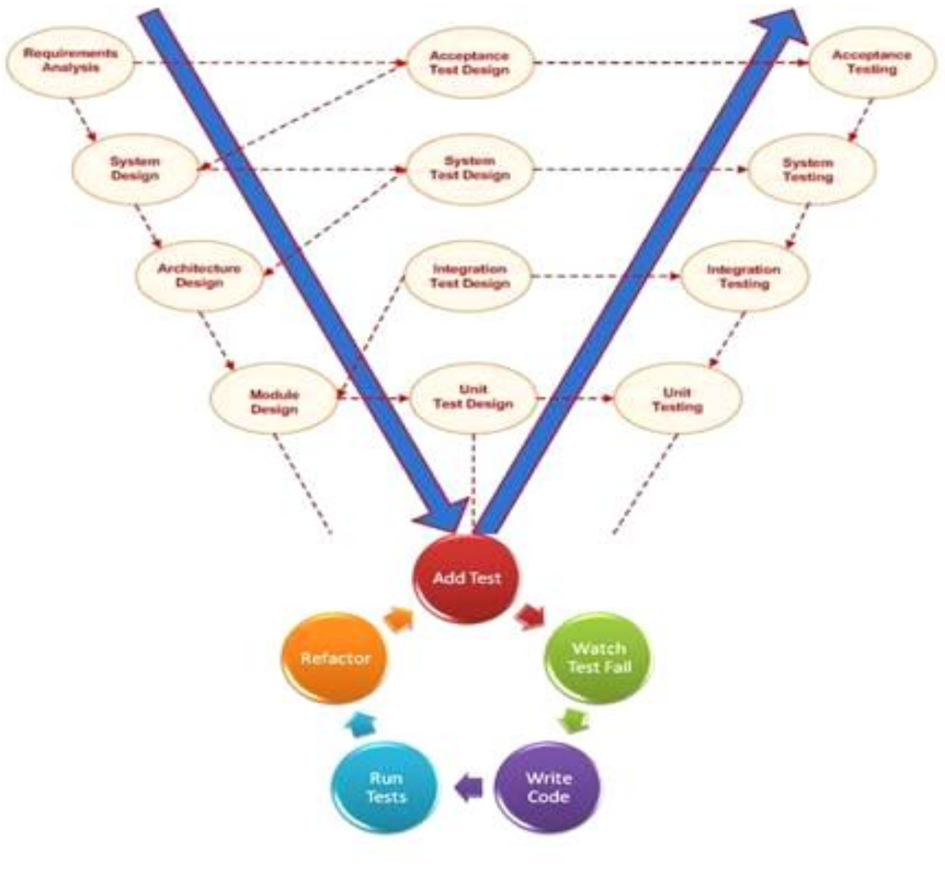


Figure 7-2. TDD-V Extended Model

In the extended V-model shown above, TDD has been implemented inside coding phase. All other phases follow V-model for software development. The process begins with requirements analysis, followed by system and architecture design. The coding phase has been modified to include TDD practices. In the coding phase, developers write tests before writing code. They follow TDD cycle of *Red* -> *Green* -> *Refactor* and make sure none of the tests fail after implementation. V-model phases of validation are followed afterwards. Unit testing, integration testing, system testing and user acceptance testing are conducted to deliver quality product.

In order to achieve maximum benefits of TDD, another possible suggestion is shown in Figure 7-3. In this model, the TDD-V extended model shown in Figure 7-2, is followed in iterations. The process starts with initial requirements gathering from the user. The requirements are divided into sets and for each set of requirements; TDD-V extended model is followed in iterations. This leads to iterative development and customer feedback can be gathered after each iteration. Any change in requirements can also be accommodated in next iteration. Number of iterations depends upon the complexity and number of requirements and the capacity of development team.

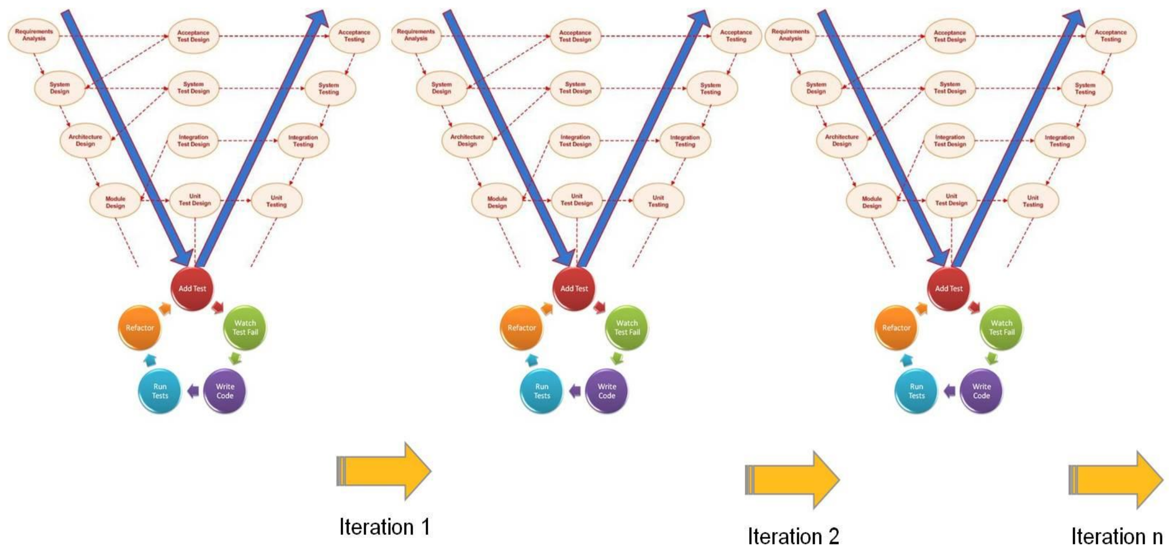


Figure 7-3. TDD-V Extended Model with Iterations

7.2.2 Mapping of Roles

TDD-V extended model doesn't need any specialized roles for its implementation. It just needs a change in mindset of developers. They need to get used to writing tests before writing code.

7.2.3 Benefits of TDD-V Extended Model

TDD when embedded inside V-model provides various benefits:

1) **Iterative development**: TDD-V extended model is based upon iterative development. Product is developed in iterations and after each iteration; product increment can potentially be delivered to customer.
2) **Ability to accommodate changing requirements**: TDD-V extended model is able to accommodate changing customer requirements. After iteration, developed increment can be shown to customer and if it doesn't meet his expectations or if requirements have changed meanwhile, they can be accommodated during next iteration.
3) **Cleaner design and code**: TDD-V extended model improves development practices. It leads to cleaner design and code by avoiding unnecessary code being written by developers.

4) **Based upon existing roles and responsibilities**: TDD-V extended model doesn’t introduce any new roles and responsibilities and is based upon existing roles of V-model. This makes its implementation easier.

7.3 ATDD inside V-Model

Like TDD, ATDD can be embedded inside V-model. However, ATDD is based upon acceptance tests as compared to unit tests in TDD. Use of acceptance tests forces developers to write code from customer perspective and makes testing an integral part of development.

7.3.1 ATDD-V Extended Model in Detail

Figure 7-4 shows ATDD-V extended model.

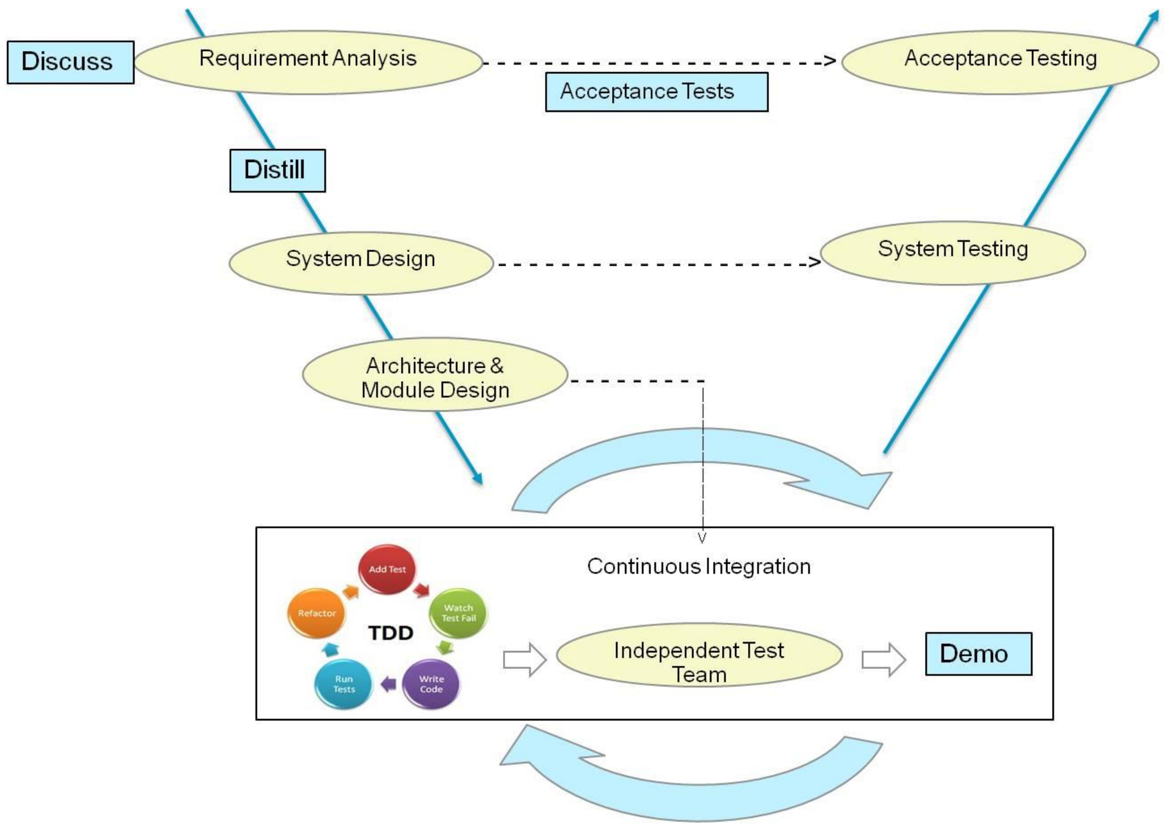


Figure 7-4. ATDD-V Extended Model

Like traditional V-model, ATDD-V extended model starts with requirements analysis phase in which requirements are gathered from users. In this phase requirements from users are expressed in the form of acceptance tests and these tests are written by users in collaboration with domain experts and testers. This phase can be mapped to ‘Discuss’ phase in ATDD. After discuss phase, tests are represented in table-based or text-based format (Distill phase). After this, high level architecture of system is designed by architects. The main phase which has been modified to accommodate ATDD inside V-model is coding phase. In this phase, test-first programming concepts are followed. Developers follow TDD and write tests before code. Small subsets of acceptance tests are taken iteratively through TDD cycle and developed increment is tested by an independent test team working closely with developers. After testing, the developed increment is shown to the customer (Demo phase) and any change needed is accommodated. Again in next iteration, small subset of acceptance tests is taken and this process is followed iteratively. After the product has been developed, overall system testing should be conducted. During the last phase of user acceptance testing, all the acceptance tests written by customer during the beginning of the project are run in the customer environment.

7.3.2 Mapping of Roles

ATDD-V extended model doesn’t need any roles different from traditional V-model. However, there is a need to change developers’ mindset. They need to get accustomed to TDD and work with acceptance tests. Another requirement is high customer involvement. Customers need to be more dedicated. They need to write acceptance tests with testers and domain experts and also take active part in demo phase.

7.3.3 Benefits of ATDD-V Extended Model

ATDD when embedded inside V-model leads to better development practices and hence consequently better product quality. Some of its benefits are listed below:

1. **Tests as a by-product**: ATDD-V extended model creates acceptance tests as a by-product. These tests can later be utilized by customers during user acceptance testing. Hence this model avoids rework.
2. **Added benefits**: Since ATDD-V extended model embeds TDD inside coding phase, hence apart from providing other benefits, it provides all the benefits of TDD also.

3. **More clarity**: In ATDD-V extended model, acceptance tests are written by customer. Hence there is no confusion between team members regarding expected behavior
4. **Accommodate changing requirements**: In ATDD-V extended model, acceptance tests are always evolving. If requirements change, customers can update tests and developers are forced to update code, otherwise tests will fail. Hence developers are indirectly forced to accommodate changing customer requirements.

7.4 FDD inside V-Model

FDD, being considered less agile because of defined roles and responsibilities and modeling activities, provides a good fit for adoption inside traditional model of software development. FDD works upon splitting of requirements into features and utilizes best practices for software development.

Abrahamsson, Salo, Ronkainen & Warsta (2002) have shown quality assurance activities of FDD during design by feature and build by feature iterative activities (Figure 7-5).

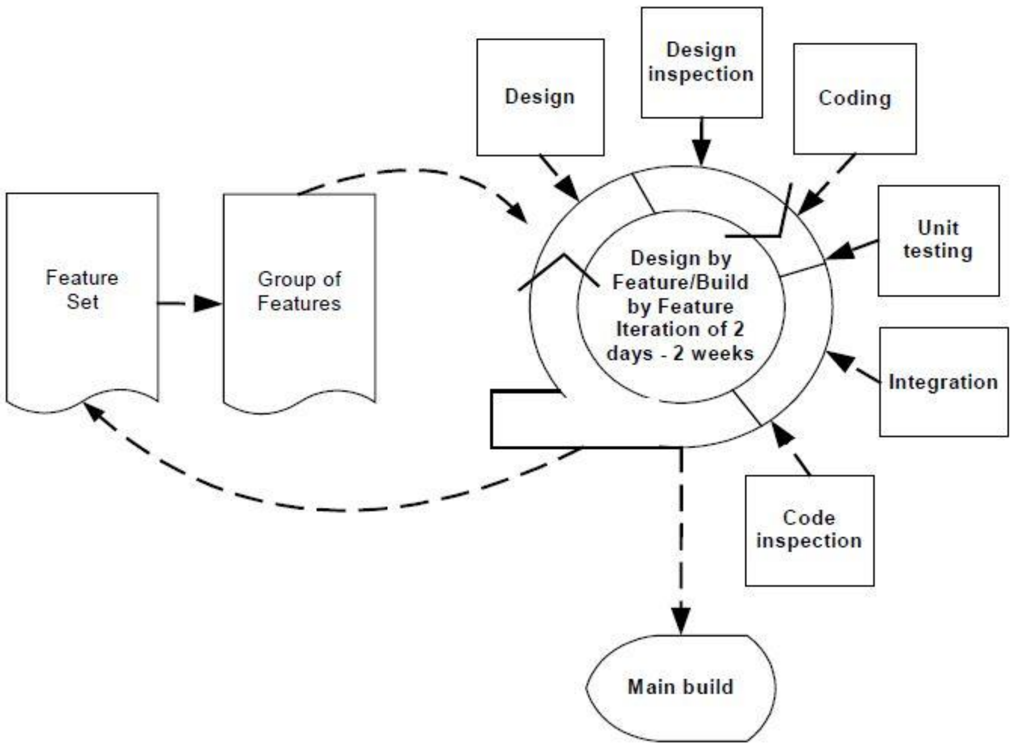


Figure 7-5. Quality Assurance Activities in FDD
(Source: Abrahamsson et al., 2002)

Figure 7-5 shows quality assurance activities during design by feature and build by feature cycles. FDD employs static as well as dynamic quality assurance activities. Static quality assurance activities include design and code inspection and dynamic activities include unit testing, integration testing etc. Simultaneous testing of features along with development is a benefit provided by FDD.

7.4.1 FDD-V Extended Model in Detail

This chapter describes in detail extended V-model with FDD embedded in it. Figure 7-6 shows the hybrid model. In order to embed FDD inside V-model some additional phases need to be added and some FDD phases can be mapped directly to V-model phases. Existing V-model phases have been highlighted in Green color and FDD phases have been highlighted in Blue color.

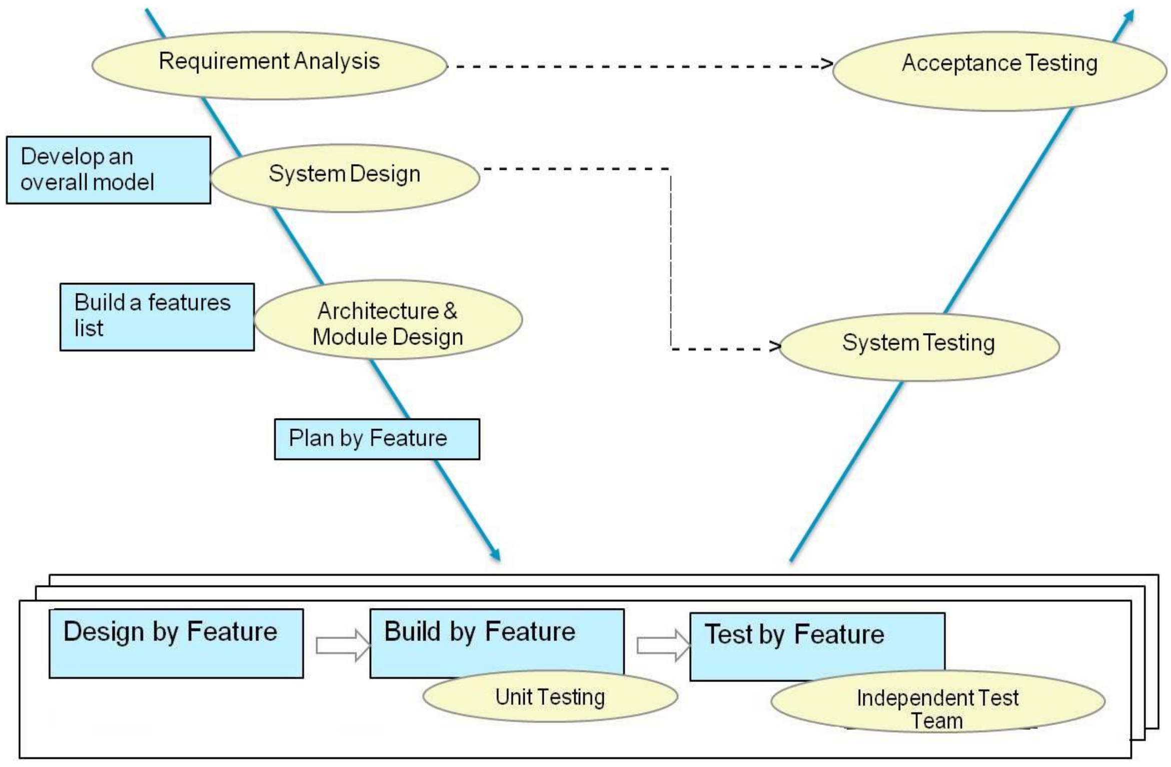


Figure 7-6. FDD-V Extended Model

The hybrid approach shown above starts with the normal requirements gathering process. High level requirements from customer are gathered and documented as in traditional process. System design phase of traditional V-model which involves creation of system design can be mapped to 'Develop an overall model' phase of FDD. In this phase, like in FDD, domain walkthroughs can be conducted by domain experts (users, business analysts).

It is suggested to involve testers also at this stage like in V-model. Testers can also participate in domain walkthroughs and can give their inputs for creation of domain object models. Based upon the information gathered from this stage, testers can also start writing system tests. The next phase includes architecture and module design which can be mapped to 'Build a features list' activity of FDD. Like in traditional V-model at this stage, system is decomposed into modules and class diagrams are created, similarly, in this phase, in FDD, domains are decomposed into subject areas and features list is prepared. Hence the two activities in V-model and FDD can be directly mapped to each other in hybrid model. Next FDD activity of 'Plan by feature' can be performed, in which class owners are identified and assigned features for development.

