THE SCRUM PROCESS FOR INDEPENDENT PROGRAMMERS

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

_____

Ananth Srirangarajan

Certificate of Approval:

_____	_____
Pradeep Lall, Co-Chair Thomas Walter Professor Mechanical Engineering	David A. Umphress, Co-Chair Associate Professor Computer Science and Software Engineering
_____	_____
Cheryl Seals Assistant Professor Computer Science and Software Engineering	George T. Flowers Dean Graduate School

THE SCRUM PROCESS FOR INDEPENDENT PROGRAMMERS

Ananth Srirangarajan

A Thesis

Submitted To

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama
May 9, 2009

THE SCRUM PROCESS FOR INDEPENDENT PROGRAMMERS

Ananth Srirangarajan

_____
Signature of Author

_____
Date of Graduation

THESIS ABSTRACT

THE SCRUM PROCESS FOR INDEPENDENT PROGRAMMERS

Ananth Srirangarajan

Master of Science, May 9, 2009
(B.S., Champlain College, 2004)

61 Typed Pages

Directed by David Umphress and Pradeep Lall

A software process is an attempt to impose structure on the software development process. The primary goal of a software process is to arrive at a repeatable, predictable process that will raise the productivity of software developers and enhance the quality of their work. Over the past few decades, many such processes have been described and implemented ranging from the traditional Waterfall model to the more recent ones that are collectively grouped as Agile Processes.

Agile Processes, in particular, have received considerable attention as they are light-weight, people-centric and relatively easy to understand and implement. However, almost all agile processes are geared towards development projects involving teams of programmers working together and ignore the needs of a programmer working alone.

This study describes an attempt to apply the tenets of the Scrum process in situations where the software development is performed by a single programmer. It was found that agile principles, as adhered to by Scrum, can be applied to projects with just

one programmer, resulting in better forecasts of the work involved, regular releases of working, verifiable software as well as improvements in quality of the code.

ACKNOWLEDGEMENTS

I would like to thank Dr. David Umphress for guiding me through the process of writing this thesis. I would like to express my deepest gratitude to Dr. Pradeep Lall, who funded me throughout my studies at Auburn, and without whom I could have never have finished my graduate studies. I would also like to thank my committee member, Dr. Cheryl Seals for her time and cooperation.

I am deeply grateful to my employer Mentor Graphics and my manager Mr. Paul Batson for facilitating my research work. Thanks are also due to my friends Manoj Rajagopalan and Santosh Kulkarni for their unfailing support and help.

I would like to take this opportunity to acknowledge all that my parents and sister have done for me. I am humbled by their faith in me and this work is as much their achievement as it is mine. Last, but not least, I wish to thank the Lord above for all his blessings.

Style manual used: ACM Computer Surveys.

Computer software used: Microsoft Office 2003.

TABLE OF CONTENTS

LIST OF FIGURES x

1. INTRODUCTION 1

1.1 Background 1

1.2 Problem Statement 4

2. LITERATURE REVIEW 6

2.1 Software Process Goals 6

2.2 Principles of Agile Processes 6

2.3 Scrum 10

2.4 Extreme Programming 13

3. SOLUTION 17

3.1 Approach 18

3.2 Why Scrum and not XP? 18

3.3 Scrum for Independent Programmers 19

3.3.1 Product Backlog 20

3.3.2 Sprint Planning 21

3.3.3 Sprint Tracking 24

3.3.4 Sprint Review 24

4. SOLUTION VALIDATION 26

4.1 Case Study 1 26

4.1.1 Case Study 1: Product Backlog 27
4.1.2 Case Study 1: Sprint Planning 27
4.1.3 Case Study 1: Sprint Tracking 29
4.1.4 Case Study 1: Sprint Review 30
4.1.5 Case Study 1: Results 31
4.2 Case Study 2 31
4.2.1 Case Study 2: Product Backlog.....32
4.2.2 Case Study 2: Sprint Planning 33
4.2.3 Case Study 2: Sprint Tracking 34
4.2.4 Case Study 2: Sprint Review 35
4.2.5 Case Study 2: Results 36
5. CONCLUSION AND FUTURE WORK 37
BIBILIOGRAPHY 39
APPENDIX A 42
APPENDIX B 47

LIST OF FIGURES

Figure 2.1: The Scrum Process 11

Figure 2.2: Extreme Programming Process 14

Figure 3.1: Scrum for Teams vs. Scrum for Independent Programmers 23

Figure 3.2: Scrum Process for Independent Programmers 25

Figure 4.1: Scan of document showing Stories to be completed during the Sprint.....28

Figure 4.2: Sprint Backlog at the beginning of the sprint.....29

Figure 4.3: Updating the Sprint Backlog during the Sprint to enable tracking 30

Figure 4.4: Screenshot of Product Backlog 32

Figure 4.5: Stories with their points in the Sprint.....33

Figure 4.6: Sprint Backlog with stories broken down into Tasks.....34

Figure 4.7 Burndown Chart with X-axis = Days and Y-axis = Hours.....35

1. INTRODUCTION

1.1 Background

In the last few decades, the single most important factor shaping the lives of people has been technology. Technology has completely transformed the way we live, communication, and work. Technology-led and enabled changes have even changed the social dynamics of human beings. Technology has permeated into every level of our lives to an extent where even simple everyday tasks requires that we interface with some form of computing machine. Of course, the more complicated tasks like weather forecasting, gene sequencing, air traffic control etc. would be impossible without large amounts of computing power. Underpinning this technology dependent world are millions and millions of lines of code: software.

The earliest pieces of code were produced in an ad-hoc manner which meant there was almost no way to measure and catalog the quality and performance of the code. This is fine when the software is only used in research or academic environments. But as more and more real world tasks become dependent on technology and its underlying software, ensuring the quality of the software has become a priority. Bad software can be catastrophic in mission-critical and life-critical situations.

The only way to control the quality of code is to monitor it when it is being produced, in effect, placing a structure on software development that will ensure the production of high grade code. This structure imposed on the development of software is

called software process or software development process. This structuring has many other benefits. Overtime enough data can be accumulated from previously executed projects, which will allow us to predict how long a current software project will take to be completed, given its scope and complexity. Process also increases the visibility into the thought process that went into development effort, which in turn makes it possible for the development effort to continue even when some members of the development team leave and new ones join in.

Many different software development processes have been defined and employed both by the industry and academia including those that belong to the agile family. But before any discussion about processes, it would be useful to understand most basic way in which structure is imposed on software development, namely, the Waterfall Model.

The Waterfall Model divides the whole lifecycle of software development into separate phases. These phases are: Requirements Analysis, Software Design, Software Implementation, Testing, Installation and Maintenance [Parekh 2005]. This model is still widely used and has many advantages. First, the phased development cycle enforces discipline with clear start and end points and markers for progress. Second, since the requirements are clearly defined at the beginning, there is minimal wastage of time and effort when actual coding starts. Also, it is much easier to catch and correct flaws at the design stage than at the testing phase because tracking down an error after all components have been integrated is not a trivial task [Contributor Melonfire 2006].

The Waterfall model has come to face a lot of criticism, though. Among the drawbacks that are listed most often is that most of the time customers have only a vague idea of what they want. So, the requirements may change or evolve over time, but the

model insists that the requirements document written in the beginning is iron-clad and not open to change. The Waterfall model is, thus, too linear and not very flexible [Parekh 2005].

The agile family of processes attempt to structure software development effort by embracing a much more light-weight, fluid and adaptable methodology. The Agile Manifesto declares the four values that must be given high prominence in all agile processes [Beck et al. 2001].

First, individuals and interactions are of more value than processes and tools [Beck et al. 2001], which implies that though process and tools are important, interaction of skilled individuals working on the project is of even greater significance [Fowler and Highsmith 2001].

Second, producing working software is more useful than a creating comprehensive documentation [Beck et al. 2001]. The focus must be on delivering the final product which is the working software. The amount and details of documentation that must be produced is completely up to the people working on the product [Fowler and Highsmith 2001].

Third, customer collaboration is of more value than contract negotiation [Beck et al. 2001], which means that while contract negotiations, where everyone's rights and responsibilities are clearly laid out, is important, it cannot replace communication. Successful developers work closely with their customers, they invest considerable effort to discover what their customers need, and they educate their customers along the way [Ambler 2006].

Agile processes must be geared towards responding to change rather than just following a plan [Beck et al. 2001] because more often than not the customer's priorities may change. While having a project plan is useful, the plan must be malleable. Otherwise, in the event of a change in the requirements the plan quickly becomes irrelevant [Ambler 2006].

Agile processes are being used widely in the software industry today with considerable success. Some of the most widely used agile process models are Extreme Programming (XP) [Beck 2000], Lean Development [Poppendieck and Poppendieck 2003], Crystal [Cockburn 2002] and Scrum [Schwaber and Beedle 2002].

1.2 Problem Statement

It is important for any software project to adhere to a process. Following a well-defined process gives developers clear goals, the order of the tasks that must completed to achieve these goals and a way to measure the progress being made to attain these goals [Tyrrell 2000].

Agile processes, which are becoming increasingly popular, are usually used for projects that involve small teams of developers, testers, designers etc. In fact, literature about agile processes talks exhaustively about teams: team building, team self-organization, daily meeting of team members, meetings between team members and customers, pair programming and so on. This is clearly a result of the emphasis placed on individuals and interactions as per the values listed in the Agile Manifesto [Beck et al. 2001]. But all this attention paid toward teams misses the important fact that there are

still many software projects and programs being written and maintained by independent programmers.

