NEED A NERD – Adapting PCSE (Practitioner Centered Software Engineering) to develop a Web Application

by

Prabhu Selvaraj

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
December 8, 2012

Keywords: software process, PCSE, Django,
timelog, changelog

Approved by

David A. Umphress, Chair, Associate Professor of Computer Science & Software Engineering
James H. Cross II, Professor of Computer Science & Software Engineering
Hari Narayanan, Professor of Computer Science & Software Engineering

Abstract

Practitioner Centered Software Engineering (PCSE) is the most recent incarnation of Auburn University's personal self-improvement process for helping software engineers control, manage, and improve the way they work. It helps them make accurate plans, consistently meet commitments, improve QPPC (Quality, Predictability, Productivity and Customer satisfaction), and deliver high-quality products. PCSE is a tailored collection of different elements from various software processes such as Personal Software Process (PSP), Team Software Process (TSP), Extreme Programming (XP), Feature Driven Development (FDD), SCRUM, Rational Unified Process (RUP) etc. PCSE was developed by Dr. David A. Umphress, Department of Computer Science & Software Engineering, Auburn University, in an effort to bring engineering discipline to one-person software development teams.

The objective of this thesis is to apply PCSE to develop a web based application "**NEED-A-NERD**" using Django. Django is a web application framework written in Python. PCSE is a one-person team process and it has been used in the past for the development of conventional applications. It would be a challenge to follow PCSE in a web development environment (to develop a web based application), as it would require a few modifications in different phases in the software process and in producing the artifacts during the different stages. This thesis would give a good measure of how efficiently PCSE could be used in the development of web applications. On successful completion, the web application is expected to be used by the employees and students of the Department of Computer Science and Software Engineering at Auburn University for their on-campus job pursuit.

Acknowledgments

I take this opportunity to thank all those who helped and guided me throughout this research. I consider it a special privilege to convey my prodigious and everlasting thanks to my advisory committee chair, Dr. David A. Umphress, Computer Science & Software Engineering department, Auburn University for all the advice, guidance and support given to me right from the beginning of the thesis. I express my deep sense of gratitude to Dr. James H. Cross II, Computer Science & Software Engineering department, Auburn University, and Dr. Hari Narayanan, Computer Science and Software Engineering department, Auburn University, for their valuable advice, insight, and critical reviews provided throughout my thesis work.

My special thanks to our PCSE research team members, Asmae Mesbahi El Aouame (PhD student), Jackie Hundley (Instructor), Russell Thackston (PhD student), Susan Hammond (PhD student), Brad Dennis(PhD student), William Symon(MS student), Michael Zekoff(MS student) and Yasmeen Rawajfih (PhD Student), Computer Science and Software Engineering department, Auburn University, for their support.

With immense pleasure and satisfaction, I express my sincere thanks to all my family members and friends for their kind help and unstinted cooperation and companionship during the thesis work.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

PCSE	Practice Centered Software Engineering
PSP	Personal Software Process
TSP	Team Software Process
XP	Extreme Programming
FDD	Feature Driven Development
RUP	Rational Unified Process
QPPC	Quality, Productivity, Predictability, Customer Satisfaction
LOC	Line of Code
CRC	Class Responsibility Collaborator
LOCr	Raw lines of code
LOCp	Planned lines of code
LOCa	Actual lines of code
Ep	Planned Effort
Ea	Actual Effort
LPI	Lower Prediction Interval
UPI	Upper Prediction Interval
EV	Earned Value
PV	Planned Value
TDD	Test Driven Development

1. Problem Description

1.1 Software Process

Software process is the set of tasks needed to produce quality software [1]. It helps developers make accurate plans, consistently meet commitments, improve QPPC (Quality, Predictability, Productivity and Customer satisfaction), and deliver high-quality products. It is a structured framework of forms, guidelines, activities, and procedures for developing software. A growing body of software development organizations implements process methodologies, employing methodologies which are intended to improve software quality, such as Personal Software Process (PSP), Team Software Process (TSP), Extreme Programming (XP), Feature Driven Development (FDD), SCRUM, Rational Unified Process (RUP), etc.

Each of these methodologies describes approaches to a variety of tasks or activities that take place during software development. However, there are no restrictions such that one has to follow a particular methodology while practicing software process. It is generally based on the type of the industry and the nature of the project, apart from other factors such as the budget, technology used, resources, etc.

The explosive growth of one-person development efforts for mobile devices -- as exemplified by the Android and iPhone markets -- suggests there is an audience for individualized process. The preponderance of processes today is for multi-person efforts. The Personal Software Process (PSP) is the only published process that addresses development at the one-person level; but, its data collection requirements have proven encumbering to the point where it has fallen out of use.

1.2 Need for Light-weight software process

There are many software development methodologies in use today and the list grows daily. Many developers have their own customized methodology for developing their software, while others use off-

the-shelf commercial methodologies. The following factors play an important role in selecting a methodology: budget, team size, project criticality, technology used, documentation, training, tool and techniques etc.

The traditional project methodologies that many developers use are considered to be bureaucratic or predictive in nature, and they have resulted in many unsuccessful projects [2]. They can be so tedious that the whole pace of design, development and deployment actually slows down.

A lightweight software process is a software development methodology that has only a few rules and practices or ones which are easy to follow. It emphasizes the need to deal with changes in requirements and changes in environment or technology by being flexible and adaptive. In lightweight software process, after each build or iteration, the developer adjusts the process to correct issues on the project, forming an improvement cycle throughout the project.

The following are the major advantages of lightweight methodologies [2].

1.) They accommodate change well.

2.) They are people-oriented rather than process-oriented. They tend to work with people rather than against them.

3.) They are complemented by the use of dynamic checklists.

4.) They focus more on software than on documents.

The number of frequent cycles in the lightweight methodologies also provides more opportunities for developers to review the project definition and redefine it for new business needs. It has room to add new requirements and change the requirements list, adjusting priorities accordingly. Another benefit of the lightweight methodologies is their focus on producing value-added releases and addressing architectural risk early in the project, which would be difficult with a heavyweight methodology.