During the next phase, 'Design by feature' and 'Build by feature' activities of FDD are performed in iterations. New activity added in the hybrid FDD-V extended model is 'Test by feature'. During these activities, feature teams are formed by Chief Programmers (Senior Developers in V-model) and features are designed, developed and tested. In this hybrid model also, independent test team is suggested to perform feature testing as and when feature is developed. Developers can carry out other quality assurance activities of design and code inspection, and unit and integration testing. Continuous integration should be carried out to avoid any integration issues. In the FDD-V extended model, the usual 2-week iteration time per feature can be extended to include one more week for testing. At the end of iteration, product can be shown to customer and domain experts for their feedback and any changes in requirements and design can be accommodated in next iteration.

After the development phase, usual V-model phases of system and user acceptance testing can be followed in which the overall system can be tested by the customer representatives.

7.4.2 Mapping of Roles

It is easier to map roles in FDD and traditional V-model since FDD prescribes defined roles and responsibilities. Table 7-2 compares roles of FDD-V extended model with those of FDD.

FDD-V Extended Model Role	FDD Role
Business Analyst & Users	Domain experts
Senior Software Architect	Chief Architect
Senior Developers	Chief Programmers
Project Manager	Project Manager
Development Manager	Development Manager
Developer	Class Owner
Tester	Tester

Table 7-2. Mapping of FDD-V and FDD Roles

Business Analysts and users can serve role of domain experts in extended V-model. The senior most software architect can serve the role of chief architect. Similarly, senior developers having considerable amount of experience can act as chief programmers.

7.4.3 Benefits of FDD-V Extended Model

Embedding FDD into V-model offers various benefits as compared to traditional V-model. Some of the benefits are listed below:

1. **Easier adoption**: FDD being less agile is easier to adopt inside traditional plan-driven organization as compared to other agile approach. Mapping of FDD roles with existing V-model roles is also easier and doesn't need additional roles.
2. **Scalable**: The FDD-V extended model can be scaled for use inside larger and complex projects which provides an added advantage as compared to other agile approaches such as XP which are difficult to scale in larger projects.
3. **Best practices**: FDD uses best practices such as domain object modeling, developing by feature, feature teams, inspections, regular builds, configuration management, reporting of results etc. (Palmer & Felsing, 2001). Hence incorporation of FDD and consequently these practices inside V-model can improve software development process and software quality.

4. **Iterative development**: The design by feature and build by feature cycles of FDD build product iteratively. Hence FDD-V extended model brings iterative development into traditional V-model.
5. **Quality assurance activities**: As shown in Figure 7-5, FDD utilizes static as well as dynamic quality assurance activities. Apart from these activities, FDD-V extended model uses additional independent test team for conducting system and regression testing.

7.5 Kanban and V

The core Kanban principles can be applied inside V-model to improve software development process. Since Kanban accepts existing roles and responsibilities within an organization and doesn't introduce any new roles, it is a very effective method to be adopted inside V-model. As explained under *chapter 3.1-Inspection of Agile Methods for V-Model Suitability*, one of the criteria for selection of an agile method is that it should not disrupt V-model completely; Kanban method satisfies this criteria very well.

In this section, Kanban principles as stated by David J. Anderson and explained in *chapter 3.4.2.1-Kanban Software Development* have been adopted inside V-model and an extended version of V-model has been discussed.

7.5.1 Kanban-V Extended Model

Core five Kanban principles as explained by David J. Anderson are:

1) Visualize workflow
2) Limit work in progress
3) Manage flow
4) Make policies explicit
5) Improve collaboratively

The five principles can be applied for software development irrespective of software development methodology used. An extended V-model with Kanban principles applied can be seen in Figure 7-7.

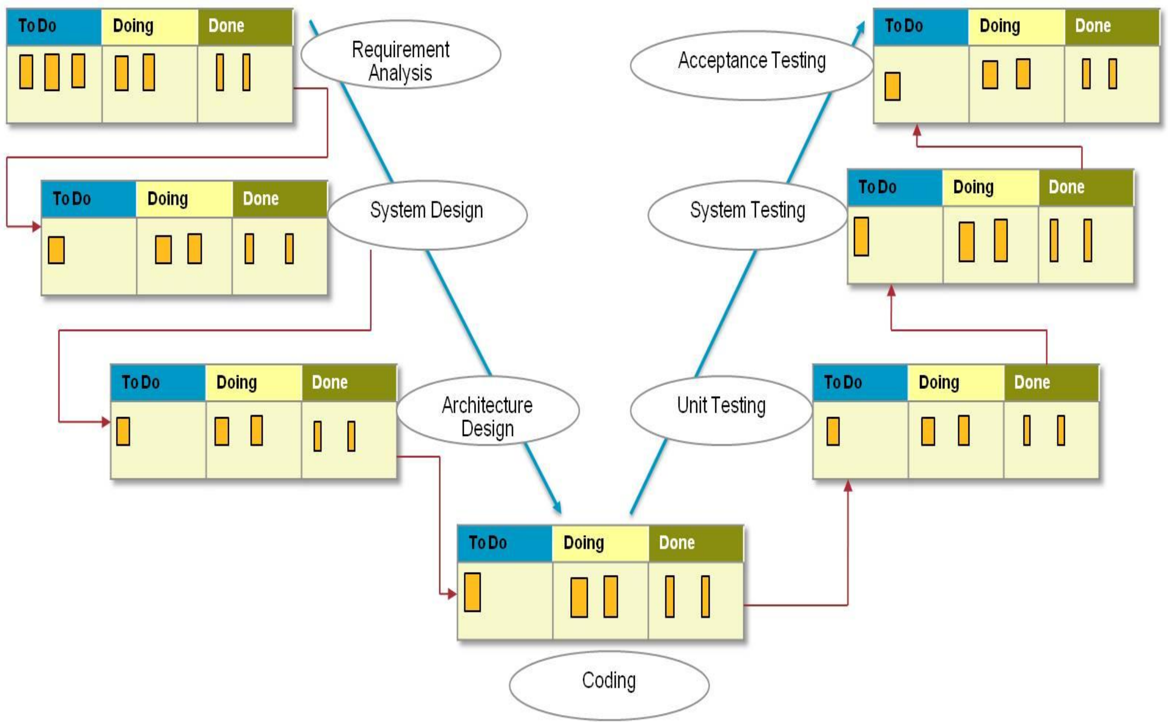


Figure 7-7. Kanban-V Extended Model

The Kanban-V extended Model shown above makes use of Kanban boards in all the V-model phases. A Kanban board is like a task board which represents user stories or work items by means of sticky notes or cards on a board or on a wall. The use of task boards in software development is as old as agile methods (Ladas, 2008). The main difference between task board and Kanban board is that Kanban puts a restriction on work in progress. It defines a limit of work items or user stories for each queue/column.

‘A task card without a limit is not a Kanban in the same way that a photocopy of a dollar bill is not money’ – Ladas (2008).

Limit for work items can be specified for each queue on Kanban boards. This limit depends upon how the user stories are decomposed into work items or tasks and it depends upon the capacity of the team. This limit should be decided initially after discussion within the team.

The Kanban-V extended model presented is inspired from ‘Kanban System for Sustaining Engineering’ by Anderson & Garber (2007) and ‘Lean + Agile Kanban’ concept of Hiranabe (2008).

One of the wastes in software development is waiting for things (M. Poppendieck & T. Poppendieck, 2003). Effective linking of processes together can reduce this waste (Riordain & Burgt, 2011). In the Kanban-V extended model, pull concept has been implemented. The next phase pulls work item from the previous phase if its To-Do list becomes empty. It makes the whole flow self-directing and reduces waste of waiting. For example, if the To-Do list in testing phase becomes empty, the items to be tested can be pulled from the Done queue of coding Phase with the maximum items being pulled dependent upon the limit set for the To-Do list queue. Within the same phase e.g. in coding phase, if the developer has finished coding the work items assigned to him, he can himself take the next work items from the To-Do list queue. The whole process and flow is continuous. After a specified time or when sufficient number of user stories have been developed and tested, a demo can be given to the customer for his feedback.

Anderson & Garber (2007) supplemented a Kanban system with colors representing quality of service for work items. For example, a blocked work item can be represented in Red color on Kanban board. Reason for blockage can also be indicated on the card. This makes it easier to identify bottlenecks and resolve them on time.

The Kanban-V extended model can be used along with Scrum method. Ladas (2008) has suggested a method to combine Scrum and Kanban in his essay Scrumban.

7.5.2 Benefits of Kanban-V Extended Model

The Kanban-V extended method presented in previous section offers some benefits of lean software development.

1) **Self-directing**: The whole process and flow is self-directing (Hiranabe, 2008). This leads to reduction in management overhead and waiting time between phases.
2) **Increased transparency**: The Kanban-V extended method based upon ‘Visualize workflow’ Kanban principle leads to increased transparency among team members. At any time, it is visible which team member is working upon which work item and if any team member is overbooked.
3) **Limits work in progress**: The method explained above sets a limit on number of work items in a queue; hence it limits work in progress and avoids piling of work items in a queue.
4) **Resolved bottlenecks**: By making use of colored cards, bottlenecks and issues can be easily identified and worked upon in time.

5) **Respect of existing roles and responsibilities**: The Kanban-V extended model respects existing roles and responsibilities with an organization and hence poses comparatively lesser challenges for its implementation.

7.6 Specification by Example inside V-model

Apart from agile projects, specification by example can be implemented inside traditional projects also. Implementation of specification by example in traditional plan-driven projects leads to defects being detected earlier in the product development lifecycle (Adzic, 2011). In this chapter, various suggestions will be provided regarding implementation of specification by example inside V-model.

7.6.1 Implementation of Specification by Example inside V-Model

Like other agile methods, implementation of specification by example inside V-model requires process as well as cultural changes. Below suggestions can be given in order to make the implementation process easier and smoother:

TDD or ATDD as the stepping stone: Implementation of specification by example becomes easier inside V-model if TDD or ATDD methods have already been implemented inside the team (Adzic, 2011). Hence as a starting step, TDD-V extended model or ATDD-V extended model can be implemented, followed by specification by example concepts.

Functional Automation: Adzic (2011) has suggested to start with functional test automation in an existing project. Test automation involves business users, testers and developers in the process and hence removes the barriers between them, making way for specification by example. Testers also get more time after automation, thus enabling them to take part in specification workshops. (Adzic, 2011)

Change team culture: In order to implement specification by example, team culture needs to be changed. One of the most important pillars of specification by example is collaboration between developers, testers and business users. In traditional V-model, analysts, developers and testers work independently and are not used to working together. In specification by example, all three roles work closely starting from the time goals are specified by the user. Adzic (2011) suggests avoiding use of agile terminology in an environment that is resistant to change.

Follow key process patterns: In order to successfully implement specification by example, it is important to follow key process patterns described in chapter 4.3.5.1-Key Process Patterns. The process patterns can be followed in combination with other agile methods in iterations.

8 Comparison of Extended V-Models

It is important to compare extended V-models discussed in previous chapter in order to assess their viability and suitability for adoption inside an organization. The comparison of extended models is directly dependent upon agile method used, hence the parent Agile testing methods should first be compared to each other before a comparison is made between extended V-models. In this chapter, first agile testing methods are assessed based upon the criteria defined in chapter 3.1-Inspection of Agile Methods for V-Model Suitability and other factors. Next extended V-models are compared based upon the same criteria. Finally, conclusions are drawn regarding best suited and viable extended V-model.

8.1 Comparison of Agile Testing Methods

In this section, agile testing methods are compared to each other on number of factors. The selection criterion which was used initially is utilized for comparison also.

1) **Quality assurance activities**: Since testing is an integral part of software development, it is important to compare quality assurance activities of agile testing methods.

   Scrum: Scrum doesn’t provide any specific guidelines on testing. It provides management guidelines and a framework within which other practices can be adopted. However, many companies practicing Scrum follow below quality assurance activities (Nawaz & Mallik, 2008)
   - Unit testing
   - Continuous Integration
   - Exploratory Testing

   As discussed in chapter 7.1.2-SCRUM inside Test Team, Scrum practices can be followed inside test team. It will lead to iterative testing in parallel with the development which will greatly impact product quality and improve the efficiency of test team. By following such a model, testers can work closely with the development team and high quality developed and tested product can potentially be delivered to the customer.

TDD and ATDD/Specification by example: TDD and ATDD or specification by example employ similar quality assurance activities. They make testing an integral part of development by forcing developers to write tests before code. Developers can’t go away with testing. By employing acceptance tests, they develop code from customer perspective.

Itkonen et al. (2005) have contrasted between heartbeat and iteration quality assurance activities. XP practices utilize heartbeat Quality Assurance activities and build quality into product during its implementation. Various XP heartbeat quality assurance activities are (Itkonen et al., 2005):

- Test Driven Development
- Continuous Integration
- Pair-Programming
- Acceptance Tests
- Refactoring

All these activities blend testing into development and lead to higher quality product being developed.

FDD: As shown earlier in Figure 7-5, FDD has strong focus on testing and utilizes number of quality assurance activities in its design by feature and build by feature phases. It is based upon static as well as dynamic quality assurance activities. ISTQB (2012) defines static testing as ‘Testing of a software development artifact, e.g., requirements, design or code, without execution of these artifacts, e.g., reviews or static analysis’ and dynamic testing as ‘Testing that involves the execution of the software of a component or system’. FDD utilizes static techniques such as design and code inspection and dynamic techniques such as unit testing and integration. Again these techniques are heartbeat quality assurance activities and build quality into the product.

FDD-V extended model has additional phase of test by feature in which an independent test team tests features s soon as they are developed. This makes testing a mandatory activity for readiness of feature.

Kanban: Like Scrum, Kanban also doesn't specify any quality assurance activities. It provides a framework and guidelines and within this framework, other technical activities can be implemented. Kanban-V extended model based upon principles of visualize workflow, limit work in progress and manage flow impact quality of product.

2) **Supported phases of software development**: All agile methods do not support and provide guidelines on all phases of software development lifecycle. For the comparison of extended models, it is important to compare which phases are covered by their parent agile methods.

Abrahamsson, Warsta, Siponen & Ronkainen (2003) have compared agile methods based upon the support for project management, supported phase of software development lifecycle and the type of guidance provided. Figure 8-1 shows the comparison. It has been extended to include Kanban, TDD, ATDD and specification by example methods.

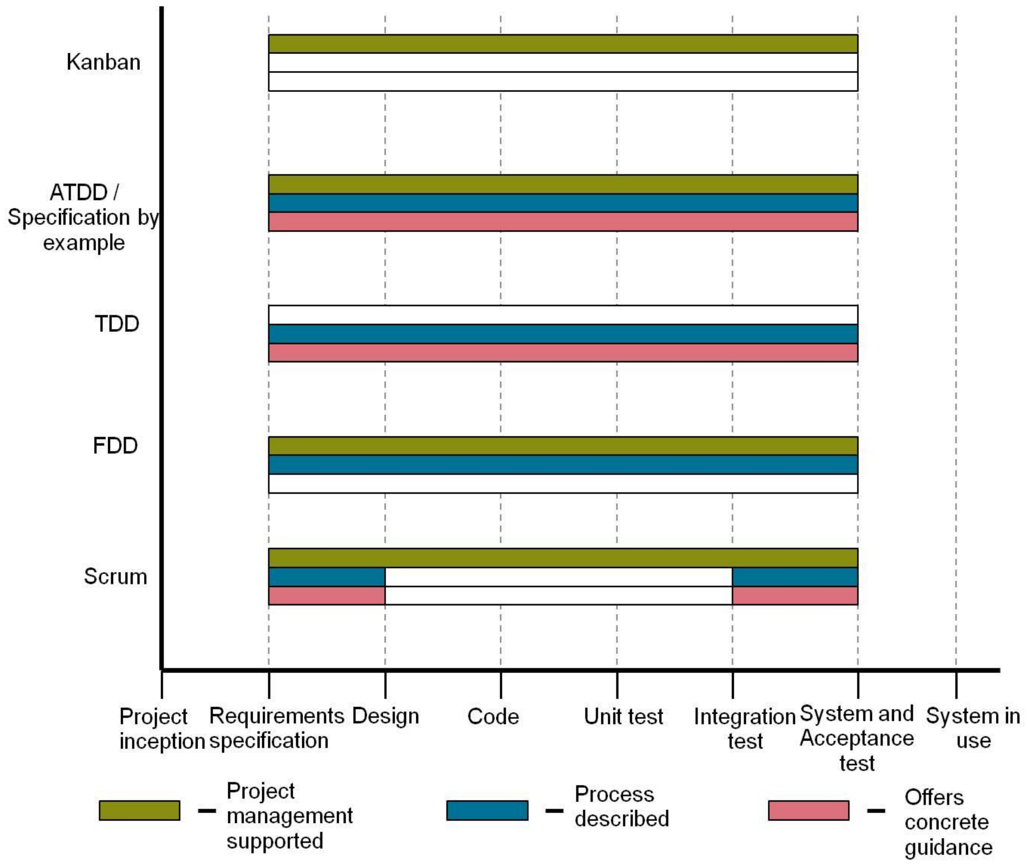


Figure 8-1. Supported Software Development Phases by Agile Methods
(Source: Adapted from Abrahamsson et al., 2003)

The various parameters based upon which comparison has been made by Abrahamsson et al. (2003) are project management support, software lifecycle coverage and type of guidance. In Figure 8-1, the green colored upper bar indicates if agile method provides support for project management. The blue colored middle bar indicates which phases of software development lifecycle are supported by agile method in question. The red colored lowermost bar indicates if the agile method provides abstract or concrete guidance for its use. White color of bar indicates missing support from agile method for that particular phase of software development (Abrahamsson et al., 2003)

As shown in Figure 8-1, Scrum, FDD and Kanban provide project management support. These methods provide guidelines on managing agile projects. ATDD and FDD cover all phases of software development lifecycle. TDD, being a XP practice provides concrete guidance since it is based upon concrete practices. (Abrahamsson et al., 2003)

3) **Heavy weight vs. Light weight methods**: Agile methods and consequently extended models emerging out of them can be categorized into heavy weight and light weight methods. Heavy weight methods promote upfront planning, documentation, specification of roles and activities. The main advantages perceived are lower overall cost, timely product delivery and better product quality (Germain & Robillard, 2005). Traditional software development methods like V-model, waterfall model fall into this category. Light weight methods on the other hand do away with much of the administrative overhead and documentation (Riehle, 2000). Agile methods fall into this category but some agile methods are more light weight as compared to others. The more an agile method is heavy weight or the lesser it is light weight; the easier it is to adopt it inside V-model which is heavy weight process. Based upon the definitions of heavy weight and light weight methods, agile methods can be differentiated based upon below factors:

- Degree of Agility
- Level of planning and documentation

Degree of Agility: Succi, D. Wells, Williams & J. Wells (2003) define agility as ‘the ability to both create and respond to change in order to profit in a turbulent business environment’. Qumer & Henderson (2008) have defined agility as “Agility is a persistent behavior or ability of a sensitive entity that exhibits flexibility to accommodate expected or unexpected changes rapidly, follows the shortest time span, uses economical, simple and quality instruments in a dynamic environment and applies updated prior knowledge and experience to learn from the internal and external environment". Agile methods differ in their degree of agility which can be measured on the basis of some factors.