An agile process when used by independent programmers can still adhere to all the core values of the agile movement. An individual programmer would, in most cases, prefer to produce working software over writing a long requirements document. He or she can easily collaborate with their customer. In fact, it may easier to coordinate with a customer when only one programmer is involved. And an individual programmer can respond to a change in the customer's requirements.

This work attempts to apply the agile process Scrum to the development effort of an individual programmer. Using Scrum will help independent developers because by its very nature, Scrum is very adaptable and not heavy. A developer can quickly learn and use this process which will result in improvements in planning, scheduling and quality of code along with the improvements in the developer-client communication.

2. LITERATURE REVIEW

2.1 Software Process Goals

Before taking a more detailed look at Agile Processes, it may be beneficial to look at software processes in general and their purpose. Any software process is only useful when it helps streamline and quantify the software development effort. To this end it must meet certain goals, which are effectiveness, maintainability, predictability, repeatability, quality, improvement and tracking [Tyrrell 2000]. A process is only effective when it produces software meets the requirements of the customer. It must allow for changing requirements and other such problems that may make it to necessary to go back and review previously completed work. It must be able to predict the length of the development effort by taking into account the available resources. A process must lend itself for reuse in other similar projects. It must ensure a high quality product and lastly must allow the managers, developers and customers to track the status of the project [Tyrrell 2000].

2.2 Principles of Agile Processes

The goals described in the previous section are the goals that all software processes must strive for. In order to achieve these goals, agile processes follow a set of principles. The Agile Manifesto, a document written by founders of the agile movement, lists twelve principles that form the basis of agile processes [Beck et al. 2001]. An

understanding of each of these principles is essential in order to successfully apply and use an agile process.

- "The highest priority of agile processes must to be to satisfy the customer through early and continuous delivery of valuable software" [Beck et al. 2001]. The customers only concern when initiating any software project is that the end product works. So, any other artifacts produced along the way like requirements documents, class diagrams etc., while useful, are of little value from the customer's perspective. Also, since modern projects must often deal with changing requirements, the initial project plan may have to be revised constantly. This shows that only working code can be seen as the measure to progress rather than meeting the original project plans [Fowler and Highsmith 2001].
- Secondly, agile processes welcome changing requirements, irrespective of when this change occurs. Agile processes actively welcome change and use it to increase the customers' competitiveness in a fast changing market place [Beck et al. 2001]. Surviving in a cut-throat market place requires that businesses be adaptable. This means that software development teams working to meet the needs of these businesses must also be ready to meet changing requirements. Instead of resisting changes, agile process must enable developers to manage the change [Fowler and Highsmith 2001].
- Thirdly, agile processes attempt to deliver working software as often as possible, ranging from a couple of weeks to a couple of months [Beck et al. 2001]. Frequent delivery of working software provides stakeholders with a clear way of

measuring progress. Being able to see the software actually working will also enable them to provide better guidance to the development team [Ambler 2006].

- Agile processes must enable collaboration between customers and the developers throughout the development cycle [Beck et al. 2001]. Such close collaboration is necessary in order for the developers to get constant feedback.
- Projects must be built around motivated individuals by giving them the environment and support they need [Beck et al. 2001]. No process will be effective if the people using it are not motivated and committed to using it. So, all support must be extended to those who know the project best and these individuals must be allowed to make key decisions about the direction of the project [Fowler and Highsmith 2001].
- Agile processes prefer that teams convey information in face-to-face conversations [Beck et al. 2001]. Direct communication is better than any requirements document or UML diagram in ensuring that everyone on the team has a clear understanding of what must be done in order to successfully complete a project.
- "Working software is the primary measure of progress used by agile processes" [Beck et al. 2001]. This ensures that there are no last minute problems when the final product is delivered as throughout the cycle the product has been developed using increments of working code.
- Agile processes must promote sustainable development. This means that teams must be able to work at a constant pace indefinitely [Beck et al. 2001]. This principle comes from the fact that it is not possible to successfully develop

software by forcing people to work overtime [Ambler 2006]. Instead the process strives to set a steady and sustainable pace of development work.

- Agile processes require that continuous attention be paid to technical excellence and good design [Beck et al. 2001]. It is easier and less time consuming to maintain and build on high-quality software than it is to do the same with code of inferior quality [Ambler 2006].
- Simplicity is the next principle that agile approaches value [Beck et al. 2001]. There's a strong taste of minimalism in all the agile methods [Fowler and Highsmith 2001]. Any software development task can be approached with a host of methods, but it's particularly important to use simple approaches, because they're easier to change. It's easier to add something to a process that's simple.
- Agile processes encourage self-organizing teams as the best architectures, requirements and designs emerge from teams which are highly integrated and open to communication [Beck et al. 2001] [Fowler and Highsmith 2001].
- The last principle of agile processes states that development teams must reflect on how to become more effective at regular intervals and adjust its behavior accordingly [Beck et al. 2001]. Thus, agile processes enable process improvement by encouraging the team itself to take the lead in making changes and adaptations to the process.

The processes that are part of the agile family attempt to meet these twelve principles in a variety of different ways. An understanding of a couple of agile processes will be a helpful in understanding how for individual programmers can reap the benefits of using

agile processes. The two processes that were studied for this purpose were Scrum and Extreme Programming (XP). These two were selected because they are among the most popular agile processes in use today [Davidson 2008], in addition to being widely discussed in agile literature.

2.3 Scrum

Scrum in among the oldest of the agile processes. The initial idea of Scrum came from a paper about how to set up self-organizing teams and the management's role in the process [Takeuchi and Nonaka 1986]. The process was then formalized in 1995 by Dr. Jeff Sutherland and Ken Schwaber [Sutherland and Schwaber 2007].

Scrum is a simple process used to organize teams and get work done more productively with higher quality. It is a light-weight approach to software development that allows teams to choose the amount of work to be done and decide how best to do it.

Scrum divides the whole development cycle into a series of iterartions called Sprints. Each sprint is usually 1 – 4 weeks in length. The length of the sprint is fixed and it is not changed even if the work is not completed [Deemer and Benefield 2007]. These iterations are continued until the project is completed. The Scrum process is illustrated in Figure 2.1.

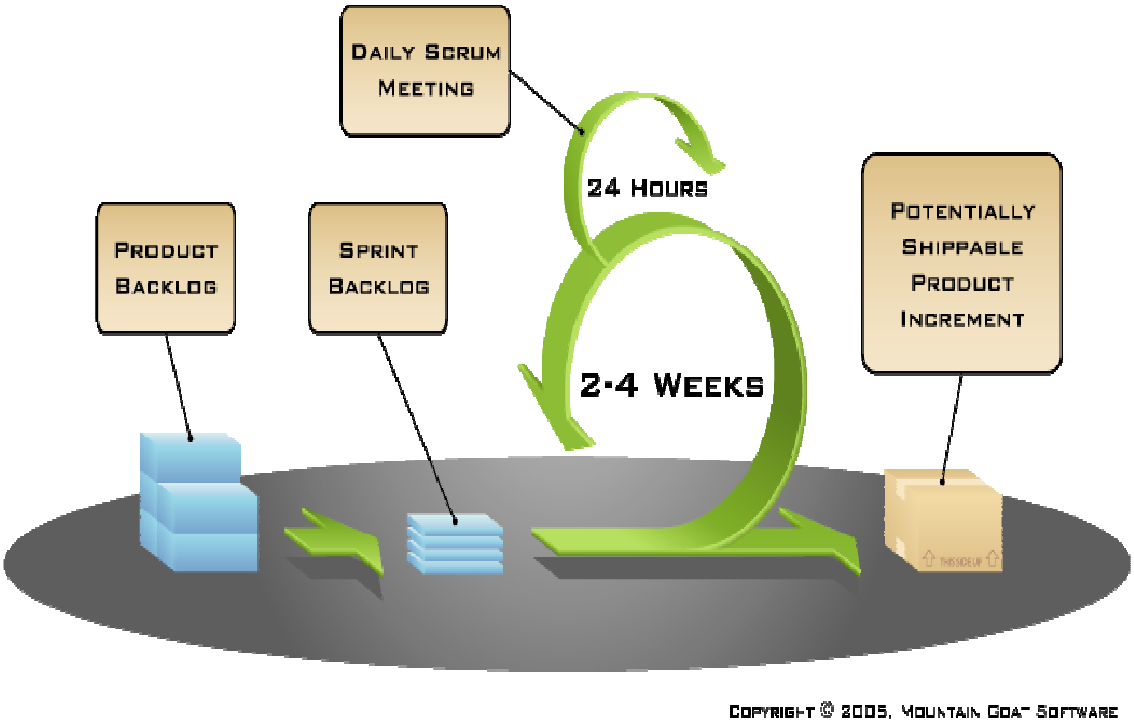


Figure 2.1: The Scrum Process [Cohn 2005]

The first step in Scrum is to arrive at a preliminary vision for the product. This vision is then converted into a rudimentary requirements document which lists all the features the system must have ranked according to the priority assigned to it by the customer. This is called the Product Backlog, and it evolves over the lifetime of the project [Deemer and Benefield 2007].

In the next step, the development team goes through the product backlog starting from the top and picks the items that they think they can complete during the forthcoming sprint. This list of items to be completed during the sprint is called the Sprint Backlog and it is never changed during the course of the sprint. This is one of the key practices in Scrum: rather than the managers deciding how much the team must complete, the team

itself arrives at a consensus regarding the workload they will take on [Deemer and Benefield 2007].

During the course of the Sprint, the development team gathers everyday to report and update each other on their progress. This progress is tracked using metrics such as the Burndown rate and/or through the use of simple charts called Burndown Charts which indicate the work that has been done and the tasks as yet unfinished [Deemer and Benefield 2007].