1.3 Practice Centered Software Engineering (PCSE)

The Practice Centered Software Engineering (PCSE) is a lightweight software engineering process developed by Dr. David A. Umphress, Department of Computer Science & Software

Engineering, Auburn University [1]. It is the most recent rendition of Auburn University's personal self-improvement process that helps software engineers control, manage, and improve the way they work. Using common industry practices, PCSE describes the following activities/phases that are performed within the software development process: Analysis, Architecture, Project plan, Iteration plan, Construction, Review, Refactoring, Integration, Post mortem and Code complete. Each of these activities is associated with a particular artifact such as the Operational specification, Scenario-Component map, Iteration map, Conceptual design, Size matrix, Time log, Change log, Iteration map, Burn-down chart, Calendar, Diary etc. The flow of the activities and the use of the artifacts are detailed in the forthcoming chapters.

1.4 PCSE in a web development environment

PCSE has been used to develop a variety of applications in the past, but all of these applications have been conventional applications rather than web-based applications. This thesis aims to apply PCSE to develop a web based application and measure how efficiently PCSE could be used in the development of web based applications by one-person teams. In the past, PCSE has been used to develop applications where the software components developed would fall into one of the following four categories:

- Logic,
- Data,
- Calculation and
- I/O

So, when we develop any kind of application, the software components we develop should belong to one of the above categories. But in case of our web based application, the components we are dealing with – that is, the components that implement the Model-View-Controller architecture used by web applications -- do not quite fit into these standard component types. We need to define new component categories, which, in turn alter the way in which PCSE deals with the different phases of the development

process including architecture, planning, and construction. In short, artifacts produced for web-based applications are not code segments written in a common language and style. Instead, a web application consists of code, HTML, XML, JSON, and other dissimilar artifacts. This thesis gives an insight into how to modify PCSE so that it can be used to develop nonhomogeneous software artifacts, focusing specifically on adaptations for web application software development.

2. Literature Review

The important literature relevant to single person process and web application development methodologies is as follows:

2.1 Agile Software Development[5]

Agile software development is a group of software development methods based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. It promotes adaptive planning, evolutionary development and delivery, a time-boxed iterative approach, and encourages rapid and flexible response to change[5].

The principles of Agile Software Development are: [5]

- Customer satisfaction by rapid delivery of useful software
- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Close, daily co-operation between business people and developers
- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity- The art of maximizing the amount of work not done - is essential
- Self-organizing teams
- Regular adaptation to changing circumstances.

2.2 The Personal Software Process (PSP):

The Personal Software Process (PSP) [6] is a structured software development process for a single developer, created by Watts Humphrey in Software Engineering Institute. PSP aims to provide software engineers with disciplined methods for improving personal software development processes that help developers produce zero-defect, quality products on schedule. One of the core aspects of the PSP is using historical data to analyze and improve process performance[6].

The PSP helps software engineers to:

- Improve their estimating and planning skills.
- Make commitments they can keep.
- Manage the quality of their projects.
- Reduce the number of defects in their work.

The following are the advantages of PSP [6]:

- It helps the developer understand his performance.
- It helps the developer to manage his work.
- It helps to plan and manage the quality of products produced.
- It helps to make detailed plans and precisely measure and report the status.
- It helps to judge the accuracy of your estimates and plans.
- It helps to communicate precisely with users, other developers, managers, and customers about the work.
- It helps to identify the process steps that cause the most trouble.
- It helps to improve the personal performance.
- It will simplify training and facilitate personal mobility.
- Well-defined process definitions can be reused or modified to make new and improved processes.

Although PSP is aimed at helping single developers to develop software, PSP cannot be used to develop the web application in context. First, PSP is encumbering due its requirements for data collection.

Another major reason is the historical data analysis which is the core aspect of the planning and estimation process used by PSP. Since we are dealing with a web based application, we don't have any historical data to work with. PSP is a predictive process which predicts the size and effort of the project based on the historical data. But we need an agile approach which is adaptive, so we can adapt the process and apply it into a new environment.

2.3 A Theoretical Agile Process Framework for Web Applications Development in Small Software Firms[8]

The software development methodologies available today are viewed by many as outdated and inappropriate for rapid development and web application development. Most of the web application development methodologies used are extensions of standard software engineering methodologies. The usual iterative waterfall model is too rigid an approach to developing web applications. The waterfall model process was perfect for developing a file maintenance program for mainframes but a far too restrictive process for developing web applications. Web application development needs to be an iterative process and most agree that a spiral approach is the best. Web application development is certainly component-oriented and the process that should be used needs to be object-oriented.

Agile processes are intended to support early and quick production of working code. This is achieved by structuring the development process into iterations, where an iteration focuses on delivering working code and other artifacts that provide value. Sometimes developers think that code is the only deliverable that matters and ignore the role of design models and documentation. Agile process critics point out that emphasis on code could lead to corporate memory loss because there is little emphasis on producing good documentation to support software creation[8].

Designing web applications is not an easy task. It is not just using HTML or web development software such as Front Page or Dreamweaver and few images, menus and hyperlinked documents, etc. Web application development process seems very complex and it has a lot of challenging requirements. It

needs more of planning, web architecture, system design, testing, quality assurance, performance, evaluation, continual update and maintenance of the system as the requirements.

Many practitioners in the field of web engineering have commented on the lack of suitable software engineering processes that can be used to build web applications. If a web application development process needs to be successful, then it has to address the following issues [8].

- Short development life-cycle times
- Delivery of tailored solutions
- Multidisciplinary development teams
- Small development teams working in parallel on similar tasks
- Analysis and Evaluation
- Requirements and Testing
- Maintenance.

2.4 PCSE Lifecycle

The Practice Centered Software Engineering (PCSE) is the most recent version of Auburn University's personal self-improvement process for helping software engineers control, manage, and improve the way they work. PCSE was developed to introduce a light-weight software process and practice the major software process tools that are widely recognized and used in the industry.