Qumer & Henderson (2008) have measured degree of agility of six agile methods using an analytical tool called 4-DAT which is based upon below four dimensions:

1) Method scope (in terms of project size, team size, code style etc.)
2) Agility characterization (measured on the basis of flexibility, speed, leanness, learning and responsiveness)
3) Characterization of agile values (based upon values mentioned in Agile Manifesto)
4) Software process characterization (measured on the basis of product engineering and process management processes)

The four dimensions of 4-DAT tool are shown in detail in Chapter 1Appendix D Except second dimension; the other three dimensions provide qualitative analysis. The results of research conducted by Qumer & Henderson (2008) can be shown in Figure 8-2.

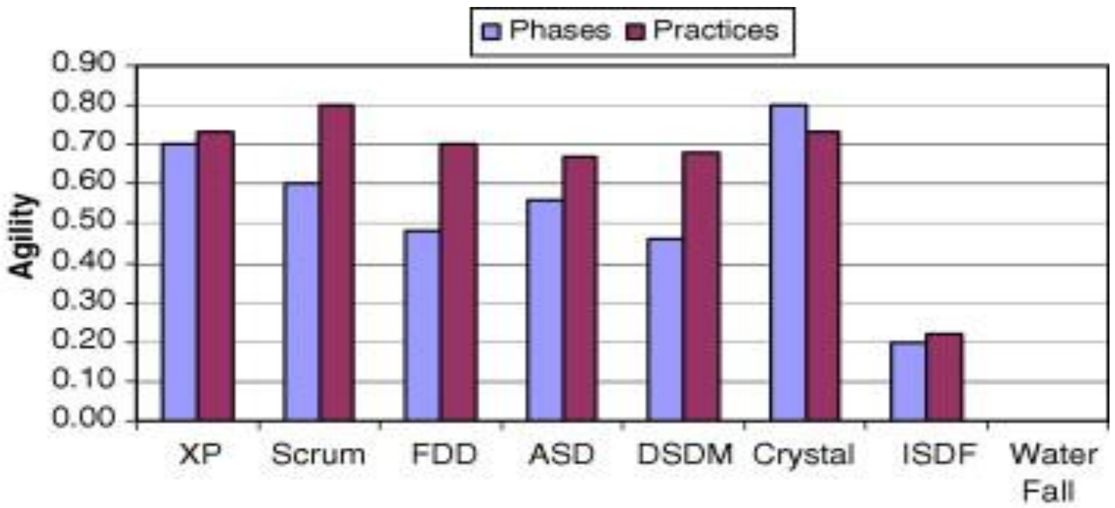


Figure 8-2. Comparison of Degree of Agility Derived from 4-DAT Tool (Source: Qumer & Henderson, 2008)

As can be seen in Figure 8-2, among XP (TDD), Scrum and FDD, Scrum is most agile in terms of practices followed by XP and FDD. XP is most agile in terms of phases followed by Scrum and FDD. FDD is least agile amongst three.

Level of planning and documentation: Agile methods employ different planning methods and processes. They also differ in the nature and number of artifacts. TDD, being a XP practice utilizes planning game, small releases, test suites etc. Scrum utilizes daily stand-ups, sprint review and retrospective meetings, user stories and burn-down charts for planning. Kanban is based upon use of Kanban boards, story cards etc. ATDD and specification by example is based upon executable specifications and living documentation. FDD is based upon high upfront planning and designing activities in terms of features, feature sets, domain models, class diagrams and sequence diagrams. In terms of level of planning and documentation, it can be concluded that FDD utilizes high planning and documentation activities followed by ATDD, Scrum, TDD and Kanban.

Based upon above discussion, agile methods can be presented on a scale of heavy weight and light weight methods as shown in Figure 8-3.

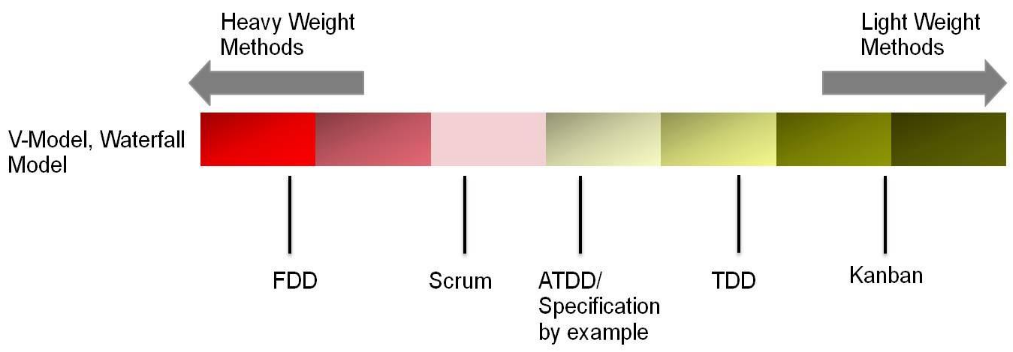


Figure 8-3. Representation of Heavy Weight and Light Weight Methods on a Scale

4) **Ease of implementation**: Agile testing method should be easier to implement. This depends upon the complexity of method and its dependency upon employees' skills. The more a method is dependent upon skills of team, the more it is difficult to implement. Kanban method is the simplest to implement, followed by Scrum and FDD.

5) **Scalability**: All agile testing methods are scalable. However, some agile testing methods are more scalable and applicable in larger projects in comparison to other agile testing methods.
   Scrum provides a way to introduce agility in plan-driven methods. It is also scalable. FDD is also scalable and can be implemented well in larger projects because of its high upfront planning. ATDD/Specification by example can also be scaled. XP is difficult to scale and is mostly applicable in smaller to medium projects, however individual XP practices can be adopted in any project. (Boehm & Turner, 2003)

8.2 Extended V-models' Comparison

In the above section, comparison was made between various agile testing methods. Based upon this comparison, extended V-models discussed in Chapter 7-Extended V-Models can now be compared to each other. Table 8-2 shows the comparison between extended V-models. If a model meets the criteria and factor of comparison to maximum extent as compared to other models, it is indicated by '+++' sign, and if it meets the criteria sufficiently, it is indicated by '++' sign. Complete evaluation mechanism is shown in Table 8-1.

Symbol	Meaning
+++	Meets the criteria to maximum extent
++	Sufficiently meets the criteria
+	Meets the criteria to some extent
0	Doesn't meet the criteria / is neutral

Table 8-1. Evaluation Mechanism

Extended V-Model					
Factor for comparison	**V-SCRUM-V**	**TDD-V**	**ATDD-V /Specification by example**	**FDD-V**	**Kanban-V**
QA activities	++	+	++	+++	+
Degree of Agility (from Figure 8-2)	+	++	+	0	+++
Level of planning and documentation	++	+	++	+++	+
Continuous Improvement	++	++	++	++	++
Customer interaction	++	+	++	+	0
Software Lifecycle support	+	+	++	++	+
Ease of implementation	++	+	+	++	+++
Scalability	++	+	++	++	++
Existing Roles' mapping	++	+	+	++	++
Project Management support	++	+	+	++	++

Table 8-2. Comparison of Extended V-Models

QA activities: FDD-V and V-SCRUM-V models employ considerable number of QA activities. ATDD-V by means of acceptance tests also impact quality of product, followed by TDD-V and Kanban-V models.

Degree of agility: The results of this factor's comparison are taken directly from comparison of degree of agility of agile testing methods with FDD-V being the least agile among all other extended models.

Level of planning and documentation: FDD-V model again employs highest level of planning and documentation, followed by V-Scrum-V and ATDD-V models.

Continuous improvement: All agile methods conduct retrospective meetings to continuously improve the process.

Customer interaction: Customer involvement and interaction is highest in case of V-SCRUM-V and ATDD-V models with customer taking part in regular meetings and representing requirements in the form of acceptance tests.

Software Lifecycle support: Although all models have been extended to support all phases of software development lifecycle, but some models provide specific guidelines for some phases based upon their parent agile testing method. Among all the models, FDD-V supports all phases of software development lifecycle.

Ease of implementation: Kanban-V extended model is the simplest and easiest to adopt in an organization. V-SCRUM-V and FDD-V extended models are also simpler as compared to TDD-V and ATDD-V.

Scalability: FDD-V and V-SCRUM-V models can be scaled to larger projects. Kanban-V and ATDD-V extended models are also more scalable as compared to TDD-V model.

Existing Roles' mapping: As shown in Table 7-1 and Table 7-2, existing V-model roles can be mapped to roles in FDD-V and V-SCRUM-V models. These extended models do not need any specialized roles for their implementation. With little training, existing roles can play new roles and support implementation of these models.

Project Management support: FDD-V and V-SCRUM-V models provide sufficient project management support.

By taking into account results of all factors of comparison shown in Table 8-2, extended V-models can be represented on a scale in terms of best fit and least fit candidate for adoption inside V-model. Extended V-model in which roles can be mapped to existing roles of V-model will pose lesser challenges for its adoption on organizational level. Similarly, extended V-model which supports all phases of software lifecycle and is scalable and provides risk coverage is better candidate for blend inside V-model. Figure 8-4 shows the comparison results.

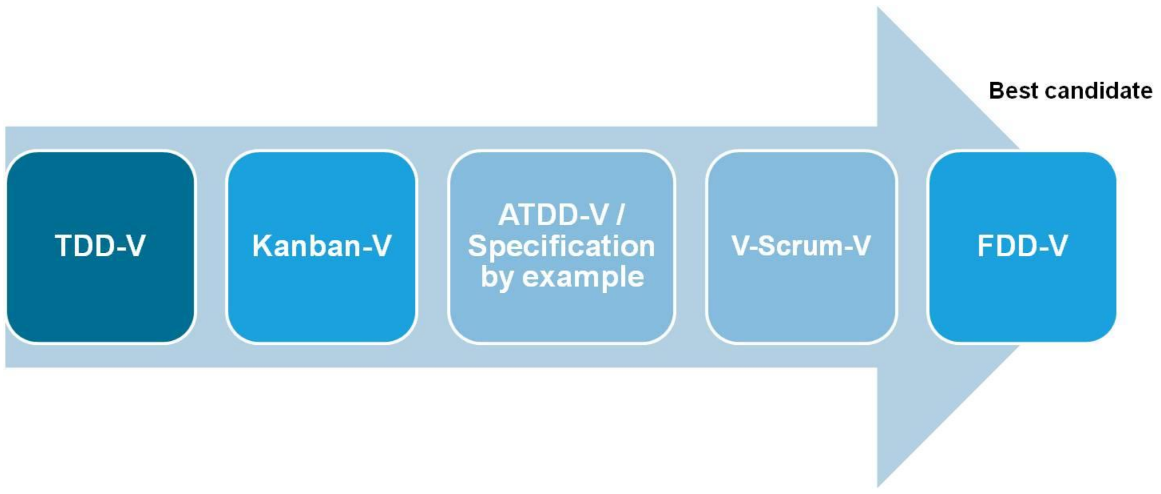


Figure 8-4. Representation of Extended V-Models on a Scale

8.3 Conclusions

In above chapters, comparison was made between extended V-models and their agile testing methods. Based upon the analysis, below conclusions can be drawn:

FDD-V extended model has the maximum associated quality assurance activities. FDD also provides project management support and supports all phases of software development lifecycle. It is highly scalable as compared to other agile methods.
FDD also seems to be best fit for V-model in terms of degree of agility and level of planning and documentation used. Hence FDD can be considered best agile method for adoption inside V-model.

Scrum and Kanban methods do not provide specific technical guidelines. These methods provide guidelines on overall management aspects of projects. These methods can be used as project management frameworks for managing projects and within these methods other technical agile testing methods like TDD or ATDD can be incorporated in order to have a complete software development lifecycle model.

The results from above analysis will be compared with case studies’ analysis.

9 Case Study Research

Case study research, like experiments, surveys, histories and archival analyses, is one of the social science research methods. Yin (2013) has given two fold definition of case study. The first part deals with the scope of case study and defines case study as an empirical inquiry that investigates a contemporary phenomena (the ‘case’) within its real-world context when the boundaries between phenomenon and context may not be clearly evident. The second part of definition deals with the features of a case study and defines case study as relying on multiple sources of evidence with data triangulation and having more variables of interest than data points. (Yin, 2013)

Yin (2013) mentions below three conditions for use of case study as a research method:

1) The main research questions are ‘how’ or ‘why’ questions
2) Researcher has little or no control over behavioral events
3) The study focuses on contemporary events (as opposed to entirely historical)

Table 9-1 shows the situations where different research methods are applicable.

Method	**Form of Research Question**	**Requires Control of Behavioral Events?**	**Focuses on contemporary Events?**
Experiment	How, why?	Yes	Yes
Survey	Who, what, where, how many, how much?	No	Yes
Archival Analysis	Who, what, where, how many, how much?	No	Yes/no
History	How, why?	No	No
Case Study	**How, why?**	**No**	**Yes**

Table 9-1. Relevant Situations for Different Research Methods
(Source: Yin, 2013)

The research question of this thesis is:

'How can agile testing methods be applied in projects managed by plan-driven methods?'

This research question (a 'How' question) and this research satisfies Yin's criteria for using case study as a research method.

The case study research method utilized in this thesis follows Yin (2013) suggestions for conducting a case study research.

Study Propositions: The propositions are mentioned in chapter 5-Hypothesis Formulation. These propositions or hypotheses have guided case studies' design, their scope and also aided in analytic generalization as part of case studies' analysis.

Case Study Design: The conducted case study research follows multiple case study design with use of three case studies based upon replication logic. The unit of analysis is the individual projects using one or more agile testing methods in traditional V-model.

Description of Case Studies: Three case studies have been utilized as part of research. All three case studies have replication logic and are based upon similar agile methodologies which made it easier to do comparative analysis. All three case studies describe software projects in public domain in Germany with agile testing methods embedded inside V-model. Table 9-2 gives brief overview of case studies utilized.

Case Study Name	Description
Case Study 1 – TINS (This is not Scrum)	Describes implementation of agile hybrid approach called TINS inside V-model in a project in public domain
Case Study 2 –Large Scale Scrum	Describes implementation of Scrum on larger scale. Project described in case study 1 was restructured and complete Scrum methodology was implemented in it.
Case Study 3 – Small Scale Scrum	Describes implementation of Scrum in contract and task manager system in context of small size project

Table 9-2. Description of Case Studies

Data Collection: The main sources of data were the interviews conducted with Capgemini, Germany employees and the data collected from the individual projects by direct observations. The interview partners varied from product owners, project managers to developers and architects. For the first case study, apart from the interviews with employees, another major source was thesis written by a Capgemini employee, Simon Boldinger, which described in detail the implementation of agile hybrid approach. Interview questions can be found in Chapter 1Appendix E The same set of interview questions were asked to all the interviewees of three projects.

Case studies' analysis: The data collected was analyzed by means of theoretical propositions and by working with the data from the ground up. Cross case synthesis technique was then employed.

Cross case synthesis technique is used to analyze two or more case studies. It can be applied to individual case studies which have been conducted as independent research studies or to case studies which are part of the same research. In cross case synthesis technique, case studies are analyzed by creation of word tables which enable drawing of cross case conclusions. In this thesis, three case studies were compared to each other based upon some uniform categories and results are presented in word table. (Yin, 2013)

10 Case Study 1 – TINS (This is not Scrum)

This case study describes agile approaches and testing methods embedded inside traditional V-model in a software project in Germany. The case study starts with the history of project and project description. It then states the reasons for introduction of agile approaches and describes in detail the agile software development approach used inside V-model and the various agile testing methods and quality assurance activities utilized in this project. The case is then analyzed for the pros and cons of using such a hybrid approach.

10.1 Project Description

This case study deals with a project for a client in public sector in Germany. The developed application calculated and accounted social benefits for unemployed people. It involved transfer of billions of Euros each year. The project involved complex requirements and could lead to people not receiving proper social benefits and unable to make a living if not implemented properly. The project was initiated in 2008 with start of development in 2010.

The system had three tiers – client, server and database. The client server architecture was based upon JAVA/JEE stacks with backend utilizing Oracle database. Around 50000 users worked on the system in parallel and around 3 million customers utilized the application. The future plan was to expand the number of customers to 10 million.

10.1.1 Organizational Structure

About 180 employees worked at peak time in the project which was divided into three subprojects - conception, design and implementation and testing. Rest of the people worked in cross functional areas like project management office.

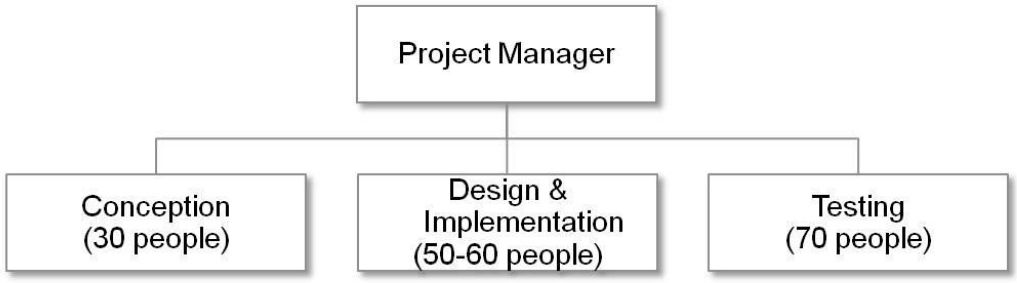


Figure 10-1. Organizational Structure – Overall Project
(Source: Boldinger, 2013)

The Design & Implementation sub-project was led by sub-project manager and consisted of Delivery Manager, Technical Chief Designer and Development Team. The development team comprised of seven teams. Four teams developed the server part, one team was involved in development of client application and two teams were support teams. Each team consisted of 6 people, with about 60 people working in total in design and implementation sub-project during peak time. (Boldinger, 2013)

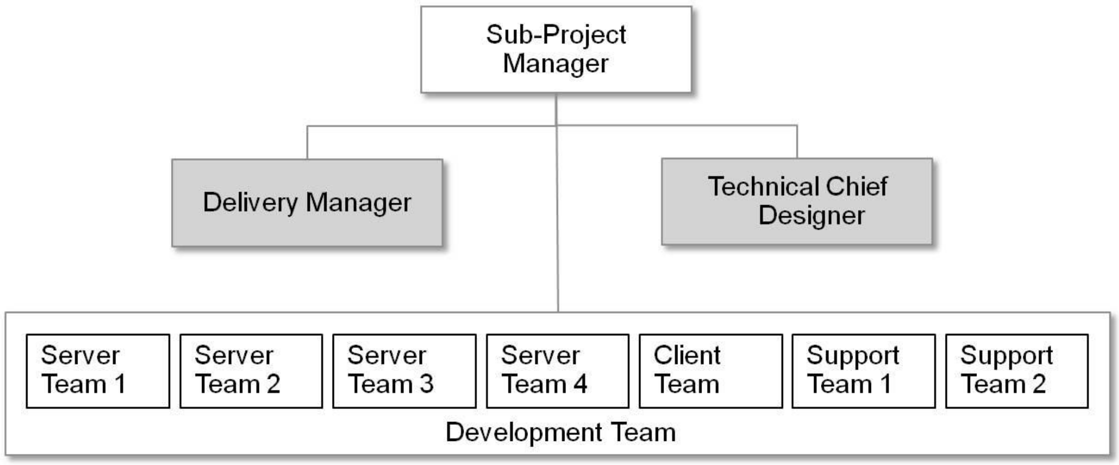


Figure 10-2. Organizational Structure - Design & Implementation
(Source: Boldinger, 2013)

10.1.2 Process Description

Before introduction of agile approaches, the whole project was structured around V-model and tailored according to waterfall model. Upfront requirements were gathered at the beginning of the project. The whole project was divided into three releases and product was developed in releases. Conception phase took about 6 months, design and implementation phase about 6 to 9 months and testing phase lasted for about 3 months in a release. Requirements specification started for the next release while the product was being developed in a previous release. Figure 10-3 shows the product development in releases.