At the end of the Sprint, the development team meets with customers; a meeting that is referred to as the Sprint Review. In the sprint review, the team demonstrates the working code that they have worked on during the sprint and gathers feedback from the customers [Deemer and Benefield 2007].

The team uses the feedback that it received in the Sprint Review and from any problems it may have encountered during the sprint as the input for the Sprint Retrospective. This is an inspection of project progress at the end of the every Sprint. The goal is to improve development process by introducing new practices, changing existing practices, etc.

Thus, it can be clearly observed that Scrum meets all the principles of an agile process. The focus on delivering working software at the end of regular intervals, the constant communication between the team and the customers ensures that customers are kept satisfied and are always in the loop about the progress of the project. The evolution of the product backlog allows the team to adapt to changing requirements. The consensus within team that decides how much workload is assumed in a sprint allows for self-organizing teams and leaves key decisions to them. The daily meeting facilitates regular

communication maintaining the cohesion within the team and this repetitive cycle can be maintained indefinitely.

2.4 Extreme Programming

Extreme Programming (XP) is one of the more popular members of the agile family of processes. It was formalized by Kent Beck starting from 1995 and popularized by his book - Extreme Programming Explained: Embrace Change [Beck 2000]. XP gets its name because it takes common sense principles and practices, such as unit tests and code reviews to the extreme, leading to practices such as test-driven design and pair programming [Beck 2000] [Miller 2002]. XP is typically recommended only for teams of roughly 2 – 10 people who are co-located and have experience working with each other [Highsmith 2000]. XP lists a set of practices that attempts to eliminate the unnecessary artifacts of most heavyweight processes and allows the development team to focus on the coding without any distractions [Miller 2002].

XP refers to the process of defining the project scope and deciding what tasks need to be completed in any given iteration as the Planning Game. The whole project is divided into releases, each of which is actually rolled out to all the customers to be used in real-world situations. The details about what items will be in any given release and when the release will occur depends on the customers. The development team only focuses on completing the functionality needed for the current iteration [Beck 1999]. XP is shown diagrammatically is Figure 2.2.

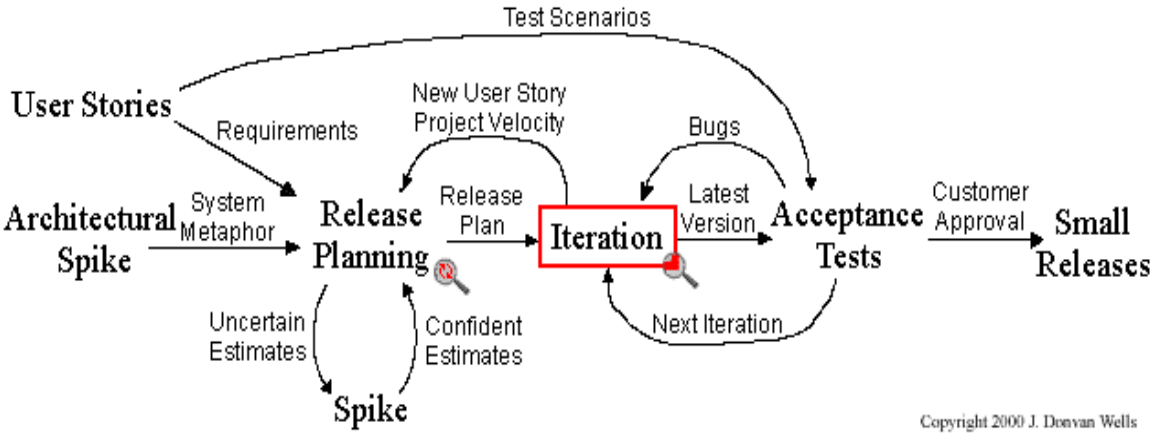


Figure 2.2: Extreme Programming Process [Wells 2000]

The releases that are rolled out to the customers are kept quite small. The system is released to the customers every few months even before the whole solution is in place. New releases are made often—anywhere from daily to monthly [Beck 1999]. This allows for feedback to be gathered when the system is put to real-world use.

The entire structure of the system is described by a 'metaphor' that is agreed upon by the both the programmers and the customers [Beck 1999]. Individual features are described as 'stories', which are gathered by simply asking the customers to explain the various features they would like in the system [Highsmith 2000].

XP encourages teams to keep the design of the software as simple as possible by focusing on delivering the functionality for the current iteration without any thought given to any future functionality [Highsmith 2000]. The design must meet all required tests and communicate everything that the developer wants communicated [Beck 1999].

XP places heavy emphasis on testing to such an extent that it requires the whole design to be test-driven. XP uses two types of testing: unit testing and functional testing. Units tests are written by the programmers before they even code a story or feature

[Highsmith 2000]. Further, the unit tests should be automated in order to receive instant feedback. Functional tests are written by the customers who use these tests to check the entire feature or a group of features [Beck 1999].

One of XP's unique practices is refactoring. Refactoring allows the system to be in a constant state of redesign [Highsmith 2000]. This ensures that the project can easily absorb changes.

XP advocates team members to code in pairs i.e., all production code is written by two people at one screen/keyboard/mouse. This collaborative programming allows for two minds to be actively engaged in looking over the code. This acts as a sort of code review and results in more defects being caught at the development stage itself [Highsmith 2000]. The code is collectively owned by the entire team, which allows any programmer to change any code when he or she sees an opportunity for improvement [Beck 1999]. The customer or at least a representative of the customer must be part of the team and must be available to answer questions or issue clarification to the team at any time [Beck 1999].

The new code that produced must be continuously integrated with the code written earlier. This is done in order to avoid integration errors which can create serious problems later on in the cycle. The integrated code must then pass all the tests, both unit and functional, failing which the changes are discarded [Beck 1999].

XP is a rather radical process model which is sometimes criticized as being too difficult to adopt and use, but it does clearly meet all the requirements for being an agile process. The small releases and the presence of customers as part of the teams allows for the customers to be part of the development process where they can clearly see their

system taking shape. The focus on simple design, pair programming, constant testing and integrations ensures high quality code working code and refactoring allows for developers to be prepared for changing requirements.

3. SOLUTION

3.1 Approach

A software process that will be of use for individual programmers presents a unique set of challenges. To begin with, developers working on their own must be convinced that using a process would be highly beneficial to them. Most of these developers simply work ad-hoc. All the requirements, design details etc. are held as a mental map by the developer alone, which makes it very difficult for someone else to read the code and make any changes or enhancements. Also, working ad-hoc makes it very difficult for the developer himself to estimate how long a project may take to finish or how difficult it might be.

Individual developers also suffer from the fact there are very few software processes that pay attention to their needs. One of the few that does is the Personal Software Process (PSP). However, PSP is a very heavy-weight. This results in very few people actually adopting and actively using PSP. Even those who do try to use it often give up when they are not required to do so. This is mainly because of two reasons: the high overhead of PSP-style metrics collection and analysis, and the requirement that PSP users need to constantly switch between product development and process recording which often breaks the their concentration [Johnson et al. 2003].

Therefore, the most important factor that was taken into account when choosing a process for individual programmers in this study is that it be as light-weight as possible;

it must not involve too much overhead in terms of time and it must not distract the developer from this primary task: coding. At the same time it must allow the developer to gather some basic data that will allow him to track progress, gather feedback from the customer and over-time is able to estimate the duration of any given project that comes his or her way based on the track record from the previous ones. Such a process, with its light foot print, will be able to overcome the resistance that most lone programmers have towards following a structured development process.

A process based on agile principles is ideal in this situation, as by definition, they are light-weight. The process was used in this study is agile process Scrum.

3.2 Why Scrum and not XP?

Of all the agile process that are defined and in use today, the two that were researched for this study were Scrum and Extreme Programming (XP), because they are the two most popular and widely used agile processes [Davidson 2008]. However, the process that was used in the single programmer environment of this study was Scrum. XP though highly effective has certain practices which are difficult to follow when only one programmer is involved in the development.

The first and the most obvious problem with using XP is that it promotes pair programming which is altogether impossible in a single developer environment. This problem may be overcome by having some other person go through a developer's code but this reduces the activity to a simple code review without any of the benefits of the pair programming. Scrum, however, does not have any stipulation pertaining to programming in pairs.

Secondly, XP advocates test-driven design. Though, this is an excellent practice, some independent developers may not be inclined to write tests for each and every unit. Also, XP calls for any changes that do not meet the tests to be discarded, something that single programmers will probably not do.

Thirdly, XP, in its purest form, requires the customers to be a part of the team. In a single developer environment, it is too much to expect the customer to be available all the time. On the other hand, Scrum, which allows for customers to be available in case developers have any questions or to issue clarifications, lends itself to be more easily adapted to a single programmer environment.

After taking into account the above factors, it was felt that Scrum is better as a candidate for a process for individual programmers. Scrum has proved that it can handle enormously complicated software development efforts. The literature on Scrum talks of projects involving hundreds of programmers, designers, testers etc. both co-located and geographically dispersed that have been successful. Though this focus on teams is a recurring theme, many of Scrum's core practices are such that they can be easily modified to work in a single programmer environment. The remaining sections in this chapter describe the how the core practices of Scrum were adapted to a single developer environment.

3.3 Scrum for Independent Programmers

The scaled down version of the Scrum process described in this study is meant to be used primarily by programmers working on their own. Using Scrum will help lone developers manage their software development lifecycle. The adapted version of Scrum

described here only defines the basic practices that must be followed leaving many of the internal decisions such as which metrics need be tracked and how they are tracked to the will of the user of the process. This allows the process to be flexible and adaptable and ensures that a broad array of users will be able to use it; from students to software professionals. This flexibility is essential as individual programmers' work practices vary widely; if the process is too cumbersome, it will dramatically increase the chances that users will abandon using it. The process must consume as little time as possible while attempting to capture at least some basic artifacts from each cycle that will be helpful to the user in later life cycles. The core practices of the Scrum process as applied in single programmer environments is explained in the following sections. Figure 3.1 shows a table comparing the use of Scrum by a team vis-à-vis Scrum as applied by independent programmers.