This section explains the PCSE life cycle. The different activities involved in PCSE are explained in detail. Each artifact that belongs to a particular activity is explained with sample examples.

All information, content, explanations and examples referred to in this chapter originated from the following sources:

- Dr. David A. Umphress personal communication.
- “Software Process” (COMP 6700) class presentations, videos.
- Interaction with the PCSE research team.

Although PCSE is well defined, it undergoes continuous improvement. There is considerable room for enhancing PCSE by introducing new techniques and tools. The current PCSE research team is working along with Dr. Umphress to identify the scope for improvement.

The following figure illustrates the different activities and their flow involved in PCSE:

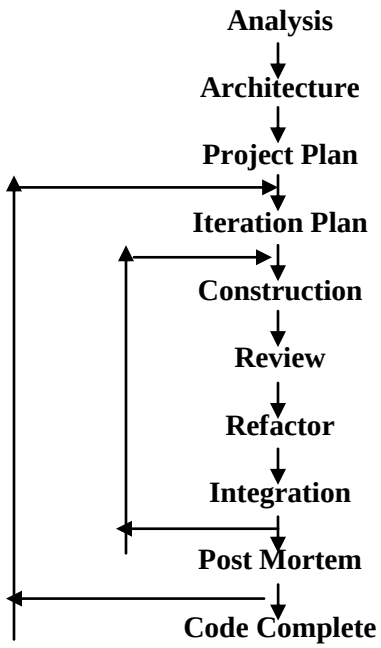


Figure 1: PCSE Life Cycle

Each of the different activities and their corresponding artifacts is described below:

2.4.1 Analysis:

Analysis is the process of breaking a complex topic into smaller parts to gain a better understanding of it. This stage includes identifying the desired behavior of the system. The outcome of the analysis phase is an operational specification which is a list of scenarios, where each scenario is a representative collection of desired behaviors expected of the software component under consideration [1].

There are two major types of operational specification:

1.) Interface operational specification – illustrates component-to-component interaction

2.) User operational specification – illustrates user-to-component interaction

The following are examples of interface and user operational specifications:

Tuple #	Type	Actor	Event/Actor response description	Example
1	Event	Test Driver	call average with valid list	Statistics.average([1,2,3,4,5])
2	Response	Blackbox	returns average of list	3.0
3	Event	Test Driver	call median with valid list containing odd number of values	Statistics.median([1,2,3])
4	Response	Blackbox	returns median of list	2.0
5	Event	Test Driver	call median with valid list containing even number of values	Statistics.median([1,2,3,4])
6	Response	Blackbox	returns median of list	3.5
7	Event	Test Driver	call stdev with valid list	Statistics.stdev(list)
8	Response	Blackbox	returns standard deviation of list	1.29

Table 1: Interface Operational Scenario – nominal [1]

Tuple #	Type	Actor	Event/Actor response description	Example
1	Event	Test Driver	call stdev with empty list	Statistics.stdev([])
2	Response	Blackbox	raises exception	Runtime Error
3	Event	Test Driver	call stdev with one-element list	Statistics.stdev([5])
4	Response	Blackbox	raises exception	Runtime Error

Table 2: Interface Operational Scenario – anomalous [1]

Tuple #	Type	Actor	Event/Actor response description	Example
1	Event	User	Start application	
2	Response	Blackbox	"Enter filename or stop"	
3	Event	User	User enters file name	assignment1test1.txt

4	Response	Blackbox	Display number of values in the file	10
5	Response	Blackbox	Display average of values in the file	42
6	Response	Blackbox	Display the median of the values in file	39.5
7	Response	Blackbox	Display the stdev of the values in the file	2.7
8	Response	Blackbox	"Enter filename or stop"	
9	Event	User	User enters "stop"	
10	Response	Blackbox	"Program terminated"	

Table 3: User Operational Scenario – nominal [1]

2.4.2 Architecture:

The main motive of this phase is to develop a high-level design and to identify major components sufficient to begin scoping the effort required by the project. This phase focuses on allocating functionality. During this phase, the output of the analysis phase is used to partition the system into conceptual components using CRC (Class Responsibility Collaborator) cards [1]. This activity entails identifying parts within the black box at a limited level of abstraction, i.e., identifying major components called proxies, usually objects and functions. This provides the basis for estimation, task identification and scheduling. CRC cards can be visualized as a textual/tabular version of UML class diagrams.

A CRC card contains the following elements:

a.) **Proxy Name:** denotes the name of the class or function

b.) **Design Approach:** either Object Oriented (or) Functional

c.) **Super class:** denotes a parent proxy

d.) **Component Type:** either Logic (or) Calculation (or) Data (or) Input/output

e.) **Collaborators:** represents other components which have a relationship with this component

f.) **Operations:** represents the functionalities

The following are examples of CRC cards:

1.) *Proxy Name:* print_proxy_history

Design Approach: Functional

Parent Proxy:

Attributes (optional):

Component Type: I/O

Collaborators: ProxyHistory

Operations: print_proxy_history

2.) *Proxy Name:* ProxyHistory

Design Approach: Object-oriented

Parent Proxy:

Attributes (optional):

Component Type: Logic

Collaborators: SourceFile

Operations: initialize, generate_project_history, generate_proxy_history calculate_average_size

3.) *Proxy Name:* SourceFile

Design Approach: Object-oriented

Parent Proxy:

Attributes (optional):