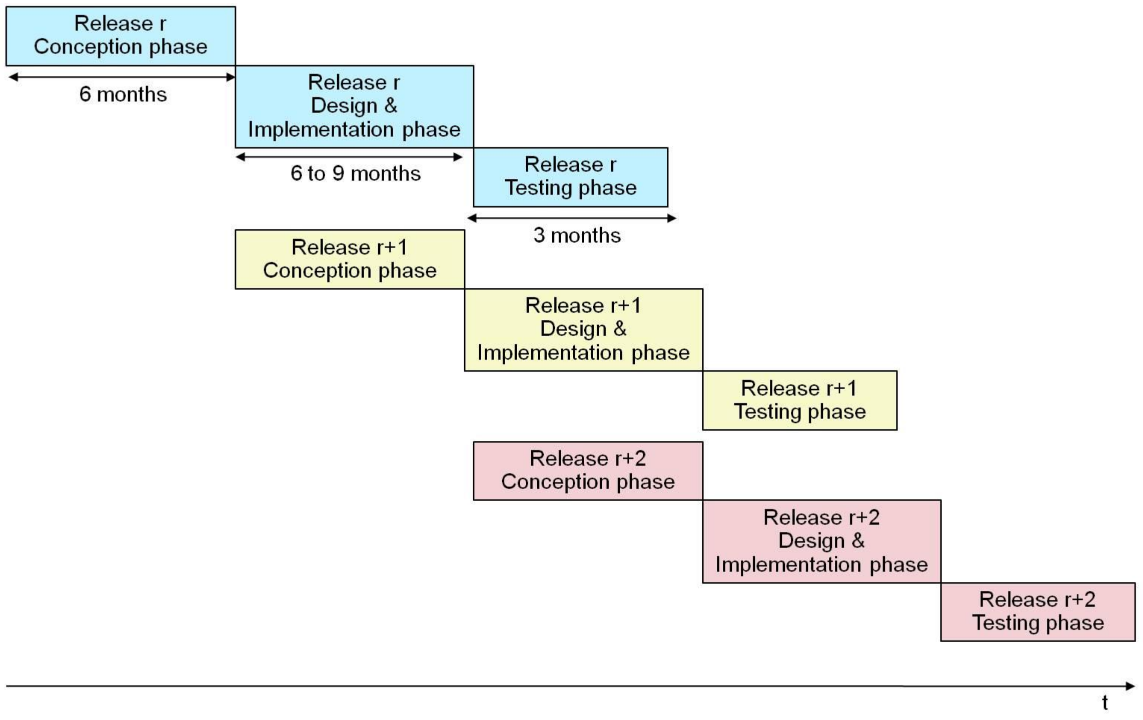

Release r Conception phase
6 months
Release r Design & Implementation phase
6 to 9 months
Release r Testing phase
3 months
Release r+1 Conception phase
Release r+1 Design & Implementation phase
Release r+1 Testing phase
Release r+2 Conception phase
Release r+2 Design & Implementation phase
Release r+2 Testing phase
t

Figure 10-3. Implementation of Software in Releases
(Source: Adapted from Boldinger, 2013)

The process described above suffered from below drawbacks (Boldinger, 2013):

- High upfront requirements gathering led to lot of effort being spent on planning activities. Any unplanned requirements or any changes to requirements were difficult to accommodate. Lot of unnecessary effort was being spent to accommodate such changes.
- Testing didn’t start until the design and implementation phase was completed. This led to lot of major bugs being discovered late in the lifecycle.
- Testing was also squeezed to meet the delivery deadline.
- The process also suffered from late design breakage or late integration.
- Long development cycle led to late feedback from the customer.

Owing to the above problems, timely delivery of high quality product was difficult. There was no regular customer feedback and changing customer requirements were also difficult to accommodate.

10.2 Introduction of Agile Approaches - TINS

The problems described above led to the introduction of agile approaches. The main goals were to reduce the planning effort, ability to accommodate changing requirements, earlier customer feedback, avoid big-bang integration at the end and detection of bugs earlier in the software development lifecycle.

Projects in public sector in Germany were forced to follow traditional V-model because of government regulations. Also, it was not possible to immediately switch to complete agile methodologies because of organization culture and larger project setting. Because of these reasons, agile development approach was introduced inside the design and implementation sub-project without impacting other phases. Other phases continued to follow V-model. In this section, this hybrid approach known as TINS (This is not Scrum) is being described. The way it was implemented inside traditional setting, its roles and responsibilities are also discussed.

10.2.1 TINS (This is not Scrum)

TINS (This is not Scrum) was the hybrid - model in model approach followed inside design and implementation phase. This approach was named TINS since it had lot of similarities with Scrum but was not completely identical to Scrum, hence to avoid any misconceptions, it was named as such. Figure 10-4 shows TINS inside traditional tailored waterfall model.

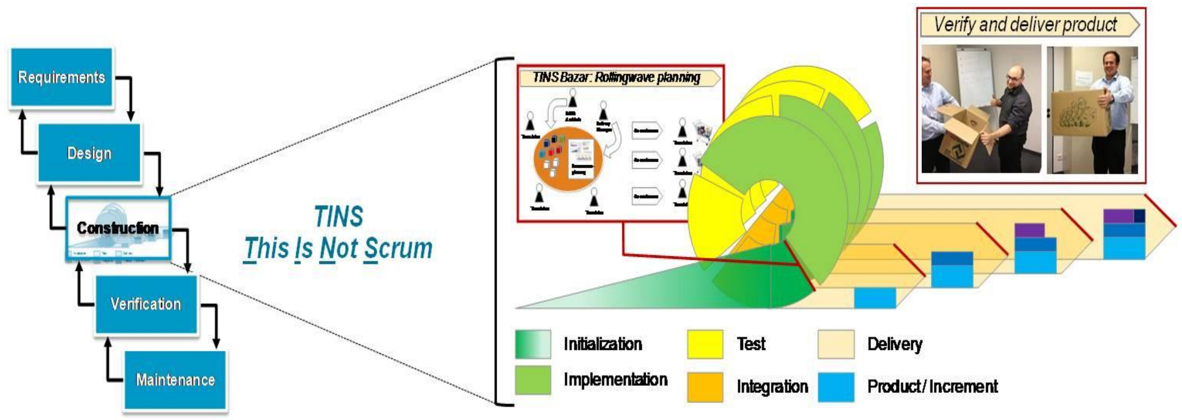


Figure 10-4. TINS inside Traditional Model

(Source: Capgemini, Germany, 2012)

In TINS, the 6 to 9 months period of development was split into 1 month long iterations. These iterations were called Stages which are similar to Sprints in Scrum. In each iteration, potential deliverable increment was developed. Figure 10-5 shows various phases of TINS approach.

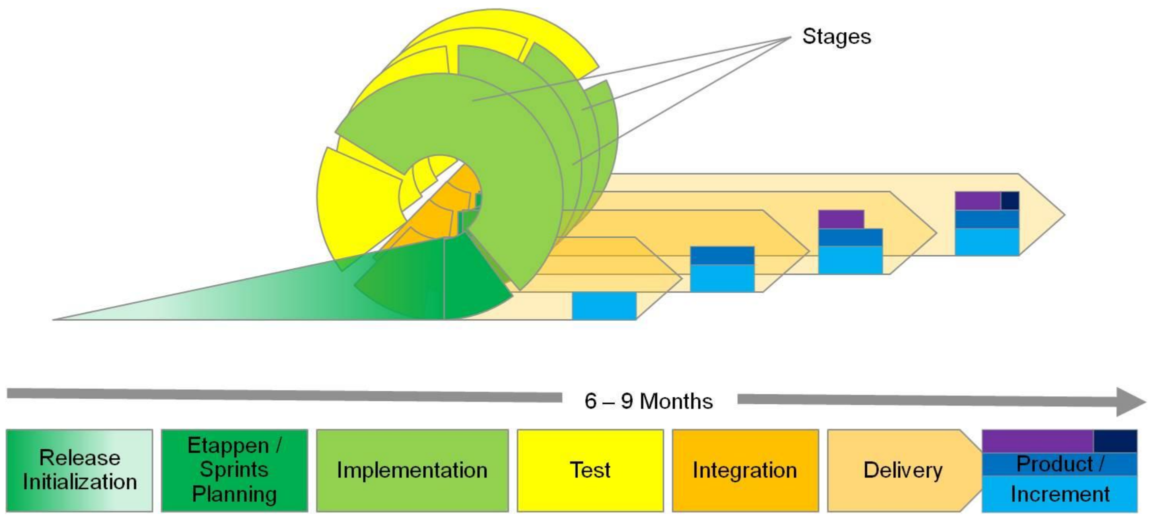


Figure 10-5. Various Phases of TINS Approach
(Source: Capgemini, Germany, 2012)

Release Initialization: In release initialization phase, overall planning for the release took place. Work packages were defined for the release and split into features. Any dependencies between work packages were also identified and a product backlog with prioritized features was prepared by TINS architect. TINS architect also defined which feature was supposed to be part of which iteration but this product backlog was updated before the start of each iteration. (Boldinger, 2013)

Stages / Sprints Planning: In this phase of TINS, planning for the iteration took place. A planning workshop called Basar was held in which the iteration was planned. Figure 10-6 shows TINS planning workshop – Basar.

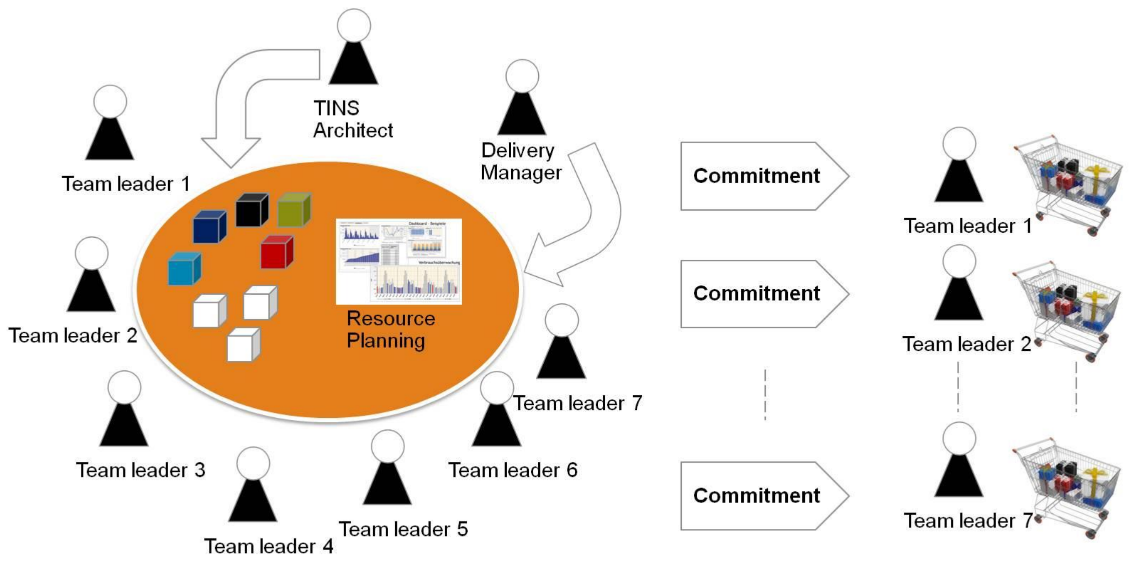


Figure 10-6. Illustration of TINS Planning Workshop (Basar)
(Source: Capgemini, 2012)

The planning workshop was organized by Delivery Manager and attended by all seven team leaders.

Retrospective meeting: In the first part of workshop, retrospective analysis of last iteration was done. Feedback was collected from all the team members. The goal of this meeting was to identify shortcomings of last iteration and to bring about improvements in next iteration. The development teams also gave overview of their achievements in last iteration including the features which could not be implemented and could be planned for next iteration. (Boldinger, 2013)

Next iteration planning: During the second part of planning workshop, measurable goals were set for the next sprint. Stakeholders were responsible for setting up these goals.

In the next part of planning, TINS architect presented the prioritized product backlog. Team leaders of all seven teams presented the resource capacities of their respective teams and committed to features which their teams would implement during the next sprint (Boldinger, 2013).

Implementation: During the implementation phase, product was developed in iterations known as Stages / Sprints. Each team member worked on its committed feature. At the end of iteration, all team members demonstrated developed increment to TINS architect and he had authority to accept or reject implemented features. Features which were not implemented were also identified and planned for next iteration based upon their priority.

10.2.1.1 TINS Roles

The various roles involved in hybrid approach described above were:

Sub-Project Manager: Sub-project manager was responsible for the overall project in design and implementation phase. He took care of project planning activities and guidance of individual team members.

Delivery Manager: Delivery manager was responsible for timely delivery of product. He ensured that high quality product was delivered on time and within budget. He was responsible for organizing planning workshops and removed any impediments faced by the team. He was also responsible for ensuring that the entire team followed TINS process properly.

TINS Architect: TINS architect was responsible for prioritization of features and presenting them in a product backlog. At the end of each iteration, he rejected or accepted implemented features based upon users' requirements.

Technical Chief Designer: Technical chief designer was responsible for design and architecture of application.

Team Leader: Team leader was responsible for his individual team. He represented his team members in planning workshop and presented resource capacities of his team. Based upon the feedback from team, team leader committed to features on behalf of the team. (Boldinger, 2013)

Development Team members: Individual team members of development teams were responsible for implementation of product in iterations. Each team member decided on its own which feature he wanted to work on in iteration.

10.2.2 Agile Testing Methods and Quality Assurance Activities

This section describes the agile testing methods and quality assurance activities employed in the project for achieving high quality software.

The testing activities in TINS followed Deployment pipeline (shown in Figure 10-7). The deployment pipeline shows the various tests conducted in the project which made sure high quality product was delivered to the customer.

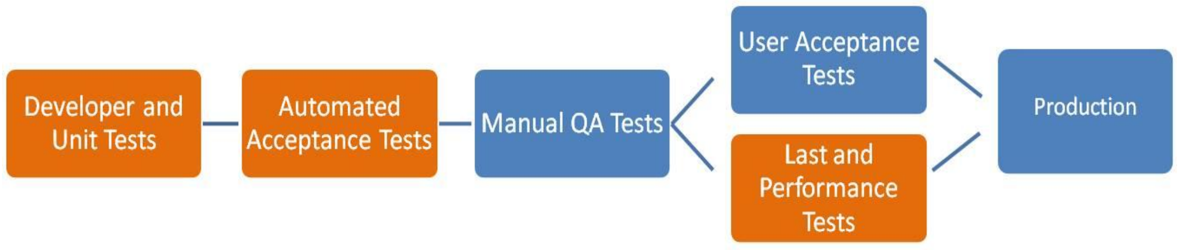


Figure 10-7. Deployment Pipeline
(Source: Anger & Eichler, 2013)

Developer and Unit Tests: Developer and unit tests were important part of implementation in TINS. For every feature, unit tests were mandatory. Unit tests were automated and whenever any feature was implemented, these automated tests were run. Any check-in made by the developer followed two-phase commit staging. In the first part of procedure, before a check-in could be made by the developer, his code was reviewed by his colleague. Only when it had passed code review, it cleared first staging area. During the second part of staging, implementation team followed build on commit which meant every time code was checked-in, automatic build was triggered and only when the build was successful, code was committed. This way it was ensured that continuous integration took place.

Automated Acceptance tests: Set of automated test cases was run every time build was triggered. The goal was to ensure that test cases were always green even during development (Anger & Eichler, 2013).

Daily Nightly Builds: At the end of the day, automated builds were started and set of automated test cases was run. Any failed test case was investigated next day and issues were resolved on time.

System and Integration testing: Developers also conducted system and integration testing in each iteration. This was part of doneness of feature. A feature was not considered done until it was tested in all respects.

Refactoring and Pair-programming: Developers followed pair-programming for implementing complex requirements. Refactoring was done continuously in order to maintain code quality.

User Acceptance tests: User acceptance testing was conducted by users while product was still implemented by the development team. This procedure was called ‘Quality Circle’.

Actual state of the software was provided to the users and few experts from the user side conducted free hands tests on the software. Experts tested software from user perspective and this provided users a chance to include improvements and other changes in software.

Quality Day: 'Quality day' was a special day in a release during which quality targets were set. For example one of the targets during one of the releases was to conduct static code analysis and improve quality of code by refactoring etc. Another target was to improve quality of tests, every developer had to either improve or fix existing tests or write new tests.

All Hands tests: Design and implementation team arranged 'All Hands tests' once in a release. All hands tests were conducted in two parts. During first half of the day, developers formed pairs and conducted pair-testing. Developers tested each other's code. One of the developers played the role of a navigator and navigated other through his code while the other developer conducted testing. During the second part of all hands tests, developers conducted freestyle testing and tested application in whatever way they wanted. At the end of the day, the winner of all hands tests, the one who found maximum number of defects or raised maximum number of bug tickets was declared.

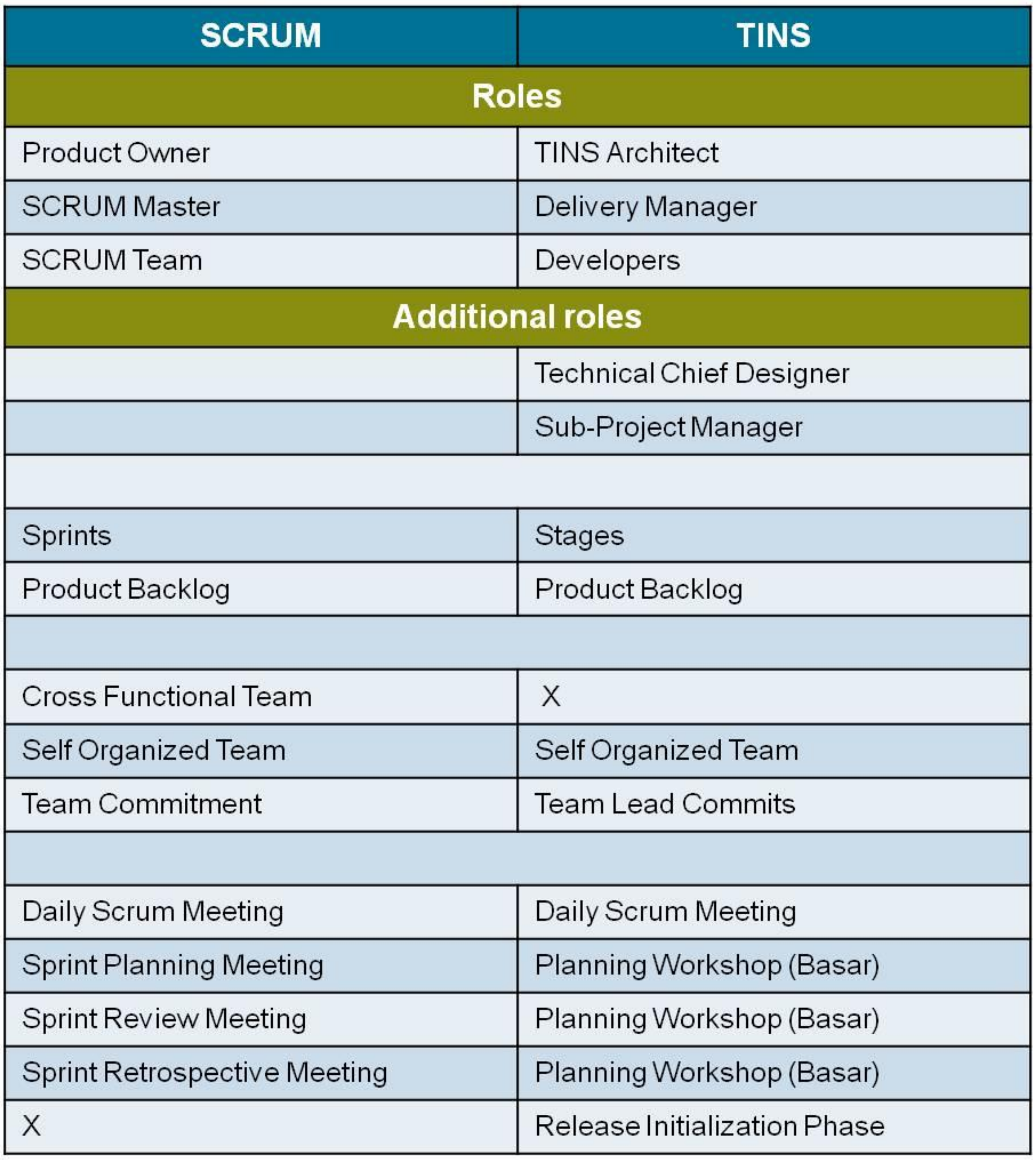
TDD: The development team sometimes used TDD. Implementation of TDD approach was difficult since the requirements were very complex in nature, hence developers found it hard to write tests before writing code. The plan was to increase TDD practices in future.