3.3.1 Product Backlog

The first step in Scrum is to arrive at the vision for the project and what it will do. All the requirements that are necessary for this vision to be realized are then put together as a list with the highest priority features at the top. This list is not absolute and final. It is continuously updated and refined by the developer and the customer together. The Product Backlog will include a variety of items, such as features, development requirements, exploratory work, and known bugs. Each of these requirements may be referred to as "stories": simply a descriptive way of presenting a requirement that makes sense to both the developer and the customer.

The Product Backlog is not a formal requirements document. It is meant to be dynamic; it is updated to reflect the changing needs of the customers, changes necessitated because of feedback from customers after they view the work from previous iterations, technical hurdles that appear, and so forth. Each of the stories is then analyzed and the developers arrive at a rough estimation of how much effort it will take to finish a particular story. These estimations need not be in any real-world unit like hours but using a system of points, where certain number of points roughly translates into the difficulty level of the story [Deemer and Benefield 2007]. The use of points is, however, discretionary. The customer takes these estimations into consideration when prioritizing the stories.

The stories in the Product Backlog can vary significantly in size, ranging from ones that can may take a few days work to ones that take a couple of hours; however, the larger ones can often be broken into smaller pieces during Sprint Planning, and the smaller ones may be consolidated. The amount of detailed specifications that is written for each story is up to the customer and the developer. The detail in the specification may also vary from one Product Backlog story to the next.

3.3.2 Sprint Planning

Since Scrum is an agile process the work is done in a series of iterations. A single iteration is referred to as a Sprint. A typical sprint should last anywhere from one – four weeks. Before the start of every sprint the developer and customer meet to discuss the Product Backlog and prepare for the forthcoming sprint. This meeting is called as Sprint Planning.

The first step is for the developer to determine how many hours a day he can dedicate to the project. This number should be realistic and take into account things like answering emails, lunch breaks or if the developer is a student, his or her class schedule. This is an important step as the work done during the sprint is tracked using hours required to complete each story.

During Sprint Planning, the developer selects the stories from the Product Backlog to commit to complete by the end of the Sprint, starting at the top of the Product Backlog and working down the list. In others words, stories that are of the highest business value for the customer are considered first before going on to the ones lower down in the Backlog. This is done until all the developer's available hours are used up. This is an important as it allows the developer to decide which stories he will complete during the coming Sprint. This list of stories that he or she commits to finished by the end of the sprint is called as the Sprint Backlog. The Sprint Backlog is never changed as long the Sprint is still in progress. This ensures that the customer does not interfere in development work by adding new stories to the sprint which will definitely be disruptive to the programmer.

As a further step, the developer can break down the individual stories into tasks. This can be beneficial if story is quite large or if he or she would like better visibility into the steps required in completing a story. The hours assigned to the story is divided up among the tasks in this case, with each task carrying the hours it would take to complete it. But this step is optional, with the decision left up to the developer's discretion. Once the Sprint Backlog is completed the sprint commences in earnest.

Process Element	Scrum for Teams	Scrum for Independent Developers
Duration of Iteration	▪ Sprints of 1 – 4 weeks	▪ Sprints of 1 – 4 weeks
Participants	▪ Scrum Team ▪ Scrum Master ▪ Product Owner	▪ Developer ▪ Customer
Product Backlog	▪ List of requirements (stories) ranked according to customer's priority ▪ Points used to size individual stories	▪ List of requirements (stories) ranked according to customer's priority ▪ Use of points optional
Sprint Backlog	▪ List of stories to be completed during a given sprint ▪ Fixed for duration of the Sprint. ▪ Each story in sprint broken down into tasks	▪ List of stories to be completed during a given sprint ▪ Fixed for duration of the Sprint ▪ Breaking up stories into tasks optional
Sprint Tracking	▪ Daily Standup meetings where the team provides updates ▪ Burndown charts/graphs used to track progress	▪ No daily meeting ▪ Burndown charts/graphs used to track progress
Sprint Review	▪ Work done during sprint demoed ▪ Customer feedback gathered to be considered for next sprint	▪ Work done during sprint demoed ▪ Customer feedback gathered to be considered for next sprint ▪ Sprint Retrospective conducted after demo
Sprint Retrospective	▪ Completed sprint inspected to see if any improvements/changes to process is required	▪ No separate retrospective meeting.

Figure 3.1: Scrum for Teams vs. Scrum for Independent Programmers

3.3.3 Sprint Tracking

Once a sprint commences, Scrum allows for the progress made towards completion of each story to be tracked. At the end of each day the developer updates the number of hours he or she will have to put in order to complete a story (or task, if the stories were broken down). This data is logged in the sprint backlog and can be maintained using simple spreadsheets or defect tracking systems or any number of software products designed specifically for agile processes. An example of a sprint backlog with tracking information can be seen in Figure 4.3 in section 4.1.3. The number of hours left to finish each story is then added up and the resulting number is called the Burndown rate.

As a further step, the burndown rate can be represented graphically using the Burndown chart. This graphs maps the number of hours completed against the number of days left until the end of the sprint.

If points were used to size the stories, the relationship between the points awarded to a story can be correlated with the number of hours it took to complete it. This metric can be used to estimate the duration for similar stories in later Sprints. However, this metric only makes sense when it is gathered over a considerable period of time. The relationship between points and hours is usually very misleading during the first few sprints.

3.3.4 Sprint Review

After the Sprint ends, there is the Sprint Review, where the developer demos what has been built during the Sprint to the customer. This review provides the customer with

a chance to actually view the system in action and also to provide any feedback to the developer. This demo at the end of each sprint ensures that the customer is always in the loop and is aware of the progress made in the project. The sprint review also allows for the developer to reflect on the successes and failures during the course of the sprint and make any adjustments to the process that will enable the next sprint to be smoother and more productive.

The Sprint Review can continue into the Sprint Planning meeting. This will enable the customer to add any new features or enhancements, which he or she may deem necessary after seeing the software demoed, to the Product Backlog. This allows for the feedback from the demo to be instantly taken into account for the next Sprint.

The scaled down version of the Scrum process for independent programmers is shown below in Figure 3.2.

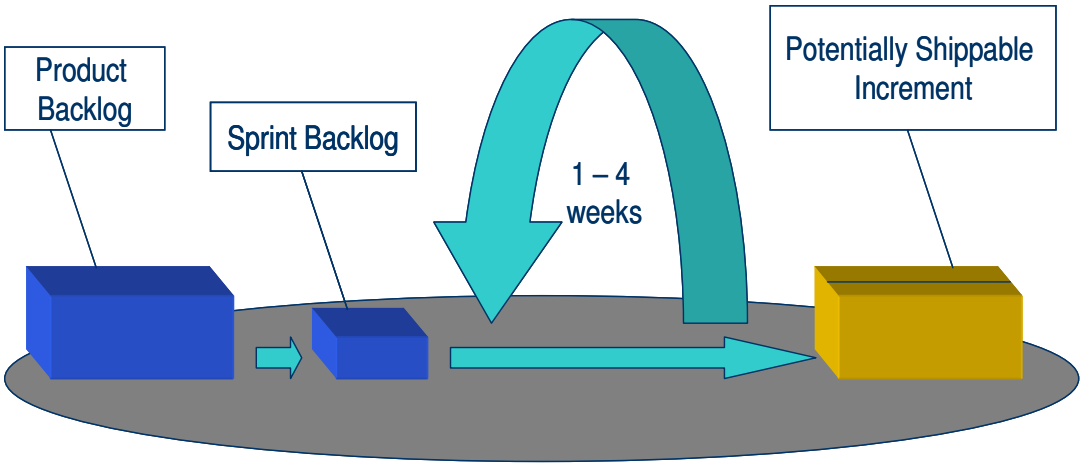


Figure 3.2: Scrum Process for Independent Programmers

4. SOLUTION VALIDATION

The scaled down version of the Scrum process described in section 3.3 was applied to two distinct projects. The first case study involved the author as the developer and his professor acting as the customer. The second case study was conducted under the auspices of Mentor Graphics Corporation with the author once again acting as the developer while Mentor Graphics acted as the customer.

4.1 Case Study 1

The scaled down Scrum process was used by the author during the development effort of the Online Simulation Tools for the Center for Advanced Vehicle Electronics (CAVE) at Auburn University. This center is dedicated to working with industry in developing and implementing new technologies for the packaging and manufacturing of electronics with special emphasis on the cost, harsh environment and reliability requirements of the vehicle industry. The CAVE Online Simulation Tools is designed to be used for trade-off studies, evaluation of What-IF scenarios, and development of system requirements. The CAVE software tool is an ongoing project that is being spearheaded by Dr. Pradeep Lall, a professor at the Mechanical Engineering department of Auburn University. The author, working as a Graduate Research Assistant for Dr. Lall applied the Scrum to development of the tool. Thus, Dr. Lall acted as the customer and

the author as the developer. The details about how the core practices of Scrum as they were applied in this case-study and the corresponding artifacts are explained in the following sections.

4.1.1 Case Study 1: Product Backlog

The Product Backlog for the CAVE tools was maintained by the customer. The Backlog was made up of feature requests by the sponsors of CAVE who would be the ultimate end-users. The backlog was constantly updated when additions of new functionality were deemed necessary and also with feature enhancements and bug fixes. The latter was mainly as a result of feedback gathered from previous Sprints.