Component Type: Calculation

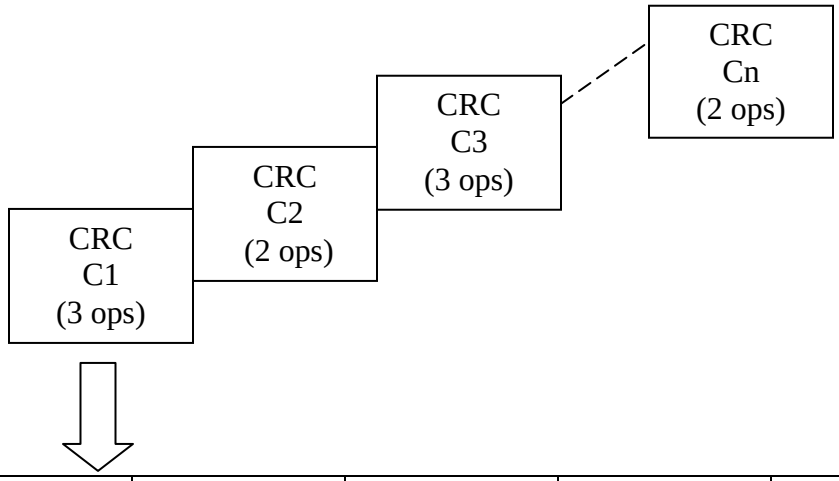
Collaborators:

Operations: initialize, count_lines, validate_count, validate_blocks, file_name, count_proxyblock, validate_proxyblocks,

count_proxymethod, get_proxytype, each_proxy

After identifying the major components (CRC cards) in the Architecture phase and the scenarios in the Analysis phase, the developer comes up with a Scenario-Component Map. This allows the developer to map each operation in the component to their respective scenarios.

The following table illustrates the scenario-component map:



	Component1	Component 2	Component 3		Component n
Scenario 1	Op 1a Op 1b	Op 2b			
Scenario 2		Op 2a Op 2b			
Scenario 3	Op 1a Op 1c		Op 3a Op 3b Op 3c		
Scenario n					Op na Op nb

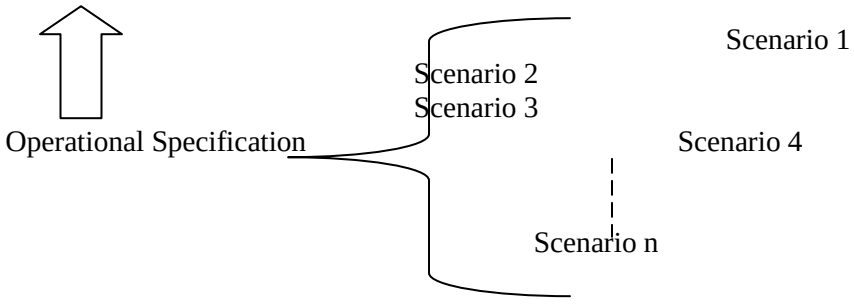


Figure 2: Scenario – Component map

2.4.3 Project Plan:

The main motive of this phase is to estimate the overall effort. In PCSE, estimation is done by Proxy-Based Estimation setup, where it relies on historical project data (i.e. both project history and proxy history). The Project and Proxy history consists of the following data:

Project History:

1.) LOCr – Raw lines of code
2.) LOCp – Planned lines of code
3.) LOCa – Actual lines of code
4.) Ep – Planned duration
5.) Ea – Actual duration

Proxy history:

1.) Proxy Name
2.) Total LOC
3.) Methods
4.) Type
5.) Size
6.) LOC/method
7.) Ln(LOC/Method)

First, a size matrix is constructed based on the historical project data, which results in calculating the size ranges. The size ranges are categorized into Very Small (VS), Small (S), Medium (M), Large (L), and Very Large (VL). The size matrix is used to derive the raw lines of code (LOCr) for the new project, from which the planned lines of code (LOCp) and the planned estimate time (Ep) can be calculated.The following section explains in detail the calculation of the size matrix and deriving the LOCp and Ep for a new project.

Let’s assume we have the following project and the proxy history.

Project history:

Project Name	**LOCr**	**LOCp**	**LOCa**	**Ep**	**Ea**
Project 1	30	30	48	145	249
Project 2	45	45	168	190	419
Project 3	65	65	146	290	438
Project 4	273	339	274	627	577
Project 5	203	270	182	589	513

Table 4: Example project history

Proxy history:

Proxy Name	**Total LOC**	**Methods**	**Type**	**LOC/meth**	**ln(LOC/meth)**
SourceFile	48	4	Calculation	12.00	2.48
SourceFile	211	10	Calculation	21.10	3.05
print_each_proxy	20	1	I/O	20.00	3.00
ProxyHistory	120	4	Logic	30.00	3.40
print_proxy_history	25	1	I/O	25.00	3.22
print_schedule	45	1	I/O	45.00	3.81
Schedule	178	3	Calculation	59.33	4.08
TaskList	19	3	Logic	6.33	1.85
Task	12	3	Data	4.00	1.39
Calendar	20	3	Data	6.67	1.90
print_tcurve	43	1	I/O	43.00	3.76

TCurve	139	7	Calculation	19.86	2.99

Table 5: Example proxy history

The Mean and the Standard Deviation are calculated using the following:

Mean = AVERAGE (ln (LOC/meth)**)**

StdDev = STDEV (ln (LOC/meth))

2.4.3.1 Calculating the size matrix:

The size matrix is calculated using the following template:

	Low	Mid	High
VS	1	=CEILING(EXP(Average-2*StdDev),1)	=CEILING(EXP(Average-1.5*StdDev),1)
S	=CEILING(EXP(Average-1.5*StdDev),1)	=CEILING(EXP(Average-StdDev),1)	=CEILING(EXP(Average-0.5*StdDev),1)
M	=CEILING(EXP(Average-0.5*StdDev),1)	=CEILING(EXP(Average),1)	=CEILING(EXP(Average+0.5*StdDev),1)
L	=CEILING(EXP(Average+0.5*StdDev),1)	=CEILING(EXP(Average+StdDev),1)	=CEILING(EXP(Average+1.5*StdDev),1)
VL	=CEILING(EXP(Average+1.5*StdDev),1)	=CEILING(EXP(Average+2*StdDev),1)	Big

Table 6: Size matrix formula

The size matrix can be calculated for each type (i.e. Calculation, I/O, Logic, and Data) (or) without considering the types. The following is the size matrix calculation based on the example proxy history provided in Table 5.