Specification by example: Another agile method implemented by development team was specification by example. It was followed in case of domain specific requirements. Without representation of such requirements in terms of examples, it was difficult to implement them.

Apart from all the tests and quality assurance activities employed in design and implementation team, testing team also tested software in traditional way. The testing team followed traditional V-model for testing and conducted system testing, integration testing, regression testing, performance testing etc.

10.3 Comparison of TINS with Scrum

Hybrid approach TINS can be compared to Scrum approach for software development. There are lots of similarities between the two approaches. TINS roles can also be directly mapped to the ones in Scrum. Table 5-1 shows comparison between TINS and Scrum.



SCRUM	TINS
Roles	
Product Owner	TINS Architect
SCRUM Master	Delivery Manager
SCRUM Team	Developers
Additional roles	
	Technical Chief Designer
	Sub-Project Manager
Sprints	Stages
Product Backlog	Product Backlog
Cross Functional Team	X
Self Organized Team	Self Organized Team
Team Commitment	Team Lead Commits
Daily Scrum Meeting	Daily Scrum Meeting
Sprint Planning Meeting	Planning Workshop (Basar)
Sprint Review Meeting	Planning Workshop (Basar)
Sprint Retrospective Meeting	Planning Workshop (Basar)
X	Release Initialization Phase

Table 10-1. TINS and Scrum comparison

Apart from roles of Scrum Master, Product owner and Scrum team, there are additional roles of Technical chief designer and Project manager in TINS. Like Sprints in Scrum, there are Stages in TINS. However in TINS there is no cross-functional team. There are only developers and testers are missing. Unlike Scrum, in TINS, team lead commits on the behalf of whole team. There are similar planning meetings in TINS as in Scrum but they are clubbed together in one planning workshop called Basar with one additional release planning phase called initialization phase.

10.4 Case Study Analysis

In this section, case study – TINS will be analyzed for its pros and cons. The problems faced while implementation of hybrid approach inside existing traditional model will be discussed.

10.4.1 Pros of TINS Hybrid Approach

This section describes the various pros and strengths of TINS approach.

1) **Early feedback**: The introduction of agile development approach inside traditional V-model provided earlier feedback to the team. The results from unit testing, continuous integration avoided Big-bang integration at the end of 6 months. Regular feedback was obtained by showing increment to the TINS architect during each iteration.
2) **Hybrid approach**: TINS made it possible to introduce agile development approach inside traditional V-model. This hybrid approach combined the best practices of two approaches - traditional and agile.
3) **Reduced development cycle**: Development cycle was reduced considerably. Earlier development cycle of 6 to 9 months was reduced to 1 month iterations. At the end of each iteration, potential deliverable increment was developed. This increment could be utilized by testing team for their tests.
4) **Transparent**: The hybrid approach increased transparency among team members. Planning workshops (Basar) and regular meetings avoided any confusion and increased trust level among team members. Other teams (like testing or specification team) could also utilize and benefit from the hybrid approach.
5) **Reduced risk**: Continuous delivery and feedback led to reduction of risks.
6) **Improved product quality**: Product quality was improved considerably through earlier testing and detection of bugs.

7) **Reduced planning**: The micro planning done before the introduction of TINS was reduced. High upfront planning was avoided and this led to reduction in planning costs and administrative overhead.
8) **Scalable**: TINS was proved to be scalable. It was implemented successfully in development team of 60 employees and could be scaled to higher level also.
9) **Ability to accommodate changing customer requirements**: Changing customer requirements could now be accommodated through iterative development.
10) **Continuous improvement**: TINS was continuously improved. Retrospective meetings provided a good way to collect team’s feedback and bring about improvements in next iteration.

10.4.2 Cons of TINS Approach

This section describes the cons and weaknesses of TINS approach. The difficulties faced during implementation of TINS are also discussed.

1) **Identification of critical path**: The use of agile approaches made it difficult to see the critical path. In traditional methods, it is easier to identify the critical path. In TINS, traditional methods were used for this purpose as and when required and TINS architect was responsible for it.
2) **Absence of complete agile methodology**: Absence of agile approaches inside the testing team and specification team made it difficult to implement complete agile methodology. If test stages could also be handled using an agile approach, then it would have been possible to gather earlier feedback by testing of increment after each iteration. This would have improved product quality to a greater extent and introduced more flexibility.
3) **Difficulty in implementing some agile approaches**: It was difficult to implement some agile approaches like TDD and ATDD. Requirements could have been provided in the form of examples or acceptance tests, had the specification team also followed agile methodology. This would have made it easier to follow TDD, ATDD or specification by example inside design and implementation team.
4) **Organization of Basar**: It was difficult to organize Basar in bigger team size by Delivery Manager. Commitment of features by team leaders on behalf of their team members also posed some planning difficulties.

10.5 Hypotheses Validation

In this section, hypotheses introduced earlier in *chapter 5-Hypothesis Formulation* will be validated against TINS case study. This validation is based upon responses received from interviewees and analysis of interviews.

Hypothesis 1: "***Agile testing methods when embedded inside V-model will lead to early and often testing and hence bugs being detected earlier in the software development life-cycle"***

This hypothesis holds true in the case of TINS. Unit tests and continuous integration led to bugs being detected earlier. Had the test team also followed agile methodology, it would have led to detection of bugs after each iteration.

Hypothesis 2: "*Agile testing methods when embedded inside V-model will lead to regular customer feedback and hence will be able to react quickly to changing customer requirements"*

This hypothesis is true in case of TINS. After each iteration, teams showed developed increment to TINS architect and experts from specification team, which provided immediate feedback and fixing of reported bugs. Any changes in customer requirements were accommodated in the next iteration.

Hypothesis 3: "*Agile testing methods when embedded inside V-model will lead to reduction in development cycle times*"

This hypothesis has already been stated as a pro of TINS approach. Development cycle times were reduced considerably from 6 to 9 months to 1 month iterations.

Hypothesis 4: *"Agile testing methods when embedded inside V-model build better quality product"*

This hypothesis has again been stated as strength of TINS. Product quality can be qualitatively measured in terms of customer satisfaction and reduced defect rate. Earlier testing and shorter feedback cycles improved product quality to a great extent.

11 Case Study 2 –Large Scale Scrum

This case study describes the implementation of Scrum and agile testing practices inside traditional V-model in a public project in Germany. It describes in brief the successful implementation of Scrum on large scale. The project described in Case Study 1 – TINS (This is not Scrum) was restructured to implement complete Scrum methodology. The project history, description, organizational structure are all same as in Case Study 1 – TINS (This is not Scrum), hence those parts are not described again. The main section highlighted in this case study is differences in the implementation of Scrum and TINS for the same project.

11.1 Hybrid Approach

TINS made it possible to implement agile methodologies in a complete traditional plan-driven environment. TINS, with its strategies and roles in between traditional V-model and Scrum, provided a pavement for future implementation of complete agile approaches. After the successful implementation of TINS, Scrum was implemented in the same project. Like TINS, it was introduced inside design and implementation phase. Other phases continued to follow traditional approaches. Figure 11-1 shows Scrum embedded inside traditional waterfall model tailored according to V-model.

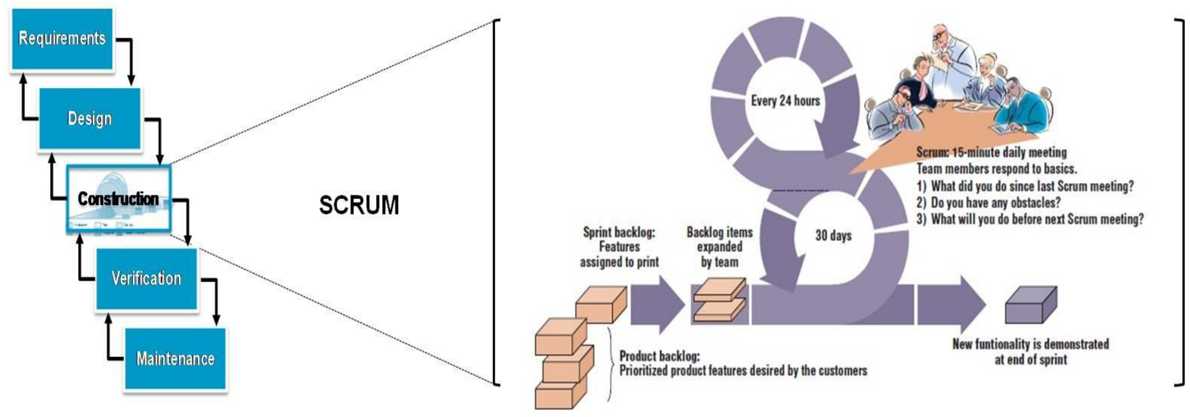

Requirements
Design
Construction
Verification
Maintenance
SCRUM
Every 24 hours
30 days
Scrum: 15-minute daily meeting
Team members respond to basics.
1) What did you do since last Scrum meeting?
2) Do you have any obstacles?
3) What will you do before next Scrum meeting?
Sprint backlog: Features assigned to print
Backlog items expanded by team
Product backlog: Prioritized product features desired by the customers
New funtionality is demonstrated at end of sprint

Figure 11-1. Scrum inside Design and Implementation Phase

(Source: Adapted from Capgemini, Germany, 2012)

The software development process started with requirements gathering and analysis like in traditional V-model. The requirements were very complex in nature. The specification phase created business logic and also defined interfaces to other systems. In the design and implementation phase, business logic was converted into technical design and development of application took place. After the application was developed, it was tested by an independent test team. This team conducted system testing, integration testing, regression testing etc. It also executed business tests which tested software against user requirements and from user perspective. Other non-functional tests like load and performance tests, security tests were also the responsibility of test team.

In the design and implementation phase, software was developed by following Scrum methodology. The development period of 6 to 9 months was split into 1 week long iterations called Stages. An iteration took 5 work days and was conducted on Thursdays. It started on previous week's Thursday and was brought to closure on next week's Thursday. In TINS, the iteration length was 4 weeks. Figure 11-2 illustrates differences in development cycles in TINS and Scrum.

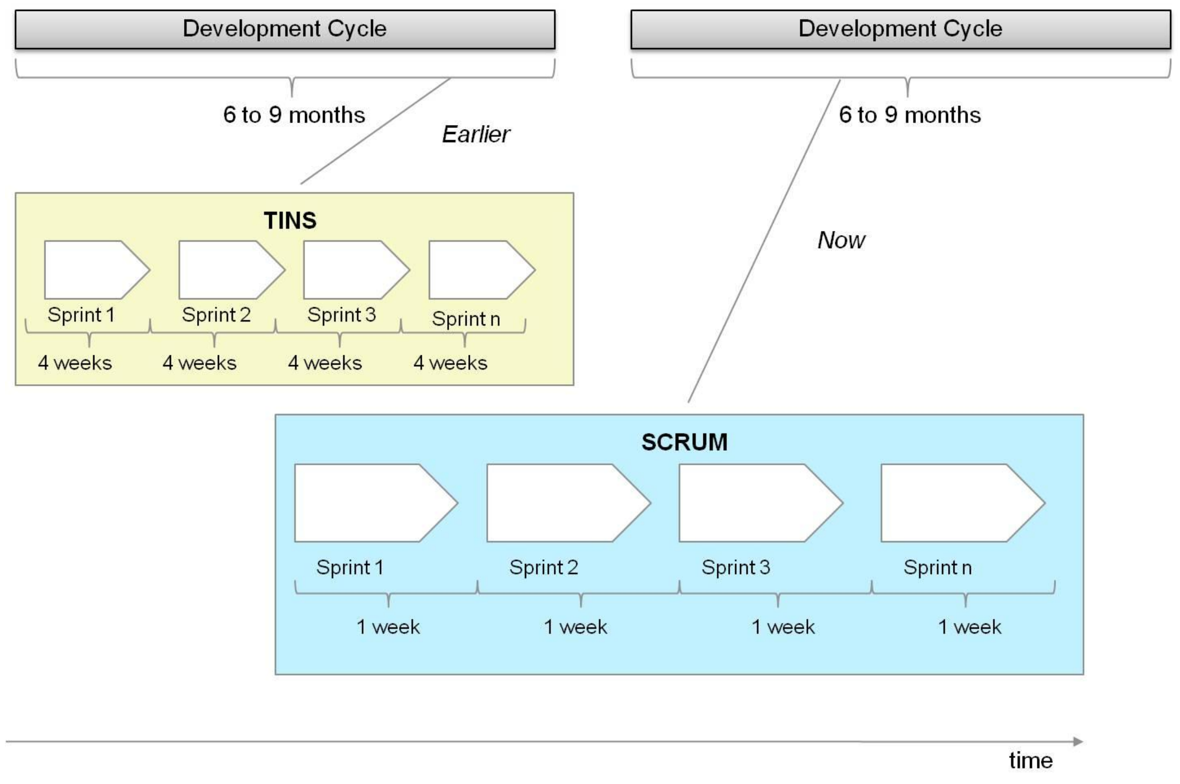


Figure 11-2. Scrum vs. TINS Development Cycles

11.2 Roles in Large Scale Scrum

The roles in large scale Scrum model were the combination from TINS and Scrum methodology. The roles included Product owners, Scrum master, development team along with delivery manager, project manager, sub-project manager etc.

Each development team had one product owner, so in total there were seven product owners headed by a super product owner. All product owners met before the sprint and discussed the distribution of features.

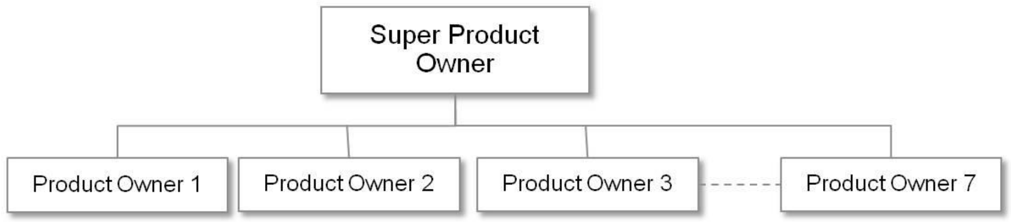


Figure 11-3. Product Owners in Large Scale Scrum model

11.3 Comparison of TINS and Large scale Scrum approaches

In this section, TINS approach discussed in Case Study 1 – TINS (This is not Scrum) will be compared to large scale Scrum approach.

1) **Iteration duration**: The two approaches differed in the duration of iterations. In large scale scrum approach, development took place in 1 week iterations as compared to 4-week iterations in TINS.
2) **Involvement and participation of test team**: Another difference lied in the increase in involvement and participation of test team in large scale scrum. Test team now took part in sprint planning meeting which enabled them to plan their test strategies in advance. Test team could also influence the product backlog. They could ask the product owner to increase the priority of a feature, if they thought something was of high priority from testing or customer point of view.
3) **Cross functional team**: In large scale scrum approach, cross functional team was achieved by involving two members from test team. These two members from test team took part in sprints and tested developed increments. This provided immediate feedback to the development team.

4) **Dedicated bug fixing team**: Large scale scrum model now had a dedicated bug fixing team. This team fixed bugs and tested fixes in each iteration. Buffer was also assigned to fix any high priority bug in between iterations.
5) **Regression test suite execution**: Development team in large scale scrum model executed regression test suite of testing team. This led to earlier detection of regression bugs. Early feedback was provided to test team in case some feature was updated and implemented in a different way and they could update their test scripts accordingly.

11.4 Case Study Analysis

In this section, large scale scrum model case study will be analyzed for its pros and cons.

11.4.1 Pros of Large Scale Scrum Approach

Apart from pros of TINS approach, large scale scrum approach offered additional benefits.

1) **Continuous improvement**: In large scale scrum model, like Scrum, sprints were planned in sprint planning meeting. Sprint review and retrospective meetings took place at the end of 1 week sprint which led to continuous improvement of process and approach. Involvement of test team in the meetings provided added benefit.
2) **Much earlier feedback**: In large scale scrum approach, feedback was provided much earlier through results of testing by test team members in sprints, shorter iterations and execution of regression test suite by development team.
3) **Scalable**: In large scale scrum model, Scrum approach was successfully implemented in project size of 180 employees with 60 employees in design and implementation team and it could be scaled even to larger size project.
4) **Flexibility**: Large scale scrum model was flexible enough to accommodate changing customer requirements. Owing to shorter iterations duration, any high priority feature could be accommodated in next iteration. In case of TINS, any high priority feature could be taken up only after 4 weeks.
5) **Better product quality**: Due to increased involvement of test team and earlier feedback, better product quality was achieved.

11.4.2 Cons of Large Scale Scrum Approach

Large scale Scrum approach suffered from same cons as TINS approach. However because of shorter iteration durations and involvement of test team in Scrum model, difficulties faced were reduced to some extent. More benefits and flexibility could be achieved if agile approaches were implemented completely inside test team and specification team.

11.5 Hypotheses Validation

The hypotheses introduced in *chapter 5-Hypothesis Formulation* could be validated against case study 2 – Large scale Scrum.

Hypothesis 1: "***Agile testing methods when embedded inside V-model will lead to early and often testing and hence bugs being detected earlier in the software development life-cycle"***

This hypothesis holds true in case of large scale scrum model. Apart from bugs detected by unit tests and continuous integration, additional bugs were detected by testing conducted by test team after each iteration. The bugs were discovered and fixed weekly after each sprint.

Hypothesis 2: "*Agile testing methods when embedded inside V-model will lead to regular customer feedback and hence will be able to react quickly to changing customer requirements"*

This hypothesis is true in case of large scale scrum approach. At the end of each sprint, developed product increment was shown to the customer and experts from specification team. Any changes requested by the customer could potentially be accommodated in next iteration depending upon their priority.

Hypothesis 3: "*Agile testing methods when embedded inside V-model will lead to reduction in development cycle times*"

This hypothesis again holds true in case of large scale scrum approach. Development cycle times were reduced considerably owing to development in 1 week iterations.

Hypothesis 4: ***"Agile testing methods when embedded inside V-model build better quality product"***

Better product quality was achieved in large scale scrum model because of involvement of test team in sprints and early feedback.

12 Case Study 3 – Small Scale Scrum

This case study describes the implementation of Scrum methodology in small scale project managed by traditional V-model. The project described here belonged to public sector in Germany. The case starts with project history and project description. It then describes the implementation of Scrum inside traditional V-model and later on analyzes the hybrid approach for its strengths and weaknesses.