4.1.2 Case Study 1: Sprint Planning

In this case study, each sprint was of duration of one week. During the Sprint Planning meeting at the start of the sprint, the developer and the customer discussed the stories that need to be completed in this sprint. The stories that were selected were the highest priority items from the Product Backlog. The list of stories derived from this meeting was usually a simple hand-written document as shown in Figure 4.1. This list was then transformed into a Sprint backlog. This Sprint backlog contained the stories themselves and the time estimated to complete each of them. The remaining columns which are empty in the beginning of the sprint show the day to day progress made towards completion of each story. A sample of such a sprint backlog at the beginning of the sprint is shown in Figure 4.2.

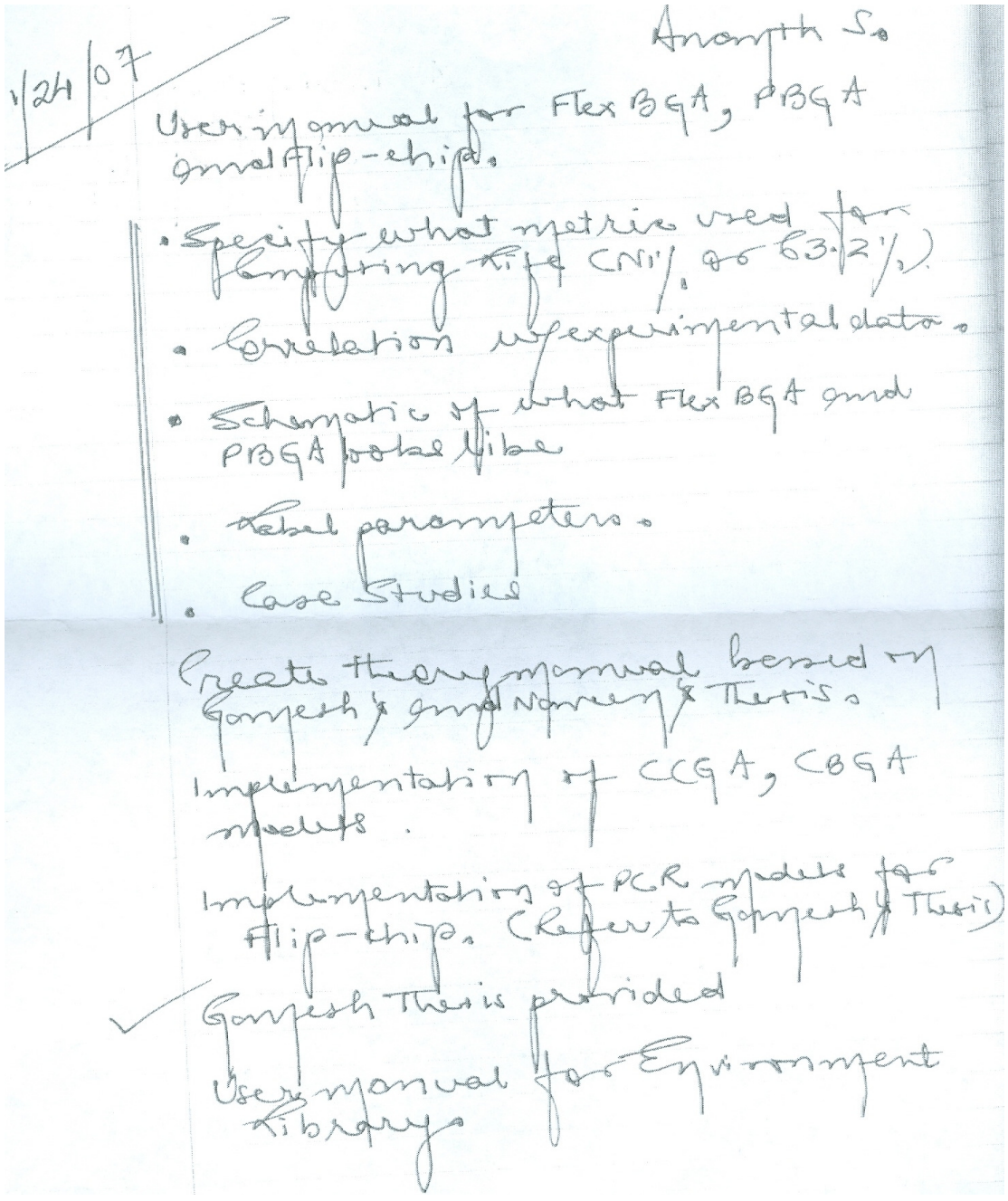


Figure 4.1: Scan of document showing Stories to be completed during the Sprint

		Hours of work to be completed						
Story	**Estimate (in hrs)**	**Day 1**	**Day 2**	**Day 3**	**Day 4**	**Day 5**	**Day 6**	**Day 7**
User Manual for Flex BGA, PBGA and Flip-Chip	16							
Implementation of CCGA and CBGA models	16							
Implementation of PCR models for Flip-Chip	8							
User Manual for Environment Library	16							
Total	**56**							

Figure 4.2: Sprint Backlog at the beginning of the sprint

The total hours listed under the column "Estimate" gives the total number of hours that the developer has committed to the sprint. As can be seen, for this sprint the developer committed a total of 56 hours over the period of seven days which is the length of the sprint.

4.1.3 Case Study 1: Sprint Tracking

The progress during the sprint was tracked simply by updating the Sprint backlog table as shown in Figure 4.2 with the number of hours remaining for a particular story to be completed. At the end of each day the total number of hours left for all stories was

added. This figure represents the burndown rate, that is, it represents the number of hours left till all the stories in the sprint are completed. Figure 4.3 shows the how Sprint Tracking worked during the course of the sprint.

		Hours of work to be completed						
Story	**Estimate (in hrs)**	**Day 1**	**Day 2**	**Day 3**	**Day 4**	**Day 5**	**Day 6**	**Day 7**
User Manual for Flex BGA, PBGA and Flip-Chip	16	16	16	16				
Implementation of CCGA and CBGA models	16	11	5	0				
Implementation of PCR models for Flip-Chip	8	8	8	8				
User Manual for Environment Library	16	13	13	9				
Total	**56**	48	42	33				

Figure 4.3: Updating the Sprint Backlog during the Sprint to enable tracking

4.1.4 Case Study 1: Sprint Review

At the end of the sprint the work done during the week was demoed to the customer. Any feedback and critique provided by the customer was added to the Product Backlog as feature requests, feature enhancements or bug fixes. These new stories were then considered for the next sprint depending on its priority. The Sprint Review then continued on to the Sprint Planning meeting for the next sprint.

4.1.5 Case Study 1: Results

The artifacts presented in the sections above are those collected from a single sprint. The same sequence was applied to numerous sprints. The data gathered from three sprints can be seen in Appendix A. The burndown chart for all three sprints shows the number of hours spent on each story during everyday of the sprint, from which the burndown rate was calculated. This data was the transformed to a burndown chart for each of the sprints. The burndown chart and the burndown graph clearly show the rate at which work is being completed, something that is essential for tracking progress. The effect of underestimating the effort needed for a story and changing requirements can be also been seen here. This is further explained in Appendix A.

4.2 Case Study 2

The second case-study into the effectiveness of the Scrum when used by individual programmers was conducted by applying it to the development and maintenance work done on the build system at Mentor Graphics Corporation.

It must be noted here that work on this system was performed by a team which used Scrum. However, the workload was distributed in such a way that each team member was able to work independently without the need for any input or collaboration with other team members. This meant that each team member can be construed as an independent programmer. This allowed the author to use Scrum in order to manage and track his work. Thus, the author acted as the developer while Mentor Graphics

Corporation took on the role of the customer. The artifacts collected from a single sprint of this case study as described in the following sections.

4.2.1 Case Study 2: Product Backlog

The Product Backlog for this study was maintained for the entire team. However, since both the team and the author took stories from the same Backlog, the Product Backlog for the whole team was considered as valid for the single developer as well.

In this case study, the Product Backlog was captured in a web-based tool. It was constantly updated with new feature requests, bud fixes, etc. by Mentor Graphics Corporation. A screenshot showing the product backlog is shown in Figure 4.4.

Figure 4.4: Screenshot of Product Backlog

4.2.2 Case Study 2: Sprint Planning

During the Sprint Planning, the stories that had the highest priority were discussed the developer committed to completing these stories by the end of the sprint which in this case study was of 3 weeks in duration.

In addition to the estimating the amount of time required to complete a story, every story was given a relative size estimate in points. Any one story was assumed to be a baseline and all the other stories were awarded points relative to this story. The stories taken in for one sprint along with their points is shown in Figure 4.5.

Story	**Points Estimate**
Resolve variable button on Tag SOD from profile page disappears	5.00
Numbers show up in Tag SOD from profile lists	3.00
When multiple tags, all but last one lost	5.00
Create tag profile doesn't allow variables	5.00
SQL error when editing a profile	3.00

Figure 4.5: Stories with their points in the Sprint.

Also in this case study each of the above stories were broken down into tasks. This provided more clarity into the each of stories. Each task for every story was then assigned hours depending on the estimation of how long it would take to complete the task. The sum total of the hours required to complete all tasks in a story is the amount of the time it would take to complete the story itself. The Sprint Backlog showing the

breakdown of stories into tasks and the corresponding time estimates is shown in Figure 4.6.