Proxy Name	Type	LOC/meth	ln(LOC/meth)
SourceFile	Calculation	12.00	2.48
SourceFile	Calculation	21.10	3.05
print_each_proxy	I/O	20.00	3.00
ProxyHistory	Logic	30.00	3.40
print_proxy_history	I/O	25.00	3.22
print_schedule	I/O	45.00	3.81
Schedule	Calculation	59.33	4.08
TaskList	Logic	6.33	1.85
Task	Data	4.00	1.39
Calendar	Data	6.67	1.90
print_tcurve	I/O	43.00	3.76
TCurve	Calculation	19.86	2.99

	Low	Mid	Upper
VS	1	3	5
S	5	8	12
M	12	18	28
L	28	43	66
VL	66	100	Big

Table 7: Size matrix calculation without considering type.

Once the size matrix is calculated, the raw lines of code (LOCr) for the new project can be calculated by the following steps:

- Identify the relative size for each of the history proxies from the size matrix. This is done by getting the LOC/method for each historical proxy and finding which bucket it falls into. For

example, the proxy “ProxyHistory” in the Table 5has a LOC/Method of 30. If we map 30 in the size matrix, it falls in the Large (L) category.

The following table shows the relative size for each of the history proxies by mapping it with the size matrix:

Proxy Name	**Type**	**LOC/meth**	**ln(LOC/meth)**	**Rel Size**
SourceFile	Calculation	12.00	2.48	**M**
SourceFile	Calculation	21.10	3.05	**M**
print_each_proxy	I/O	20.00	3.00	**M**
ProxyHistory	Logic	30.00	3.40	**L**
print_proxy_history	I/O	25.00	3.22	**M**
print_schedule	I/O	45.00	3.81	**L**
Schedule	Calculation	59.33	4.08	**L**
TaskList	Logic	6.33	1.85	**S**
Task	Data	4.00	1.39	**VS**
Calendar	Data	6.67	1.90	**S**
print_tcurve	I/O	43.00	3.76	**L**
TCurve	Calculation	19.86	2.99	**M**

Table 8: Relative size mapping from size matrix.

- In order to calculate the raw lines of code (LOCr) for the new project, list down all the proxies and the number of operations from the components identified in the Architecture phase.
- Identify the estimated relative size (i.e. VS, S, M, L, or VL) from the size matrix for each of the listed proxies in the new project by comparing it with the historical proxy.
- The raw lines of code (LOCr) for each proxy in the new project are calculated by multiplying the number of operations/methods with the relative size (Mid value) from the size matrix.

2.4.3.2 Example for calculating raw lines of code (LOCr):

Let's assume we have the following new proxies in our new project:

Proxy Name	**Operations**
print_checkconsistency	1
check_consistency	1
Cvalidate	8

Table 9: New proxies

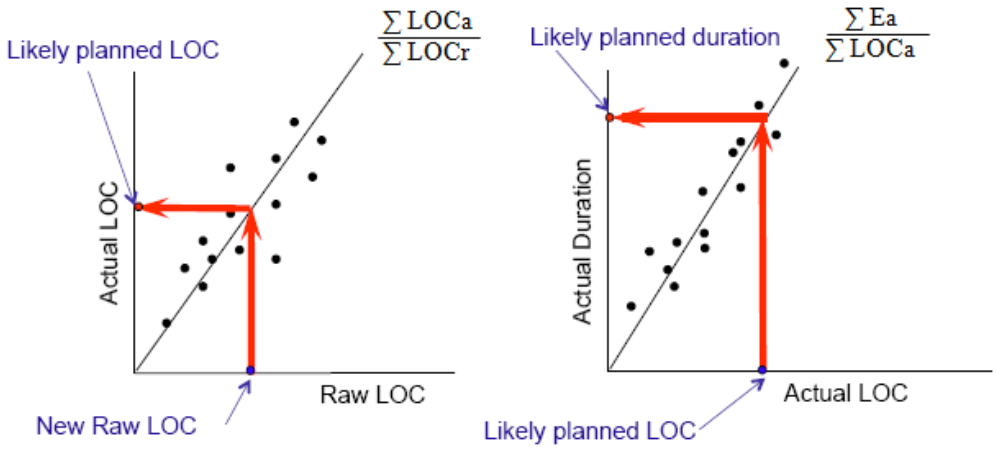
By comparing the historical proxies, we identify the estimated relative size for each of these proxies from the size matrix. Then, the mid value of the corresponding relative size from the size matrix is taken and multiplied with the number of operations to get the LOCr. Finally the total LOCr is calculated by summing up the LOCr of all the new proxies as shown below:

Proxy Name	**Operations**	**Estimated Rel. Size**	**LOCr**
print_checkconsistency	1	**M**	**18**
check_consistency	1	**L**	**43**
Cvalidate	8	**L**	**344**
Total LOCr = 405			

Table 10: Raw lines of code (LOCr) calculation for new proxies

2.4.3.3 Calculating Planned lines of code (LOCp) and Planned duration (Ep):

Once the raw lines of code (LOCr) is calculated, we can calculate the planned lines of code (LOCp) and the planned duration(Ep) with the help of the project history. The following estimation fundamentals will help us understand better how LOCp and Ep are calculated:



$$New\ \text{LOCr} * \frac{\sum \text{LOCa}}{\sum \text{LOCr}} = New\ \text{LOCp}$$

$$New\ \text{LOCp} * \frac{\sum \text{Ea}}{\sum \text{LOCa}} = New\ \text{Ep}$$

Figure 3: Estimation Fundamentals

In the left graph, the dots represent the number of historical projects. The x-axis represents the raw lines of code (LOCr) and the y-axis represents the actual lines of code (LOCa). The line that passes through the historical projects represents the average of LOCa to LOCr. The blue dot represents the raw lines of code (LOCr) for the new project/development (i.e. 405 lines in our example case). The red line that passes through the line is a mapping of LOCr to LOCa for the new development, which results in the planned lines of code (LOCp) for the new development (i.e. *New*LOCp is obtained by multiplying *New*LOCr with the average). The average ∑ LOCa / ∑ LOCr actually represents the productivity. For example, if the average is 1.33, it means that 1.33 lines of code can be written in one minute, but again it depends upon the language, code etc.