12.1 Project Description

The project described in this case study belonged to public sector domain in Germany. It started as task manager application in 2010 and was later moved under Enterprise Service Bus (ESB) project. The system was based on oracle techniques and lots of performance issues were noticed during runtime. At the end of 2012, decision was made to switch to JAVA technology for implementation and the result was a new project which was a spin off from ESB project. This new project was called Contract and Task manager which is the focus of this case study.

The software had two components service and graphic user interface. Through graphic user interface client could retrieve all contracts and tasks, but it was not utilized at the moment and the focus was on service component. The system was based upon Service Oriented Architecture (SOA).

12.1.1 Organizational Structure

The total number of employees in project were 45 with 30 people working in development team and 3 people working in Project Management Office.

The development team was further divided into 3 sub-teams in order to achieve scalability while implementing agile methodologies.

12.2 Hybrid Approach

Government regulations forced the companies to follow traditional V-model for software development for public sector projects in Germany. In order to gain flexibility, fast delivery, high product quality, reduced costs etc., organizations strived to adopt agile methodologies. Hence a mid way solution was found and the result was a hybrid approach.

The project described above followed hybrid approach by blending Scrum methodology inside V-model for software development. Requirements were gathered at the beginning of release from the client. Product was then developed using Scrum methodology. The various phases followed inside one sprint were analysis, design, implementation and testing. A separate independent test team also conducted testing after the product was developed. Figure 12-1 shows this hybrid approach for product development.

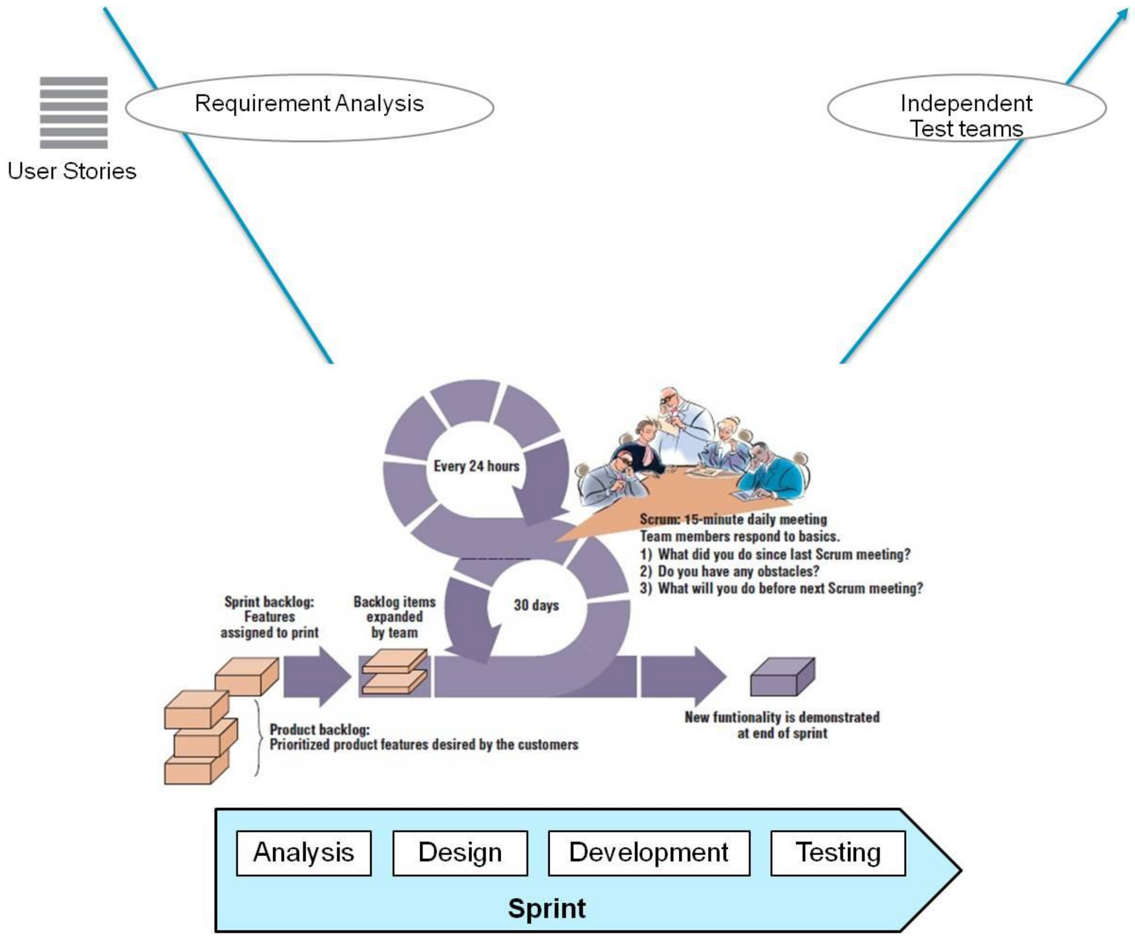

Requirement Analysis
User Stories
Independent Test teams
Every 24 hours
Scrum: 15-minute daily meeting
Team members respond to basics.
1) What did you do since last Scrum meeting?
2) Do you have any obstacles?
3) What will you do before next Scrum meeting?
30 days
Sprint backlog: Features assigned to print
Backlog items expanded by team
Product backlog: Prioritized product features desired by the customers
New funtionality is demonstrated at end of sprint
Analysis
Design
Development
Testing
Sprint

Figure 12-1. Hybrid Software Development Approach

Development cycle of 4 months was split into 4 weeks sprints.

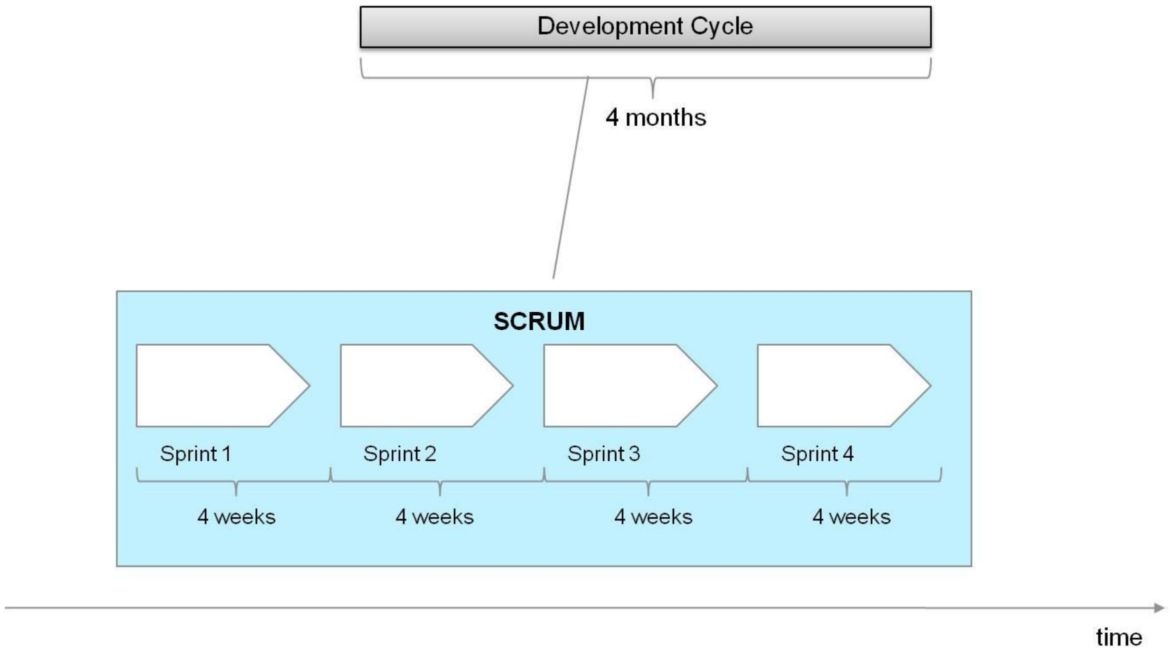


Figure 12-2. Development Cycle in Small Scale Scrum model

The various roles involved were Scrum master, product owner and development team. The development team was divided into 3 sub-teams and each sub-team followed Scrum approach and had its own product owner. The three product owners were headed by a chief product owner. The product owners had the responsibility to prioritize requirements and present them in the form of user stories in a product backlog. They also provided clarifications regarding requirements if needed. A lead analyst helped product owners in getting detailed information about requirements. A cross functional team consisting of developers, architects and testers was responsible for product development. At the end of each sprint, product increment was shown to the product owner and he had the authority to accept or reject developed user stories. Sprints were planned in sprint planning meetings. Sprint review and retrospective meetings helped in continuous improvement. Daily standup meetings were conducted in order to track progress and identify any impediments to development. Scrum master supported team in removing any blockages and helped them to follow Scrum process.

12.2.1 Agile Testing Methods and Quality Assurance Activities

This section describes the various quality assurance activities implemented inside the project.

Unit tests: Developers wrote unit tests for all the user stories they implemented. These unit tests were automated and executed during each sprint.

Continuous integration: Every time a developer checked-in his code, an automatic build would be triggered. Any integration problems would be detected and fixed immediately within the same sprint.

TDD: The development team implemented test first programming concepts of TDD. They wrote tests before writing code and followed *Red* -> *Green* -> *Refactor* cycle of TDD.

Regression tests: Within a sprint, regression tests were conducted by testers. In sprint review meeting, it was important to demonstrate that regression tests were Green. It was part of doneness of a user story. If any test was Red, that user story was rejected and not considered done.

User acceptance testing and free hands testing: As a part of sprint review meeting, product owners conducted user acceptance testing and free hands testing on the developed increment and tested product from user perspective.

Independent test teams: After the product was developed, independent testing was conducted by customer and an external vendor. A team at customer end conducted system acceptance testing and load and performance testing in their environment for about 6 weeks. Sometimes there were some specific user stories which required support from development team and in such cases; development team had to support customer test team in their tests. An external vendor conducted system integration and acceptance tests in a separate environment. The vendor didn’t follow Scrum process. Figure 12-3 shows various types of tests conducted within the project.

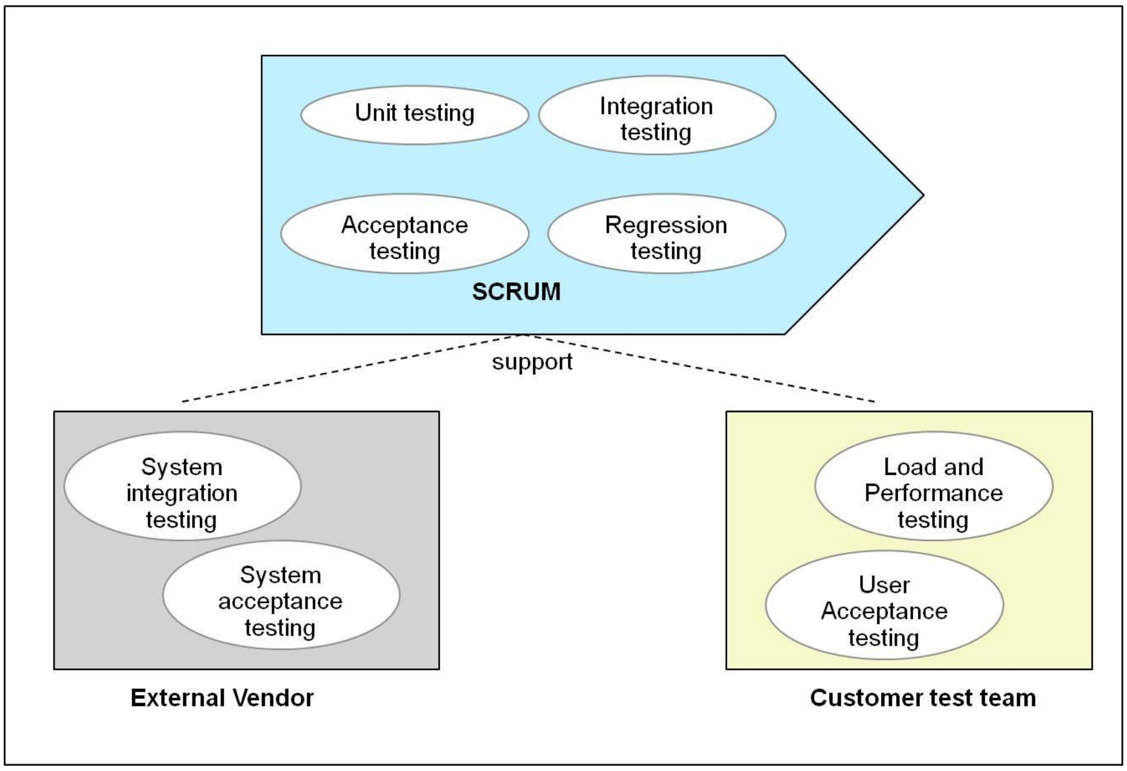


Figure 12-3. Testing in Small Scale Scrum Model

12.3 Case Study Analysis

In this section, the small scale scrum model will be analyzed for its strengths and weaknesses.

12.3.1 Pros of Small Scale Scrum Approach

The various pros of hybrid model were:

1) **Avoidance of 90%-complete-syndrome**: The hybrid small scale Scrum approach was an honest approach. It prevented the 90%-complete-syndrome explained by the ninety-ninety rule which states that 'The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time' (Bentley, 1985). A user story was either 0% done or 100% done, any other status was not accepted.
2) **Realistic estimations**: Since in hybrid approach, developers were responsible for providing estimations for story points, the estimations were realistic.
3) **Reduced development cycle time**: There was reduction in development cycle time. It was reduced from 4 months to 4 weeks.

4) **Improved testing**: Since testers were part of scrum team, hence product was tested in each sprint. Testing was part of doneness of user story. Product was also tested afterwards by independent test teams which greatly improved product quality.

12.3.2 Cons of Small Scale Scrum Approach

The various weaknesses of hybrid model were:

1) **Lack of participation of business department**: The major weakness of hybrid approach was the lack of participation of business department in the process. There was lack of business-IT alignment and business department didn't participate directly in the requirements gathering and analysis process. The requirements stated by the IT department sometimes conflicted with the business requirements. There was need for one common department representing requirements from both business as well as IT perspective.
2) **Support to independent test teams**: Sometimes the development team had to support customer test team in their testing. Some user stories specified this support requirement. The problem was that these user stories could not be estimated and hence could not be managed. Also, the customer test team worked independently from the development team which made the coordination difficult.

12.4 Hypotheses Validation

The hypotheses stated in *chapter 5-Hypothesis Formulation* can be validated against hybrid approach discussed in previous sections.

Hypothesis 1: "***Agile testing methods when embedded inside V-model will lead to early and often testing and hence bugs being detected earlier in the software development lifecycle"***

This hypothesis is true in case of small scale scrum approach. Owing to agile testing methods and testing conducted by scrum testers, bugs were detected earlier in the software development lifecycle.

Hypothesis 2: "*Agile testing methods when embedded inside V-model will lead to regular customer feedback and hence will be able to react quickly to changing customer requirements"*

Regular customer feedback was provided after each sprint. Lead analyst helped product owners in clarifying and updating requirements. Any changes in requirements were accommodated in next sprint.

Hypothesis 3: "*Agile testing methods when embedded inside V-model will lead to reduction in development cycle times*"

Development cycle time was reduced from 4 months to 4 weeks. At the end of 4 weeks, potential deliverable increment was developed.

Hypothesis 4: *"Agile testing methods when embedded inside V-model build better quality product"*

This hypothesis also holds true in case of case study described above. Product quality, measured in terms of reduced defect rate and increased customer satisfaction, was improved considerably because of use of agile methods (like TDD, continuous integration etc.) and scrum methodology inside V-model.

13 Case Studies' Comparison

In this chapter, the three case studies discussed in *Chapters 10-Case Study 1 – TINS (This is not Scrum), 11-Case Study 2 –Large Scale Scrum* and *12-Case Study 3 – Small Scale Scrum* have been analyzed and compared to each other. The cross case synthesis technique described by Yin, 2013 (See *chapter 9-Case Study Research* for its description) was utilized for case studies' comparison. Categories for comparison were identified and three case studies were analyzed based upon these categories.

The three case studies presented three different situations in which agile testing methods were embedded inside traditional V-model.

- **Case Study 1 – TINS (This is not Scrum)** discussed a situation in which agile methodology similar to Scrum was embedded inside traditional V-model. The project described was large scale project. It demonstrated a possible solution for large projects transitioning to agile methodologies from complete traditional approaches.
- **Case Study 2 –Large Scale Scrum** discussed a situation in which Scrum methodology was implemented inside V-model. It showed successful implementation of hybrid approach combining Scrum and traditional concepts in a large scale setting.
- **Case Study 3 – Small Scale Scrum** discussed a situation in which Scrum methodology was embedded inside V-model in a small scale project.

Despite of three different situations, the three case studies have common parameters based upon which comparison can be made.

Category	Case Study 1 – TINS	Case Study 2 –Large Scale Scrum	Case Study 3 – Small Scale Scrum
Software development model	Methodology similar to Scrum embedded inside V-model	Scrum embedded inside V-model	Scrum embedded inside V-model
Development period	6 to 9 months	6 to 9 months	4 months
Iteration duration	4 weeks	1 week	4 weeks
Agile quality assurance practices used	TDD, Specification by example, Developer and Unit tests, Automated Acceptance tests, Daily Nightly Builds, System and Integration testing, Refactoring and Pair-programming, User Acceptance tests, Quality Day, All Hands tests, Regression testing, Performance testing	TDD, Specification by example, Developer and Unit Tests, Automated Acceptance tests, Daily Nightly Builds, System and Integration testing, Refactoring and Pair-programming, User Acceptance tests, Quality Day, All Hands tests, Regression testing, Performance testing, Dedicated Bug fixing team, Involvement and Participation of Test team	TDD, Unit tests, Continuous integration, Regression testing, User Acceptance testing, Free hands testing, System integration testing, Load and Performance testing
Roles	Project Manager, Sub-Project Manager, Delivery Manager, TINS Architect, Technical Chief Designer, Team leaders, Development team members	Project Manager, Sub-Project Manager, Delivery Manager, Super Product owner, Product owners, Scrum master, Development team members, Team leaders	Chief Product owner, Product owners, Development team members, Lead analyst
Number of development teams	7	7	3
Development team Size	60 employees	60 employees	30
Scalable	Yes	Yes	Yes
Scrum testers	No	Yes	Yes
Commitment on user stories	Team lead committed	Team lead committed	Team committed
Cross functional team	No	Yes	Yes
Independent test team	Yes	Yes	Yes

Strengths	Early Feedback, Hybrid approach, Reduced development cycle, Transparent, Reduced Risk, Improved Product Quality, Reduced Planning, Scalable, Ability to accommodate changing customer requirements	Much earlier feedback, Continuous improvement, Scalable, Flexibility, Better product quality, Hybrid approach, Reduced development cycle, Transparent, Reduced Risk, Reduced Planning, Ability to accommodate changing customer requirements	Avoidance of 90%-complete-syndrome, Realistic estimations, Reduced development cycle time, Improved testing, Hybrid approach, Ability to accommodate changing customer requirements
Weaknesses	Identification of Critical Path, Absence of complete agile methodology, Difficulty in implementing some agile approaches	Identification of Critical Path, Difficulty in implementing some agile approaches	Lack of participation of business department, Support to independent test teams

Table 13-1. Case Studies' Comparison

Table 13-1 shows the comparison between three case studies based upon various categories.