Backlog Item / Story	Tasks	Estimate
Numbers show up in Tag Sod from profile lists	Debug code	6.00
	Test changes	3.00
When multiple tags, all but last one lost	Research Solution	6.00
	Implement Solution	6.00
	Test changes	3.00
Create tag profile doesn't allow variables	Update existing code to allow for variables	12.00
	Description on the Tag field that it supports variable now	1.00
	Test changes	4.00
Resolve variable button on Tag Sod from profile page disappears	Determine all of the issues with the resolve variables button	6.00
	Fix issues with the resolve variables button (will be broken up after we determine the issues)	12.00
	Test Fixes	3.00
	InstallId field gets reset when I hit Show My BuildId checkbox.	1.00
	Save Profile doesn't save - Wrong GISA Tag(s) description.	1.00
mysql error when editing a profile	Research the code	12.00
	Code and debug	6.00
	Test the solution	6.00

Figure 4.6: Sprint Backlog with stories broken down into Tasks

4.2.3 Case Study 2: Sprint Tracking

The progress in the sprint was tracked in the same way as in the previous case study. At the end of each day, the hours for each task in the Sprint Backlog was updated to reflect the amount of work that still needs to be done in order to complete it. The

resulting burndown rate is represented graphically in Figure 4.7. Here, the x-axis corresponds to the days in the sprint while the y-axis corresponds to hours.

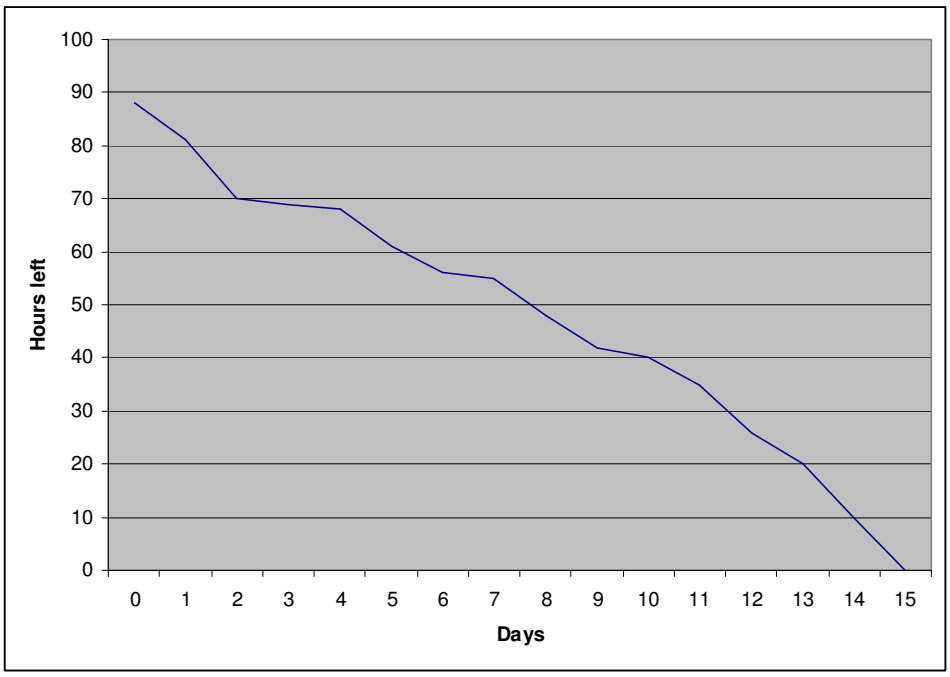


Figure 4.7 Burndown Graph with X-axis = Days and Y-axis = Hours

4.2.4 Case Study 2: Sprint Review

At the end of the sprint, the work was demoed to the customers and their feedback was solicited. This feedback was then added to the Product Backlog to be taken up during the next Sprint Planning meeting. Also, once the demo was done some time was set aside to conduct a retrospective of the process. Any problem that may have occurred was noted and ways to rectify them identified so that these changes may be incorporated into the next sprint.

4.2.5 Case Study 2: Results

The data gathered from three sprints of this case study can be seen in Appendix B. This data shows the stories taken on during each sprint, the breakdown of the stories into tasks and hours assigned to each task. Once the sprint commenced, the hours left at the end of each day was noted in a burndown chart which was then transformed into a burndown graph. It was found that breaking down stories into tasks helps in improved estimations for each task and correspondingly each story. This conclusion can be arrived at by observing the task breakdowns tables and the burndown graphs of Appendix B. In the first iteration shown, the stories were poorly broken down into tasks which meant that the estimate to complete each task was underestimated. The result of this underestimation can be seen in the burndown graph which shows the burndown rate going up rather than down. However, in the later iterations shown, the task breakdown improved. This, in turn, led to better estimates which resulted in a smoother burndown rate.

5. CONCLUSION AND FUTURE WORK

In this work, we first looked at the importance of adhering to a process whether working as a team or independently. We then saw the goals that processes aim for and the principles that agile processes, in particular, adhere to in order to reach these goals. This was followed by an overview of the two most popular agile processes, Scrum and XP.

We then looked at how the Scrum process can be adapted to be used a programmer working independently. This scaled down version of Scrum was applied in two case-studies. The first case-study was conducted in an academic and research environment while the second was conducted within the software industry. The data gathered from the two case-studies shows that Scrum can be successfully by individual programmers. It shows how requirements of a project can be broken down into concise stories, a group of which can then be incorporated into a sprint to form the Sprint Backlog. This ensures that the developer carefully estimates the time available and takes on only as much work as can be completed in that time. Also, the results from the second case study show how breaking down stories into tasks can improve the estimates for time required to complete the task and, thus, the story. The tracking of progress once is an important goal for a process and this goal is met through the use of burndown charts and graphs in every sprint. The demo conducted at the end of every sprint allowed the customers' to gain visibility into the progress of the project and it also allowed to

developer to gather feedback at regular intervals. Scrum, being an agile process, had very low overhead, i.e. it did not distract the developer from coding. The developer only had to spend a few minutes gathering the data necessary for the burndown charts and graphs. The process was also highly flexible, allowing the developer to make key decisions while providing a basic structure to follow during the development cycle.

The area which calls for further research is the necessity to arrive at a good correlation between the size of a story (as described by points) and the time it would take to complete it. It was observed during the first few sprints that the size of a story seemed to have no relation with the time it took to finish it. However, as the developer gained more experience with the process, the estimations showed improvements. Using Scrum over an extended period of time and analyzing the data from all the sprints during this time period will result in a useful metric to estimate the time required for stories similar to the ones in previous iterations.

This study and its results clearly show that consistent use of the Scrum will definitely help individual programmers in their development work, and thus, deliver high-quality software.

BIBLIOGRAPHY

"CONTRIBUTOR MELONFIRE", Understanding the pros and cons of the Waterfall Model of software development, http://articles.techrepublic.com.com/5100-10878_11-6118423.html, (22 September 2006).

AMBLER Scott W., Examining the Agile Manifesto, http://www.ambysoft.com/essays/agileManifesto.html, Last Accessed: August 19, 2006.

BECK Kent, BEEDLE Mike, BENNEKUM Arie van, et al., Manifesto for Agile Software Development, http://agilemanifesto.org/, 2001.

BECK Kent, Embracing Change with Extreme Programming, *IEEE Xplore*, http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00796139, (October 1999).

BECK Kent, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

COHN Mike, The Scrum Development Process, http://www.mountaingoatsoftware.com/scrum, 2005.

DAVIDSON Michelle, Survey: Agile Interest High, But Waterfall Still Used by Many, http://searchsoftwarequality.techtarget.com/news/article/0,289142,sid92_gci1318992,00.html, (27 June 2008).

DEEMER Pete, BENEFIELD Gabrielle, The Scrum Primer: An Introduction to Agile Project Management with Scrum, http://www.rallydev.com/documents/scrumprimer.pdf, 2007.

FOWLER Martin, HIGHSMITH Jim, The Agile Manifesto, http://www.ddj.com/architect/ 184414755, (1 August 2001).

HIGHSMITH Jim, Extreme Programming, http://rockfish-cs.cs.unc.edu/COMP290-agile/xp-highsmith.pdf, (February 2000).

JOHNSON Philip, KOU Hongbing, AGUSTIN Joy, et al., Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined, http://csdl.ics.hawaii.edu/techreports/02-07/02-07.pdf, 2003.

MILLER Roy, Demystifying Extreme Programming: "XP distilled" Revisited, *IBM developerWorks*, http://www.ibm.com/developerworks/java/library/j-xp0813/, (13 August 2002).

PAREKH Nilesh, The Waterfall Model Explained, http://www.buzzle.com/editorials/1-5-2005-63768.asp, (5 January 2005).

POPPENDIECK Mary, POPPENDIECK Tom, *Lean Software Development: An Agile Toolkit*, Addison-Wesley, 2003.

SCHWABER Ken, BEEDLE Mike, *Agile Software Development with Scrum*, Prentice Hall 2001.

SUTHERLAND Jeff, SCHWABER Ken, The Scrum Papers: Nuts, Bolts and Origins of an Agile Process, http://www.crisp.se/scrum/books/ScrumPapers20070424.pdf, (22 March 2007).

TAKEUCHI Hirotaka, NONAKA Ikujiro, The New New Product Development Game, *Harvard Business Review*, (1 January 1986).

TYRRELL Sebastian, The Many Dimensions of the Software Process, *Crossroads: The ACM Student Magazine*, http://www.acm.org/crossroads/xrds6-4/software.html, (2000).

WELLS Donovan, Extreme Programming: A Gentle Introduction, http://www.extremeprogramming.org/, 2000.