In the right graph, the dots represent the number of historical projects. The x-axis represents the actual lines of code (LOCa) and the y-axis represents the actual duration (Ea). The line that passes through the historical projects represents the average of Ea to LOCa. The blue dot represents the planned

lines of code (LOCp) for the new project/development.The red line that passes through the line is a mapping of LOCa to Ea for the new development, which results in the planned duration (Ep) for the new development (i.e. *New*Ep is obtained by multiplying *New*LOCp with the average). The average ∑ Ea / ∑ LOCa actually represents the productivity. For example, if the average is 2.68, it means that it almost takes 3 minutes to write a line of code, but again it depends upon the language, code etc.

Let's assume from the historical database, we have the following data:

Project Name	**LOCr**	**LOCa**	**Ea**
Project 1	30	48	249
Project 2	45	168	419
Project 3	65	146	438
Project 4	273	274	577
Project 5	203	182	513
∑ LOCa / ∑ LOCr = 1.33 ∑ Ea / ∑ LOCa = 2.68			

Table 11: Historical project data

Therefore, If the raw size (LOCa) is 405, then the planned size (LOCp) is ceil(405 * 1.33) = **539 lines** and the planned duration (Ep) is ceil(539 * 2.68) = **1445 minutes.**

2.4.3.4 Calculating confidence:

Calculating confidence involves calculating lower prediction interval (LPI) and upper prediction interval (UPI). The rule is as follows:

For size estimates:

LPI = Estimate * largest historical underestimation error

UPI = Estimate * largest historical overestimation error

For time estimates:

LPI = Estimate * fastest historical productivity

UPI = Estimate * slowest historical productivity

From the historical data, we can calculate the size prediction interval and time prediction interval as follows:

Project Name	**LOCr**	**LOCa**	**Ea**	**LOCa/LOCr**	**(LOCa/Ea) * 60** (Unit: LOC/Hr)
Project 1	30	48	249	1.6	11.4
Project 2	45	168	419	3.73	24.0
Project 3	65	146	438	2.25	19.8
Project 4	273	274	577	1.00	28.8
Project 5	203	182	513	0.90	21.0

New LOCr = 405; *New* LOCp = 539

Table 12: Size and Time prediction intervals calculation

2.4.3.5 Size prediction interval:

Therefore, Lower Prediction Interval **(LPI)** = biggest underestimation error

= New LOCr * min(LOCa/LOCr)

= 405 * 0.90

= **365 lines**

Upper Prediction Interval **(UPI)** = biggest overestimation error

= New LOCr * max(LOCa/LOCr)

= 405 * 3.73

= **1511 lines**

2.4.3.6 Correlation coefficient:

The correlation coefficient, a concept from statistics is a measure of how well trends in the predicted values follow trends in past actual values. It is a measure of how well the predicted values from a forecast model "fit" with the real-life data.

The correlation coefficient is a number between 0 and 1. If there is no relationship between the predicted values and the actual values, the correlation coefficient is 0 or very low (the predicted values are no better than random numbers). As the strength of the relationship between the predicted values and actual values increases so does the correlation coefficient. A perfect fit gives a coefficient of 1.0. Thus the higher the correlation coefficient the better [9].

Let the correlation coefficient ranges be set as follows:

$1.0 < r^2 < .75$ --------- High (means estimation very close to actual values)

$.75 < r^2 < .50$ ---------- Medium (means estimation fairly close to actual values)

$.50 < r^2$ ---------- Low (means estimation is very poor)

2.4.3.7 Confidence on size prediction interval:

In order to find the confidence on the size prediction interval, calculate correlation coefficient which is done as follows:

= [CORREL(∑ LOCa / ∑ LOCr)]^2

= **0.70 [Medium]**

Note:

set UPI to *New* LOCp if UPI<*New* LOCp

set LPI to 1 if LPI >*New* LOCp

also set confidence to low

2.4.3.8 Time prediction interval:

Therefore, Lower Prediction Interval (**LPI**) = biggest overestimation error

= *New* LOCp / max(LOCa/Ea) * 60

= 539 / 28.8 * 60

= **1123 minutes**

Upper Prediction Interval (**UPI**) = biggest underestimation error

= *New* LOCp / min(LOCa/Ea) * 60

= 539 / 11.4 * 60

= **2837 minutes**

2.4.3.9 Confidence on time prediction interval:

In order to find the confidence on the time prediction interval, calculate correlation coefficient which is done as follows:

= [CORREL(∑ Ea / ∑ LOCa)]^2

= **0.93 [High]**

Note:

set UPI to *New*Ep if UPI<*New*Ep

set LPI to 1 if LPI >*New*Ep

also set confidence to low

2.4.4 Iteration Plan:

IBM Rational Unified Process (RUP) [10], an iterative software development process framework, says that iteration plan is a fine-grained plan with a time-sequenced set of activities and tasks, with assigned resources, containing task dependencies, for the iteration.

Iteration also involves the redesign and implementation of a task from the operational specification list, and the analysis of the current version of the system. It helps identify problems or faulty assumptions at periodic intervals.

PCSE has the following major goals during this phase:

a) Select/revise scenario set

b) Set iteration goal

c) Schedule work

2.4.4.1 Select/revise scenario set:

During every iteration, one can select the next set of scenarios identified in the Analysis phase, for the implementation. The operational specification can also be revisited to modify any scenario or add new scenarios.