In summary,

- Case study 1 and case study 2 differed in the way agile methodology was implemented inside V-model. In Case study 1, methodology similar to Scrum was implemented, whereas in case study 2, complete Scrum methodology was implemented. The two differed in the duration of iterations. In comparison to case study 1, testers were part of Scrum team in Case study 2 making it a cross functional team. Owing to shorter sprint durations, various additional benefits like much earlier feedback, better product quality and improved flexibility etc. were obtained in case study 2.
- Case study 3 differed from case study 1 and 2 in the way Scrum methodology was implemented inside V-model. Apart from initial requirements gathering; analysis, design, implementation and testing were all conducted inside Scrum. Development cycle duration (4 months) was also different from the release duration (6 to 9 months) of case study 1 and 2. The project size was smaller and number of development teams was also less. In contrast to team lead's commitment in case study 1 and 2, whole team committed in case study 3. An external vendor additionally tested application in case study 3.

14 Conclusions and Future Work

14.1 Summary and Conclusions

As Boehm & Turner (2003) have rightly said 'Neither agile nor plan-driven methods provide a methodological silver bullet'. In today's competitive world, organizations need to strike a balance between agility and stability. They need to be flexible and responsive to changing customer needs and deliver high quality product at the same time. Testing is an important aspect of software development. Proper testing increases confidence in product and leads to increased customer satisfaction by delivering high quality product.

This master thesis analyzed various agile testing methods and suggested ways they can be adopted inside V-model. Since there are number of agile methods, hence criterion for selection of agile methods was used. Quality assurance activities of agile methods was the foremost condition for their selection. Other factors for selection included degree of scalability, degree of agility, roles, smallest unit of work etc. Based upon these criteria, agile testing methods were selected and discussed in detail.

Based upon literature review, four hypotheses were formulated. These hypotheses are summarized below:

- **Hypothesis 1**: "Agile testing methods when embedded inside V-model will lead to early and often testing and hence bugs being detected earlier in the software development life-cycle"
- **Hypothesis 2**: "Agile testing methods when embedded inside V-model will lead to regular customer feedback and hence will be able to react quickly to changing customer requirements"
- **Hypothesis 3**: "Agile testing methods when embedded inside V-model will lead to reduction in development cycle times"
- **Hypothesis 4**: "Agile testing methods when embedded inside V-model build better quality product"

Agile and plan-driven methods were compared to each other in four areas – Application, Management, Technical and Personnel. Weaknesses were identified in both the approaches and the need for hybrid approaches combining traditional and agile methods was discussed. The challenges of this hybrid approach implementation were also discussed. The challenges of agile methods adoption occur at three levels – Management, People and Process & Environment. Out of these challenges, management and people challenges occurring at the culture and strategy level in an organization were found to be most difficult to tackle. Implementation of agile methods inside traditional hierarchical organization needs support from management and proper care needs to be taken during transition.

One of the most important contributions of this research was the presentation of five extended V-models with embedded agile testing methods. Care was taken that proposed models required minimum changes to existing processes and roles. The models suggested ways to embed various methods like Scrum, TDD, ATDD, specification by example, FDD, Kanban inside V-model. Use of an independent test team was suggested in all the models. This team tested product increment as soon it was developed and worked closely with the development team.

Extended V-models and their parent agile testing methods were compared to each other on various grounds. These included QA activities, supported phases of software development, degree of agility, level of planning and documentation, ease of implementation, scalability, customer interaction etc. It was concluded that FDD-V extended model employs considerable number of QA activities and supports all phases of SDLC. It is also scalable and has the lowest degree of agility and was found to be the best candidate for adoption inside V-model. Another conclusion made was that Scrum and Kanban methods don’t provide any specific technical guidelines and these methods can be used as frameworks within which other agile testing methods like TDD, ATDD etc can be adopted.

In order to validate hypotheses, case study research method was used. Interviews were conducted and three case studies i.e. three projects which implemented agile testing methods inside V-model were discussed. These projects belonged to public sector in Germany. The three case studies represented different situations in terms of project size, project history and implementation of hybrid approach. Cross case synthesis technique was then used to compare and analyze case studies. Case studies were compared to each other based upon some uniform categories.

Below is the summary of conclusions drawn upon comparison of extended V-models and case studies.

- Case studies demonstrated that agile testing methods can be successfully implemented inside V-model
- Case studies showed various benefits achieved by implementation of agile methods namely early feedback, reduced development cycle, increased transparency, better product quality, increased flexibility to accommodate changing customer requirements etc.
- Validation of all four hypotheses against three case studies proved the hypotheses to be true
- Case Study 1 – TINS (This is not Scrum) and Case Study 2 –Large Scale Scrum showed that Scrum is scalable and can be implemented successfully in larger teams
- Comparison of extended V-models showed FDD as the best approach which can be adapted inside traditional models, followed by Scrum, ATDD/Specification by example, Kanban and TDD
- Case studies as well as extended V-models supported the view that Scrum can be used as a framework inside traditional V-model. Within this framework, other agile testing methods like TDD, FDD, specification by example etc. can be embedded.
- Participation of other phases like conception and testing in agile methodologies reduces challenges for agile methods implementation inside development team
- Apart from Scrum testers, presence of independent test teams improved the product quality

14.2 Future work

There are three main areas on which future work could be based.

- **Study of other Agile testing methods**

There exists number of agile testing methods. Study of other agile testing methods and extended V-models apart from the ones discussed in this master thesis could be part of future work.

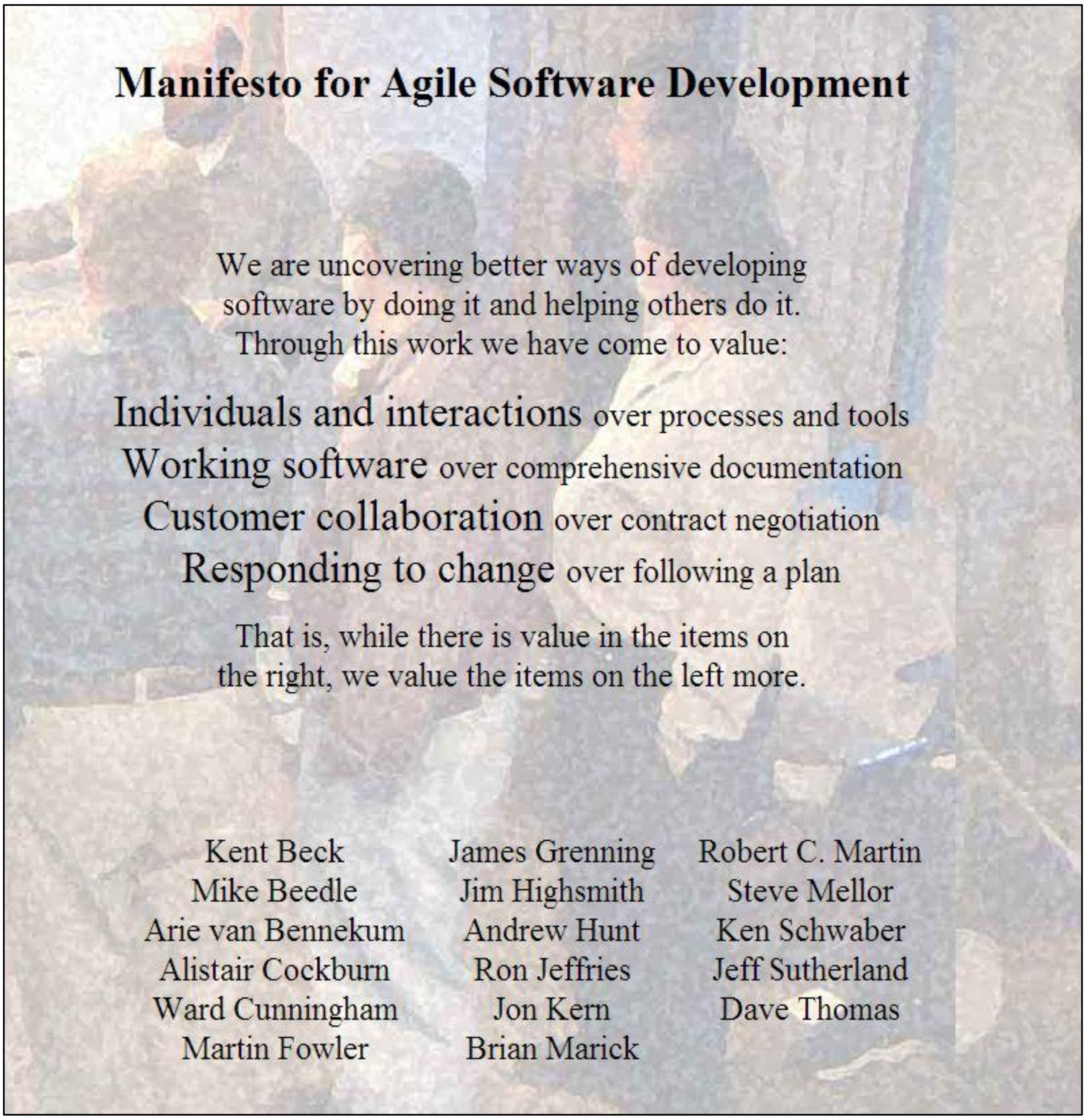
- **Extended V-models' validation**

The master thesis proposed some extended V-models. However, the adoption of agile testing methods and extended V-models depends upon the specific project environment. Extended V-models presented in this thesis could be implemented inside an organization to check their validity and viability.

- **Different Case studies**

All three case studies discussed in this thesis implemented Scrum agile methodology. Case studies implementing other agile methods like Kanban, FDD, ATDD etc. could be studied as part of future work. The three case studies belonged to public sector in Germany and showed that hybrid approaches combining agile and traditional methods worked well in public sector. Case studies in other sectors could also be studied to check if hybrid approaches work well in other domains too.

Appendix A - **Agile Manifesto**



Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

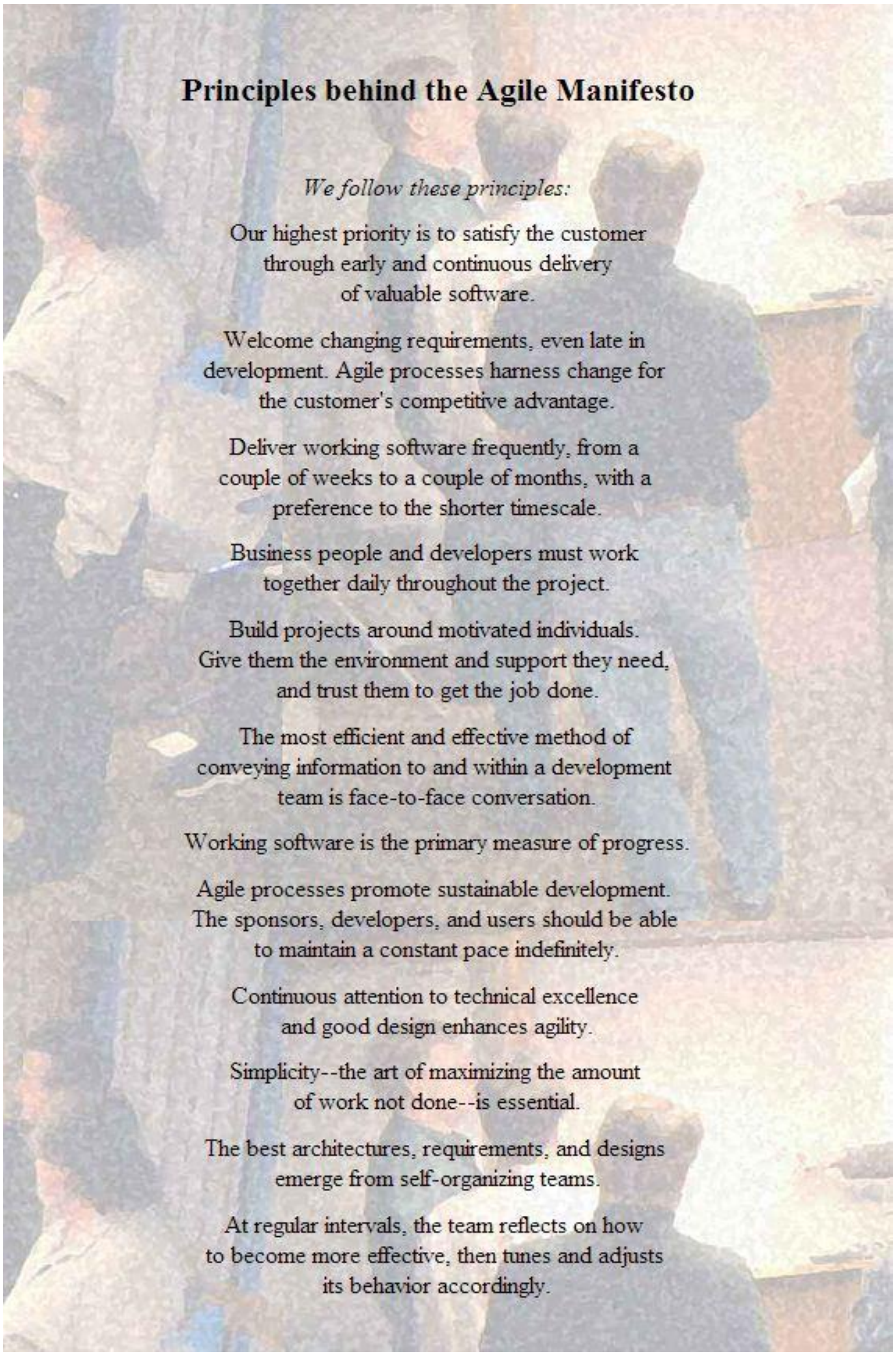
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

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

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

(Source: http://agilemanifesto.org/)

- Agile Principles



Principles behind the Agile Manifesto

We follow these principles:

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

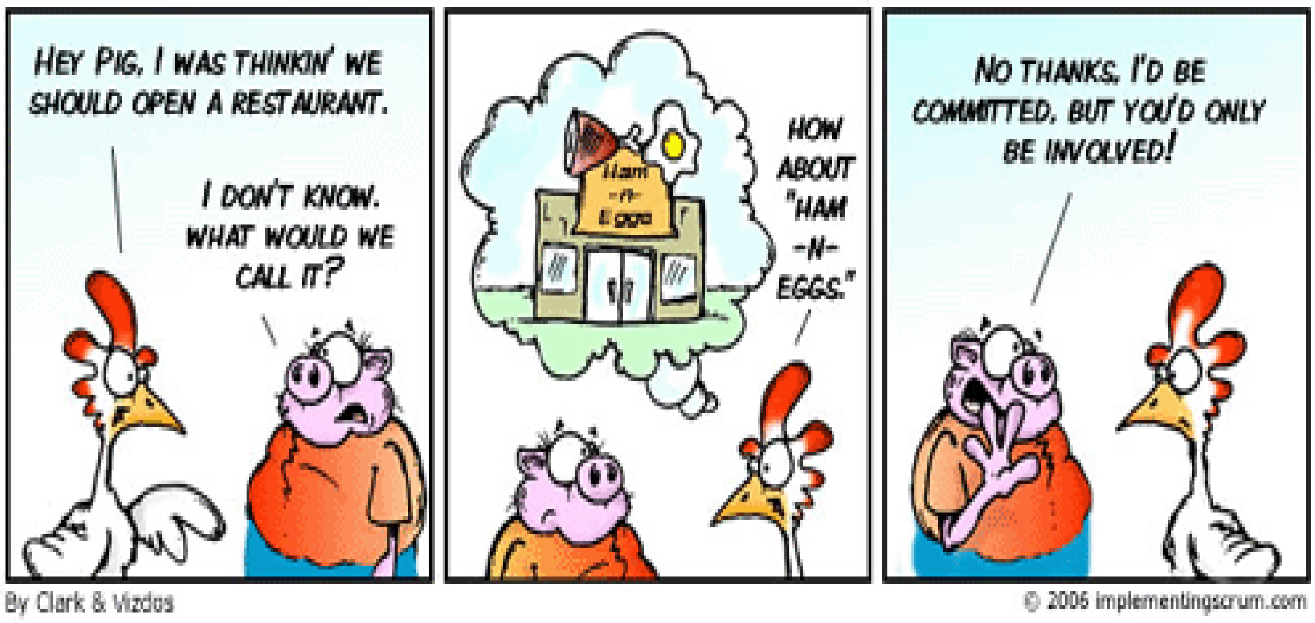
Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

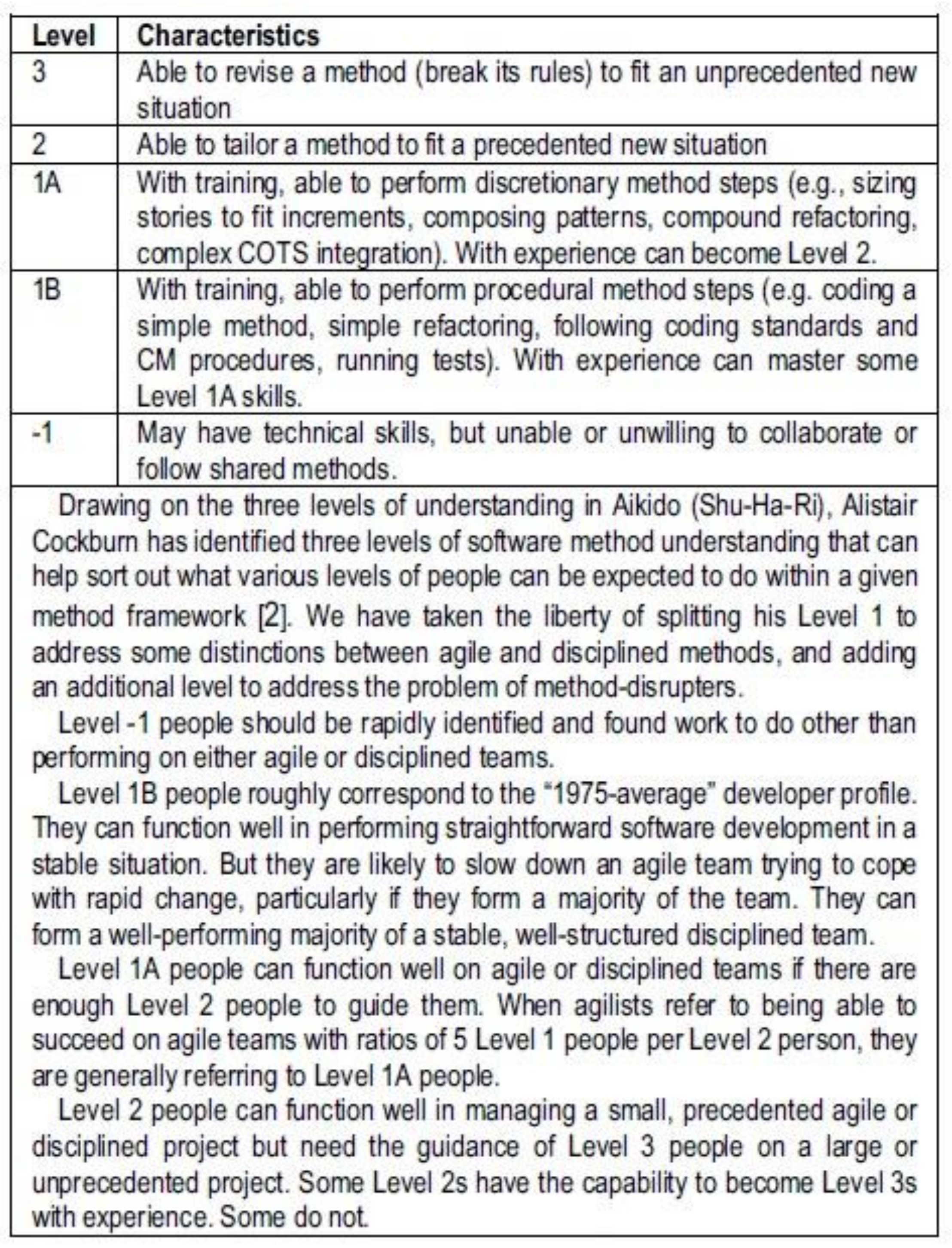
(Source: http://agilemanifesto.org/principles.html)

Appendix B - Chickens and Pigs analogy of Scrum Roles


HEY PIG, I WAS THINKIN' WE SHOULD OPEN A RESTAURANT.
I DON'T KNOW. WHAT WOULD WE CALL IT?
Ham -n- Eggs
HOW ABOUT "HAM -N- EGGS."
NO THANKS. I'D BE COMMITTED, BUT YOU'D ONLY BE INVOLVED!
By Clark & Vizdos
© 2006 implementingscrum.com

(Source: implementingscrum.com)

Appendix C - **Explanation of Personnel levels in Boehm's Home Ground Polar Chart**



Level	Characteristics
3	Able to revise a method (break its rules) to fit an unprecedented new situation
2	Able to tailor a method to fit a precedented new situation
1A	With training, able to perform discretionary method steps (e.g., sizing stories to fit increments, composing patterns, compound refactoring, complex COTS integration). With experience can become Level 2.
1B	With training, able to perform procedural method steps (e.g. coding a simple method, simple refactoring, following coding standards and CM procedures, running tests). With experience can master some Level 1A skills.
-1	May have technical skills, but unable or unwilling to collaborate or follow shared methods.