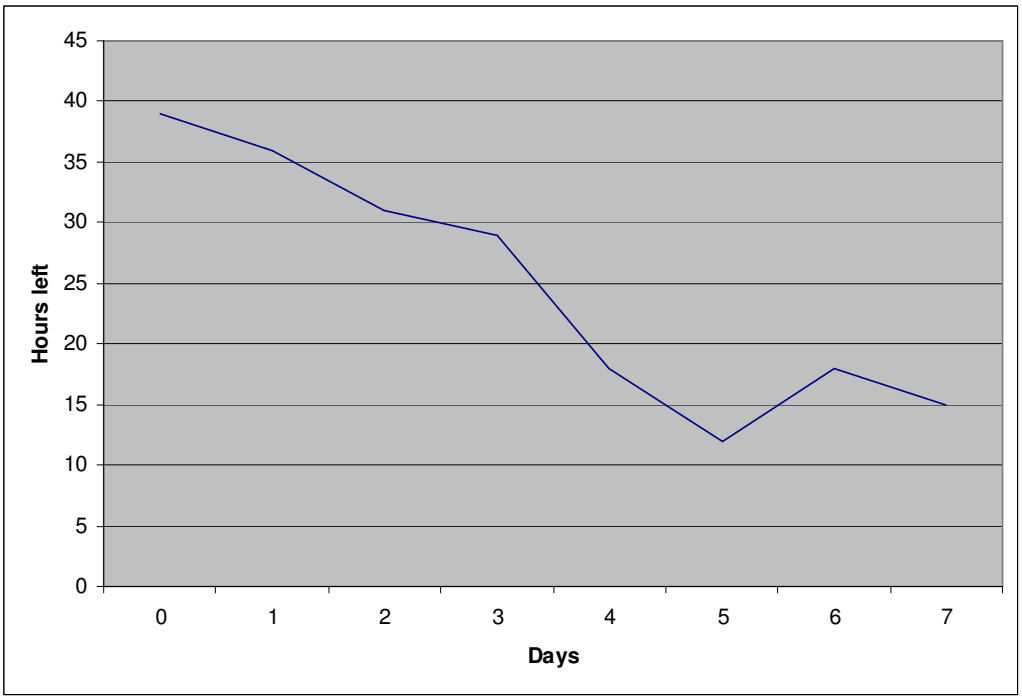
APPENDIX A

Case Study 1

Iteration # I

		Hours of work to be completed						
Story	**Estimate (in hrs)**	**Day 1**	**Day 2**	**Day 3**	**Day 4**	**Day 5**	**Day 6**	**Day 7**
Change Layout of the main screen as discussed	3	0	0	0	0	0	0	0
Mechanism for creation of auxiliary local database; 'Load' command should allow user to upload CSV data	24	24	24	24	18	12	18	15
Environment Library menu must be visible from all pages; not just the first webpage	6	6	1	0	0	0	0	0
Semiconductor Database incomplete; Populate Semiconductor database	6	6	6	5	0	0	0	0
Total	**39**	36	31	29	18	12	18	15

Iteration # I: Burndown Chart



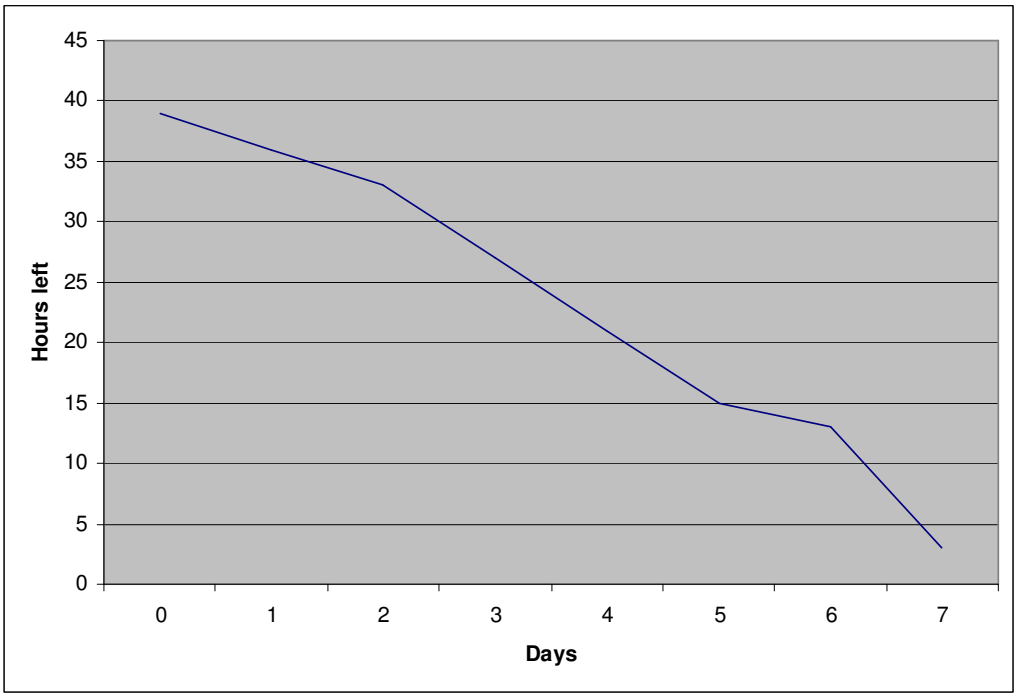
Iteration # I: Burndown Graph

The effect of underestimating the time required to complete a story can clearly be seen in the results of this iteration. When it was realized that completing a story would require a lot more time, the number of hours was correspondingly increased. This is reflected in the burndown graph where the line goes up. This particular story ("Mechanism for creation of auxiliary local database") was not completed during this sprint and was carried over to the next sprint, the results of which can be seen in the next section.

Iteration # II

		Hours of work to be completed						
Story	**Estimate (in hrs)**	**Day 1**	**Day 2**	**Day 3**	**Day 4**	**Day 5**	**Day 6**	**Day 7**
Bug: Menu should appear without offset irrespective of screen resolution	3	0	0	0	0	0	0	0
Populate Tools menu in Environment Library with Add, Delete and Modify features	12	12	12	12	12	12	10	0
Restrict access to Add, Delete and Modify features to Super-users only	3	3	3	3	3	3	3	3
Complete field set of properties for all materials in database. Include mechanical, thermal and electrical properties	6	6	3	3	0	0	0	0
Mechanism for creation of auxiliary local database; Provide Excel template and convert it to CSV file	15	15	15	9	6	0	0	0
Total	**39**	36	33	27	21	15	13	3

Iteration # II: Burndown Chart



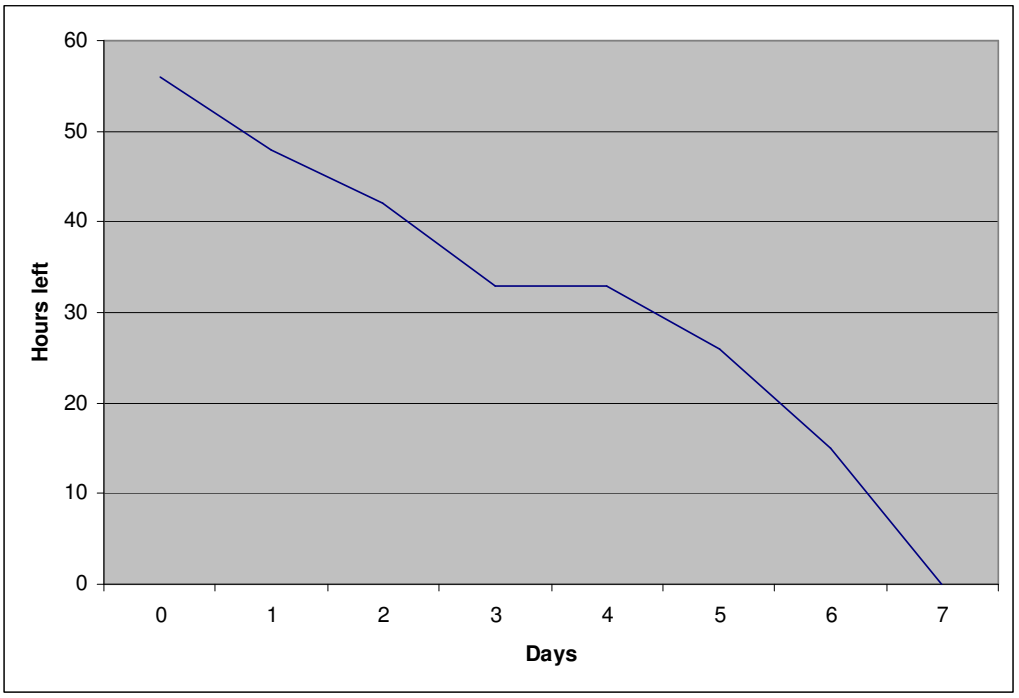
Iteration # II: Burndown Graph

This sprint had a story that was incomplete in the previous sprint ("Mechanism for creation of auxiliary local database"). The requirement that was described by this story also changed which is reflected description provided for the story in the burndown chart. This shows that Scrum, even when used by independent programmers is capable of handling a change in requirements without adversely impacting the schedule of the project.