2.4.4.2 Set iteration goal:

The iteration goal is a plan or a set of tasks intended to be achieved by the end of the iteration. Some of the iteration goals are:

1) A list of major classes or packages that must be completely implemented.
2) A list of scenarios or use cases that must be completed by the end of the iteration.
3) A list of risks that must be addressed by the end of the iteration.
4) A list of changes that must be incorporated in the product (bug fixes, changes in requirements) etc.

2.4.4.3 Schedule work:

One of the important artifacts of iteration plan in PCSE is the iteration map, where each part in the component identified during the architecture phase is mapped to an iteration. However the number of iterations is not limited. There are situations where we write mock code before we write production code. Each of these can be easily captured and tracked using an iteration map. The iteration map also leads an easy way to schedule and track the effort, using calendar, burn-down chart, and diary. At the other end, the iteration map can also be mapped to scenarios.

After we know the effort for each iteration, the work can be scheduled using a calendar and tracked using the burn down chart and diary. The calendar, burn-down chart, and dairy are important artifacts in PCSE in order to schedule and track the effort.

The following figure illustrates how calendar is used to measure and track effort for every iteration.

SEPTEMBER 2010

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
			1 90 90	2 60 150	3 0 150	4 90 240
5 0 240	i1 6 120 360	7 360 720	i2 8 240 960	9 0 960	10 90 1050	11 0 1050
12 0 1050	i3 13 90 1140	i4 14 120 1260	15 90 1350	16 150 1500	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

Figure 4: Calendar

The iteration map is scheduled to complete 13 parts in 4 iterations with an estimated duration of 1445 minutes. The calendar is used to plan the effort on a daily basis. In the above figure, the numbers in the center (green in color) represents the number of minutes the developer has planned on writing the part on the given day. The number at the bottom right end (blue in color) of each day is the cumulative total of minutes planned so far (cumulative planned time), from which the day an iteration will end is known). For example, iteration1 (i1) requires an effort of 333 minutes. So, from the calendar above, iteration1 will be completed on day 6. The end of each iteration is marked in red at the top left end of the completion day. The actual effort spent during each day can be tracked using a burn-down chart and a diary.

2.4.4.4 Burn-down chart:

A burn-down chart is a graphical representation of work left to do versus time. The outstanding work (or backlog) is often on the vertical axis, with time along the horizontal. That is, it is a run chart of outstanding work. It is useful for predicting when all of the work will be completed. The end of each day is termed as the recording point and the end of each iteration is termed as assessment point [11].

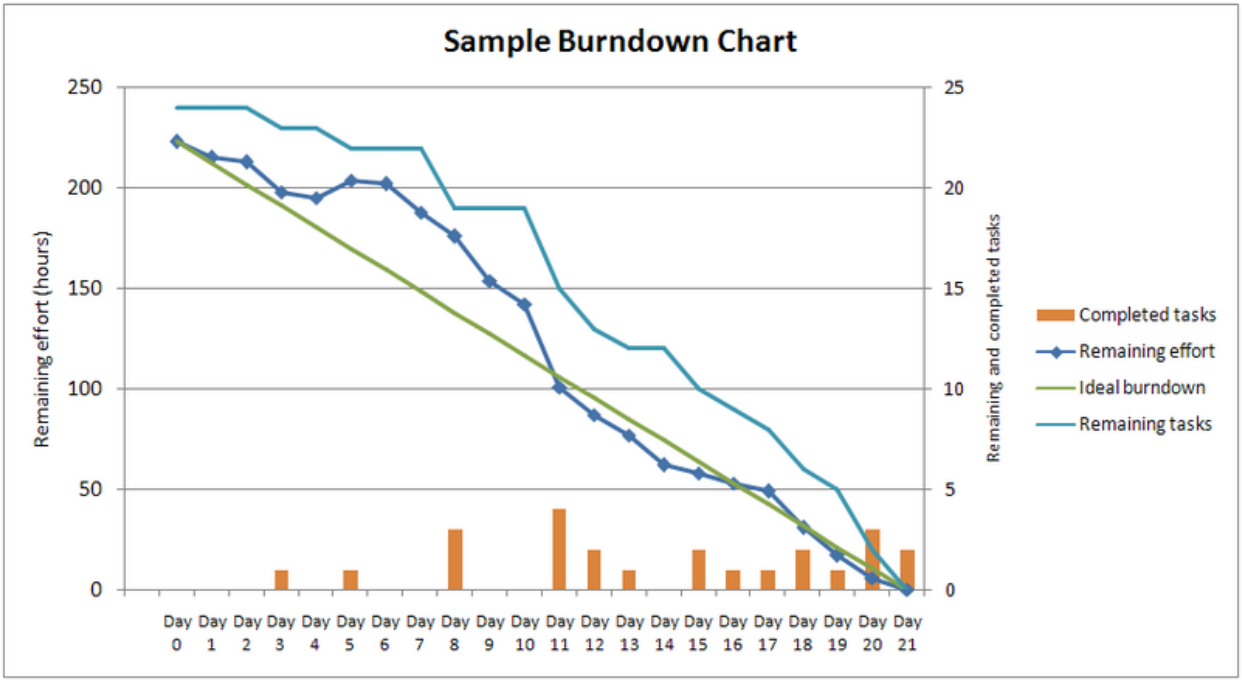


Figure 5: A sample burn-down chart for a completed iteration, showing remaining effort and tasks for each of the 21 work days of the 1-month iteration.

2.4.4.5 Burn down chart example case:

The following is an example burn-down chart constructed based on the work scheduled in the iteration map and the calendar. It represents the Planned Effort(Ep), Actual Effort (Ea) and the replannedeffort. i1, i2, i3, i4 etc. represent each iteration. The x-axis represents the days and the y-axis refers to remaining effort.

After each day the progress is marked on the chart and after each iteration the developer projects forward to see whether or not the target end date will be hit.

In the diagram below, the developer had gone through three iterations, and the dotted line (work effort to maintain schedule) suggests he had fallen quite badly behind schedule. After the first iteration the developer was making good progress, but things changed in the second and third iteration. Although the developer was behind the planned schedule at the end of the second iteration, he thought that he could still finish the remaining parts. Since the remaining time was not realistic to finish the remaining parts, the developer decided to reschedule at the end of third iteration.