Drawing on the three levels of understanding in Aikido (Shu-Ha-Ri), Alistair Cockburn has identified three levels of software method understanding that can help sort out what various levels of people can be expected to do within a given method framework [2]. We have taken the liberty of splitting his Level 1 to address some distinctions between agile and disciplined methods, and adding an additional level to address the problem of method-disrupters.

Level -1 people should be rapidly identified and found work to do other than performing on either agile or disciplined teams.

Level 1B people roughly correspond to the "1975-average" developer profile. They can function well in performing straightforward software development in a stable situation. But they are likely to slow down an agile team trying to cope with rapid change, particularly if they form a majority of the team. They can form a well-performing majority of a stable, well-structured disciplined team.

Level 1A people can function well on agile or disciplined teams if there are enough Level 2 people to guide them. When agilists refer to being able to succeed on agile teams with ratios of 5 Level 1 people per Level 2 person, they are generally referring to Level 1A people.

Level 2 people can function well in managing a small, precedented agile or disciplined project but need the guidance of Level 3 people on a large or unprecedented project. Some Level 2s have the capability to become Level 3s with experience. Some do not.

(Source: Boehm & Turner, 2003)

Appendix D - 4-DAT framework's four dimensions

4-DAT's four dimensions	
Dimension 1 (Method Scope)	
Scope	*Description*
1. Project size	Does the method specify support for small, medium or large projects (business or other)?
2. Team size	Does the method support for small or large teams (single or multiple teams)?
3. Development style	Which development style (iterative, rapid) does the method cover?
4. Code style	Does the method specify code style (simple or complex)?
5. Technology environment	Which technology environment (tools, compilers) does the method specify?
6. Physical environment	Which physical environment (co-located or distributed) does the method specify?
7. Business culture	What type of business culture (collaborative, cooperative or noncollaborative) does the method specify?
8. Abstraction mechanism	Does the method specify an abstraction mechanism (object-oriented, agent-oriented)?
Dimension 2 (Agility Characterization)	
Features	*Description*
1. Flexibility	Does the method accommodate expected or unexpected changes?
2. Speed	Does the method produce results quickly?
3. Leanness	Does the method follow the shortest time span, use economical, simple and quality instruments for production?
4. Learning	Does the method apply updated prior knowledge and expe-

	rience to create a learning environment?
5. Responsiveness	Does the method exhibit sensitiveness?
Dimension 3 (Agile Value Characterization)	
Agile Values	*Description*
1. Individuals and interactions over processes and tools	Which practices value people and interaction over processes and tools?
2. Working software over comprehensive documentation	Which practices value working software over comprehensive documentation?
3. Customer collaboration over contract negotiation	Which practices value customer collaboration over contract negotiation?
4. Responding to change over following a plan	Which practices value responding to change over following a plan?
5. Keeping the process agile	Which practices help in keeping the process agile?
6. Keeping the process cost effective	Which practices help in keeping the process cost effective?
Dimension 4 (software process characterization)	
Process	*Description*
1. Development process	Which practices cover the main life cycle process and testing (quality assurance)?
2. Project management process	Which practices cover the overall management of the project?
3. Software configuration control process/support process	Which practices cover the process that enable configuration management?
4. Process management process	Which practices cover the process that is required to manage the process itself?

(Source: Qumer & Henderson, 2008)

Appendix E - Case Studies' Interview Questionnaire

History of Project	
1.	What is the size of project? Number of employees
2.	How old is this project?
3.	How many releases have been released so far?
4.	What is the length of each release and iteration?
Product developed	
1.	What is the product developed? Is it UI or some other software?
2.	What is the risk factor involved in this product?
3.	For which client is this product developed?
4.	How many users use this software?
5.	What is the technology used for development? JAVA, C..?
6.	What is the overall architecture? Client-Server, SOA or..?
Process Description	
1.	What is the process flow? Various phases followed in product development
2.	Input, Output and deliverable of each phase with emphasizes on Testing phases
3.	Are Testing phases followed in parallel with development or afterwards?
4.	How much is the customer interaction and involvement?
Testing Phases	
Checklist of Test phases: Which testing phases do you follow and who performs which testing phase?	
	✓ Unit Testing ✓ Integration Testing ✓ System Testing ✓ User Acceptance Testing (Alpha and Beta Testing) ✓ Load / Performance Testing

	✓ Regression Testing ✓ Any additional Test phases followed?
Agile Testing Methods	
1.	Which Agile testing practices or methods are used inside testing or development? Like TDD, FDD etc.
2.	How are these practices followed? Like use of boards or user stories or..?
Roles	
1.	What are the various roles involved in product development? Like analyst, developer, scrum master etc…?
2.	What are the respective tasks performed by these roles?
3.	How is the interaction between these roles?
Pros and Cons	
1.	What are the Pros & Cons of following Agile Testing methods inside V Model?
2.	What are the reasons for following such a mixed approach? Why not complete traditional or complete Agile?
3.	What is it that you would like to do better?
Challenges	
1.	What are the difficulties faced while implementing such a mixed approach?
2.	What is the overall impression of this approach?
Future Plans	
1.	Do you want to ultimately shift to complete Agile methodology?
2.	Why or why not?
Hypothese Validation	
1.	Hypothesis 1 – Valid or Invalid with respect to this project. Kindly provide some details. *“Agile testing methods when embedded inside V-model will lead to early*

	and often testing and hence bugs being detected earlier in the software development lifecycle"
2.	Hypothesis 2 - Valid or Invalid with respect to this project. Kindly provide some details. "*Agile testing methods when embedded inside V-model will lead to regular customer feedback and hence will be able to react quickly to changing customer requirements"*
3.	Hypothesis 3 - Valid or Invalid with respect to this project. Kindly provide some details. "*Agile testing methods when embedded inside V-model will lead to reduction in development cycle times*"
4.	Hypothesis 4 - Valid or Invalid with respect to this project. Kindly provide some details. *"Agile testing methods when embedded inside V-model build better quality product"*

References

Brykczynski, B., Meeson, R., & Wheeler, D. A. (1994). *Software Inspection : Eliminating Software Defects.* Alexandria, VA: Institute for Defense Analyses.

Ambysoft (2008, February). *Agile Adoption Survey*. Retrieved June 20, 2013 from http://www.ambysoft.com/surveys/agileFebruary2008.html

Ambysoft (2008, December). *Software Development Project Success Survey*. Retrieved June 10, 2013 from http://www.ambysoft.com/surveys/success2008.html

VersionOne (2012). *7th Annual State of Agile Development Survey*. Retrieved July 20, 2013 from http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-Survey.pdf

Royce, W. W. (1970, August). Managing the development of large software systems. In *proceedings of IEEE WESCON, 26(8).*

International Software Testing Qualifications Board, ISTQB (2012, October). *Standard glossary of terms used in Software Testing*. Version 2.2.

Agile [Def. 1]. (n.d.). In *Merriam Webster Online*, Retrieved July 25, 2013, from http://www.merriam-webster.com/dictionary/agile

Layton, M.C. (2012). *Agile Project Management for Dummies*. Hoboken, NJ: John Wiley & Sons.

Schwaber, K., & Sutherland, J. (2013, July). *The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game*, Retrieved September 12, 2013 from https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf#zoom=100

Manifesto for Agile Software Development (2001). Retrieved July 30, 2013 from http://agilemanifesto.org/

Principles behind the Agile Manifesto. Retrieved July 30, 2013 from http://agilemanifesto.org/principles.html

Boehm, B., & Turner, R. (2005). Management Challenges to Implementing Agile Processes in Traditional Development Organizations. *Software, IEEE*, *22*(5), 30–39.

Balaji, S., & Murugaiyan, M.S. (2012, June). Wateerfall Vs V-Model Vs Agile: A Comparative Study on SDLC. *International Journal of Information Technology and Business Management*, *2 (1)*, 26-30.

Qumer, A., and Henderson-Sellers, B. (2008). An Evaluation of theDegree of Agility in Six Agile Methods and its Applicability for Method Engineering. *Information and Software Technology (50)*, 280-295.

Schwaber, K., & Sutherland, J. (2012). *Software in 30 Days—How Agile Managers Beat the Odds, Delight Their Customers, and Leave Competitors in the Dust*. Hoboken, NJ: John Wiley & Sons.

Schwaber, K (2004). *Agile Project Management with Scrum*. Redmond, Washington: Microsoft Press.

Pham, A., & Pham, P.V. (2012). *Scrum in Action: Agile Software Project Management and Development*. Boston, USA: Course Technology.

Sliger, M., & Broderick, S. (2008). *The Software Project Manager's Bridge to Agility*. Addison-Wesley Professional.

Szalvay, V. (2004, November). An Introduction to Agile Software Development. *Danube Technologies*.

Ambler, S., & Lines, M. (2012). *Disciplined Agile Delivery—A Practitioner's Guide to Agile Software Delivery in the Enterprise*. Upper Saddle River, NJ: IBM Press.

Hightower, R., Onstine, W., & Visan, P. (2004). *Professional java tools for extreme programming*. Indianapolis, IN: Wiley Publishing.

Dooley, J. (2011). *Software Development and Professional Practice*. Apress.

Stephens, M., & Rosenberg, D. (2003). *Extreme Programming Refactored—The Case Against XP*. Apress.

Russo, B., & Scotto, M., & Sillitti, A., & Succi, G. (2010). *Agile Technologies in Open Source Development*. NY: Information Science Reference-Imprint of: IGI Publishing.

Poppendieck. M, Poppendieck. T (2003). *Lean Software Development: An Agile Toolkit*, Upper Saddle River, NJ: Addison-Wesley Professional.

Anderson, D.J. (2010, December). The Principles of the Kanban Method. Retrieved September 12, 2013 from http://www.djaa.com/principles-kanban-method-0

Bell, S. C., & Orzen, M. A. (2011). Lean IT—Enabling and Sustaining Your Lean Transformation. NY: Taylor & Francis.

Ambler, S.W. (2006). *Agile Testing and Quality Strategies: Discipline Over Rhetoric*. Retrieved September 17, 2013 from http://www.ambysoft.com/essays/agileTesting.html

Crispin, L., & Gregory, J. (2008). *Agile testing: A practical guide for testers and agile teams*. Pearson Education.

Carter, J. (2010). The Agile Tester. *VersionOne*. Retrieved from July 31, 2013 from http://www.versionone.com/pdf/WP_AgileTester.pdf

Bender, J., & McWherter, J. (2011). *Professional test driven development with C#: developing real world applications with TDD*. Indianapolis, IN: Wiley Publishing.

Collino, A. (2009, September). A Tester Perspective on Agile Test Driven Development. *Testing Experience, 18*.

Reid, S. (2009). Are All Pigs Equal. *Testing Experience, 2*.

Park, S. S., & Maurer, F. (2008, May). The benefits and challenges of executable acceptance testing. In *Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral, ACM,* 19-22.

Koudelia, N. (2011). *Acceptance Test-Driven Development*. University of Jyväskylä, Jyväskylä.

Hendrickson, E. (2008). Driving development with tests: ATDD and TDD. *Quality Tree Software, Inc.* Retrieved August 2, 2013 from http://testobsessed.com/wp-content/uploads/2011/04/atddexample.pdf

Ambler, S.W. (n.d.) *Feature Driven Development (FDD) and Agile Modelling*. Retrieved June 12, 2013 from http://www.agilemodeling.com/essays/fdd.htm

Palmer, S. R., & Felsing, M. (2001). *A Practical Guide to Feature-Driven Development*. Pearson Education.

MacDonald, K. J. (n.d.). *Agile Method Brief – Feature Driven Development (FDD)*. Retrieved June 20, 2013 from http://www.projectconnections.com/templates/detail/agile-techniques-fdd.html

Kaner, C. (2008, April). A Tutorial in Exploratory Testing. In *QAI QUEST Conference, Chicago*.

Bach, J. (2003). *Exploratory Testing Explained.* Retrieved 25 June, 2013 from http://www.satisfice.com/articles/et-article.pdf

Bach. J, & Bolton. M. (2012). Rapid Software Testing. Version (3.0). *Satisfice, Inc.* Retrieved 25 June, 2013 from http://www.satisfice.com/rst.pdf

Bach, J. (2000). Session-based test management. *Software Testing and Quality Engineering, 2*(6).

Guide to the Software Engineering Body of Knowledge, SWEBOK (2004). IEEE Computer Society.

Itkonen, J., & Rautiainen, K. (2005, November). Exploratory testing: a multiple case study. In *Proceedings of the 4th International Symposium on Empirical Software Engineering (ISESE 2005), 84-93*. IEEE.

Itkonen, J., Mantyla, M. V., & Lassenius, C. (2007, September). Defect detection efficiency: Test case based vs. exploratory testing. In *Proceedings of International Symposium on Empirical Software Engineering and Measurement, 61–70*. IEEE.

Adzic, G. (2011). *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Publications Co.

Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project,7007*.

Boehm, B., & Turner, R. (2003). *Balancing agility and discipline: A guide for the perplexed*. Addison-Wesley Professional.

Boehm, B. (2002). Get ready for agile methods, with care. *Computer, 35*(1), 64-69.

Vinekar, V., Slinkman, C. W., & Nerur, S. (2006). Can agile and traditional systems development approaches coexist? An ambidextrous view. *Information systems management*, *23*(3), 31-42.

Itkonen, J., Rautiainen, K., & Lassenius, C. (2005). Towards understanding quality assurance in agile software development. In *ICAM 2005*.

Nerur, S., Mahapatra, R., & Mangalaraj, G. (2005). Challenges of Migrating to Agile Methodologies. *Communications of the ACM*, *48*(5), 72-78.

Boehm, B., & Turner, R. (2005). Management Challenges to Implementing Agile Processes in Traditional Development Organizations. *Software, IEEE*, *22*(5), 30-39.

Schein, E. H. (2010). *Organizational Culture and Leadership* (Vol. 2). John Wiley & Sons.

Adler, P. S., & Shenhar, A. (1990). Adapting your Technological Base: the Organizational Challenge. *Sloan Management Review*, *25*(25-37).

Rong, G., Shao, D., & Zhang, H. (2010, November). SCRUM-PSP: Embracing Process Agility and Discipline. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific* (pp. 316-325). IEEE.

West, D. (2011). Water-Scrum-Fall Is The Reality Of Agile For Most Organizations Today. *Forrester, July, 26*.

Hayata, T., & Han, J. (2011, July). A hybrid model for IT project with Scrum. In *Proceedings of 2011 IEEE International Conference on Service Operations, Logistics and Informatics* (pp. 285-290). IEEE.

Schiffmann, J. (2012, February). Agile methods for (phase-oriented) test managers?. *Agile Record, 9,* 42-45.

Nawaz, A., & Malik, K. M. (2008). *Software Testing Process in agile development*. Blekinge Institute of Technology, Ronneby, Sweden.

Festinger, L. (1962). *A theory of cognitive dissonance* (Vol. 2). Stanford university press.

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). *Agile Software Development Methods*. VTT Publications (478).

Ladas, C. (2008, July). *Scrum-ban*. Retrieved August 22, 2013 from http://leansoftwareengineering.com/ksse/scrum-ban/

Anderson, D.J., & Garber, R. (2007). A Kanban System for Sustaining Engineering on Software Systems. *Corbis Corporation*. Retrieved August 22, 2013 from http://www.lean.org/FuseTalk/Forum/Attachments/Kanban%20for%20Software%20Development-Corbis.pdf

Hiranabe, K. (2008, January). *Kanban Applied to Software Development: from Agile to Lean*. Retrieved August 22, 2013 from http://www.infoq.com/articles/hiranabe-lean-agile-kanban

Riordain, I. P., & Burgt, B.V. (2011, December). Lean Test Management – The future of testing?. *Testing Experience, 16,* 4-7.

Abrahamsson, P., Warsta, J., Siponen, M. T., & Ronkainen, J. (2003, May). New directions on agile methods: a comparative analysis. In *the Proceedings of the International Conference on Software Engineering* (pp. 244-254). IEEE.

Germain, É., & Robillard, P. N. (2005). Engineering-based processes and agile methodologies for software development: a comparative case study. *Journal of Systems and Software*, *75*(1), 17-27.

Riehle, D. (2000). A Comparison of the Value Systems of Adaptive Software Development and Extreme Programming: How Methodologies May Learn from Each Other. In *Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2000)*. Retrieved 25 August, 2013 from http://www.riehle.org/computer-science/research/2000/xp-2000.html

Succi, G., Wells, D., Williams, L., & Wells, J. D. (2003). *Extreme programming perspectives* (Vol. 176, p. 312). Addison-Wesley.

Yin, R.K. (2013). *Case Study Research: Design and Methods* (Vol. 5). Thousand Oaks, California: Sage Publications.

Boldinger, S. (2013). Einführung agiler Vorgehensweisen in traditionellen Softwareentwicklungsprojekten. Master Thesis, Otto - Friedrich - University of Bamberg, Bamberg, Germany.

Anger, R. & Eichler, F. (April, 2013). Keine Ausreden: TestAutomatisierung in Agilen Projekten. *Objektspektrum*. Retrieved August 15, 2013 from http://www.sigs-data-com.de/fileadmin/user_upload/zeitschriften/os/2013/04/anger_eichler_OS_04_13_mvut.pdf

Bentley, J. (1985). Programmimg pearls. *Communications of the ACM, 28*, 896–901.

Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *Computer, 21*(5), 61-72.

The Expertise Paradox (2011). Retrieved September 18, 2013 from http://blog.lib.umn.edu/hamdi002/blog/2011/11/the-expertise-paradox.html

Eikenberry, K. (2009, March). *The Paradox of Expertise*. Retrieved September 18, 2013 from http://salesandmanagementblog.com/2009/03/18/guest-article-the-paradox-of-expertise-by-kevin-eikenberry/

Jansen, J. J., Van Den Bosch, F. A., & Volberda, H. W. (2006). Exploratory Innovation, Exploitative Innovation, and Performance: Effects of Organizational Antecedents and Environmental Moderators. Management science, 52(11), 1661-1674.

West, D., Grant, T., Gerush, M., & D'Silva, D. (2010). Agile development: Mainstream adoption has changed agility. *Forrester Research.*