Iteration # III

Story	Estimate (in hrs)	Hours of work to be completed						
		Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
User Manual for Flex BGA, PBGA and Flip-Chip	16	16	16	16	16	16	6	0
Implementation of CCGA and CBGA models	16	11	5	0	0	0	0	0
Implementation of PCR models for Flip-Chip	8	8	8	8	8	1	0	0
User Manual for Environment Library	16	13	13	9	9	9	9	0
Total	**56**	48	42	33	33	26	15	0

Iteration # III: Burndown Chart



Iteration # III: Burndown Graph

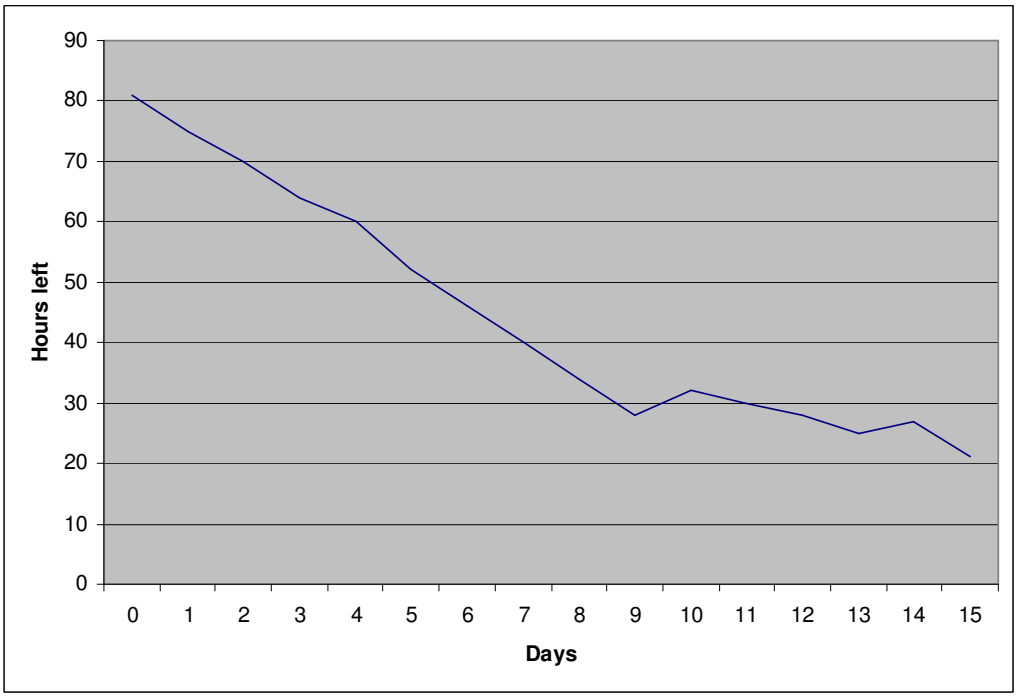
APPENDIX B

Case Study 2

Iteration # I

Story (Points)	Tasks	Estimate
Create Database Schema for Promote Profile (1)	Create Database Schema	3.00
Populate Database for promote Profile (3)	Populate Database Manually	3.00
	Link database to existing Managers Page	12.00
Save Promote Profile (2)	Add 'Save Profile' Button	3.00
	Handle 'Save Profile' action	8.00
Show Loaded Promote Profile (3)	Modify page to handle the situation when PromoteProfileID is passed in	12.00
Edit Promote Profile (2)	Add 'Edit Profile' Button – The Save Profile button must change to Edit Profile when PromoteProfileID is available	4.00
	Handle 'Edit Profile' action	8.00
Add capability to handle publishes/promotes without units list (5)	Modify 'Publish' Page	12.00
	Modify 'Promote' Page	12.00
Document Functionality (1)	Write Documentation for Promote Profiles	4.00

Iteration # I: Task Breakdown



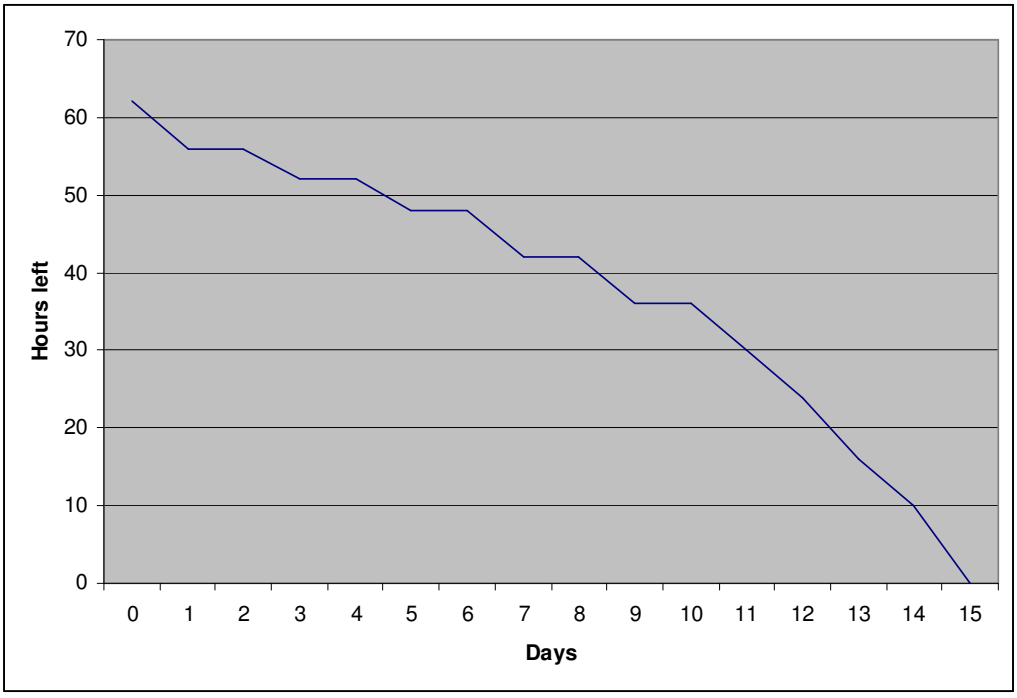
Iteration # I: Burndown Graph

This sprint was one of the first ones conducted as part of this case study. Being inexperienced at task breakdowns at this point, these weren't very effective. This in turn affected the estimate for each of the tasks which meant that some of them were underestimated. This is reflected in the burndown graph with the spikes in the line showing the point during the sprint when the underestimations were discovered.

Iteration # II

Story (Points)	Tasks	Estimate
Emails for tags need to include information such as Build_ID, tag, and the original attach to (8)	Research the Code	12.00
	Modify function call to pass Build_ID information.	12.00
	Modify function to accept Build_ID information.	3.00
	Modify Email function to include the Build_ID, Tag etc in the Email.	6.00
	Test on local system	18.00
	Move to Development system for QA testing	2.00
	Comment code to flag changes	3.00
	Review acceptance throughout process	6.00

Iteration # II: Task Breakdown



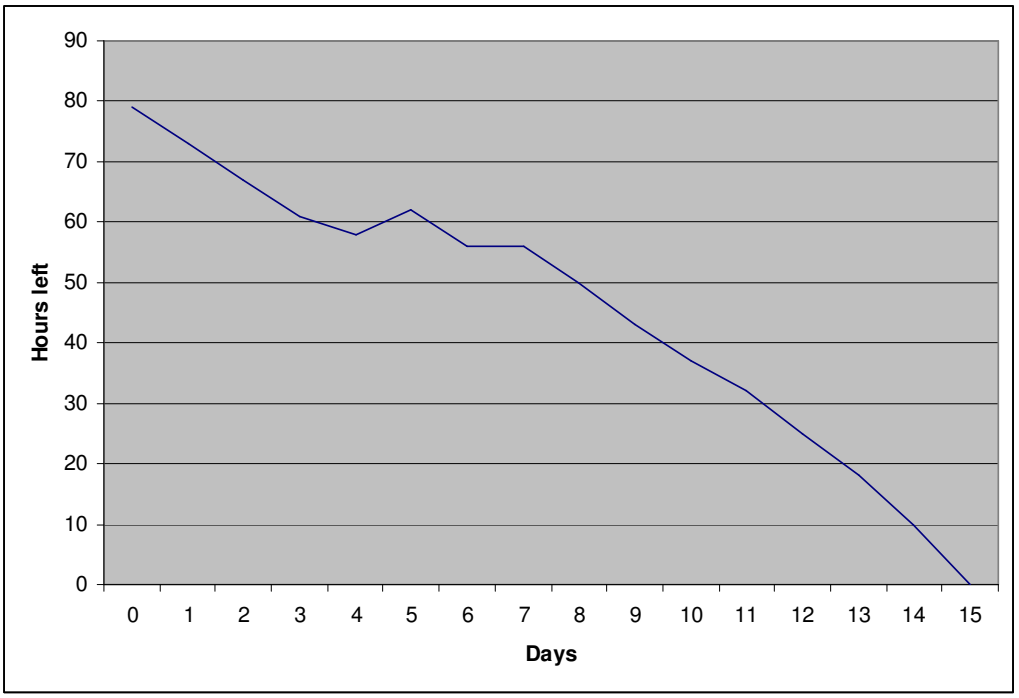
Iteration # II: Burndown Graph

The number of hours for this sprint available for this sprint was is considerable lower for this sprint than others though the duration of the sprint was still the same. This was because the developer took part in mandatory training sessions on alternate days which reduced the time that could be set aside for development work. The points in the burndown graph where the line flattens shows the days in which no work was done.

Iteration # III

Story (Points)	Tasks	Estimate
Edit Install Profile from Managers Page (15)	Add Edit Install Profile option to the 'Select Command' drop-down in managers page	4.00
	Create GUI to for Edit Install Profile page for clean install type	12.00
	Add capability to handle tasks & machines option	18.00
	Add functionality to save edited profile.	6.00
	Test Edit functionality for clean profiles	6.00
	Create GUI for Update/Patch install profiles	12.00
	Add functionality to save edited Update/Patch install profiles.	6.00
	Test Edit functionality for Update/Patch profiles	6.00
	Add 'Reset' functionality	6.00
	Test 'Reset' functionality	3.00

Iteration # III: Task Breakdown



Iteration # III: Burndown Graph

The data from this iteration shows a significant improvement in task breakdowns. This resulted in better time estimates for each task and enabled the developer to complete the one story that was part of the sprint to be completed successfully.