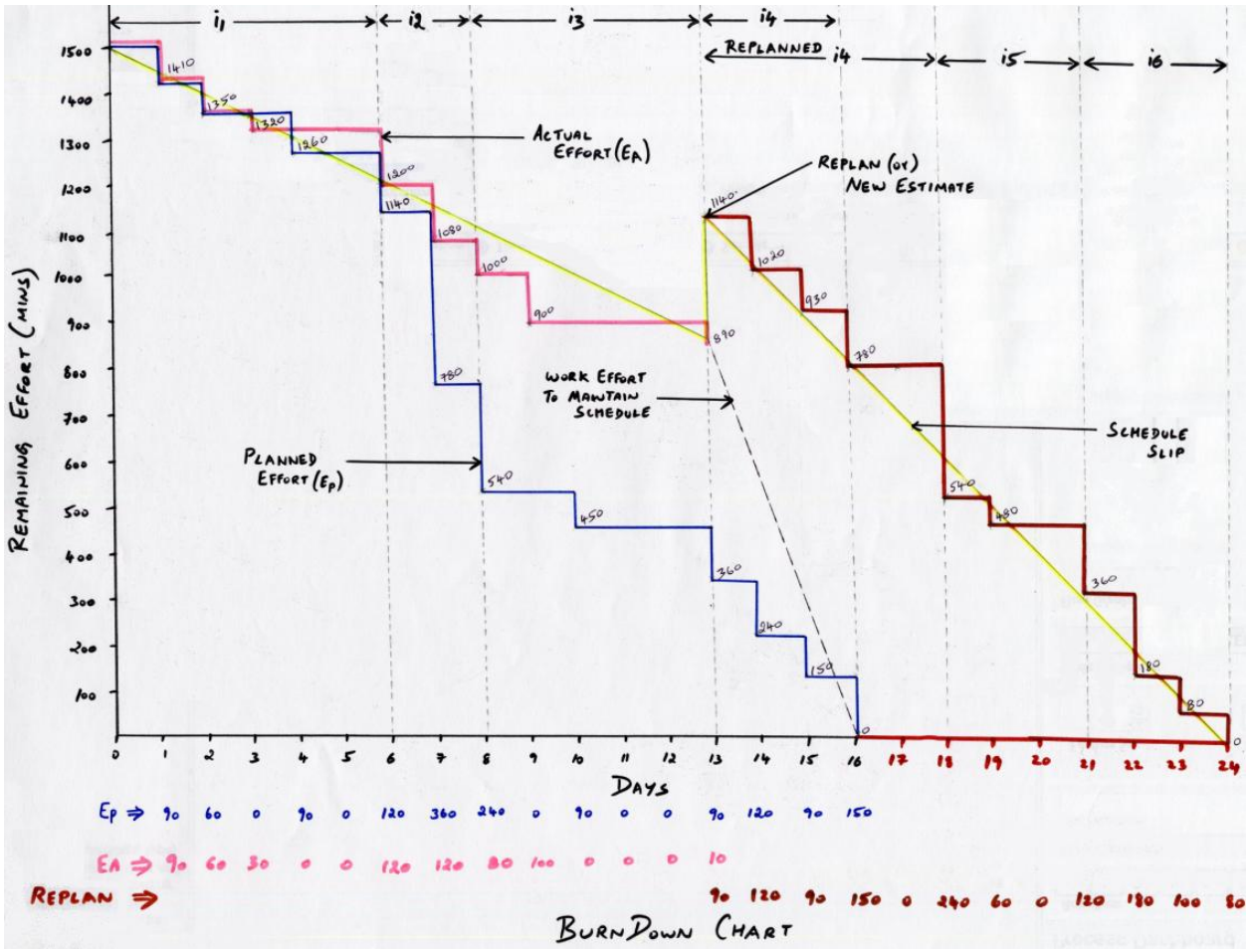


Figure 6: Example burn-down chart

2.4.4.6 Diary:

A diary is used for measuring the project progress. It provides a way to track and make decisions at periodic intervals, particularly when the developer completes the tasks in a different order than

originally planned. Planned Value (PV) and Earned Value (EV) are two important measures that represent the planned work and completed work respectively. PV and EV are also termed as Planned Velocity and Earned Velocity.

Planned Value (PV): The percentage of total planned project time that the planned task represents.

Earned Value (EV): The planned value of a task is earned when the task is completed. There is no partial credit for partially completed tasks; i.e. Earning a EV means method is completed passing all the tests.

If the Earned value (EV) is less than the Planned value (PV), then the developer is accomplishing the work late or behind schedule. If the Earned value (EV) is equal or more than the Planned value (PV), then the developer is accomplishing the work on, or ahead of plan.

A diary holds both the planned and the actual data. The following are the important data used in the Diary for measuring the progress:

Planned Time: Planned effort for a day.

Planned Burn Down: Total planned effort – Planned Time.

Planned Iteration: Mark the completion of a set of planned parts.

Cum PV: The running cumulative sum of the planned values.

Actual Time: Actual effort spent on planned tasks on a day.

Cum Actual Time: The running cumulative sum of the actual time.

Actual Burn Down: Total actual effort – Actual time.

Cum EV: The running cumulative sum of earned values for the completed tasks.

Backlog Δ: represent the total amount of work.

The following is an example diary constructed based on the work scheduled in the iteration map and the calendar. From the historical project data, the estimated planned duration (Ep) was calculated as 1445 minutes. The number of parts determined in the iteration plan was 13 parts in 4 iterations (i1: 3 parts, i2: 5 parts, i3: 2 parts, i4: 3 parts). The number of days scheduled in the calendar was 16 days.

After each day the progress is marked on the chart and after each iteration the developer project forward to see whether or not he will hit the target end date. At the end of iteration 1 (i1) on day 6, the developer had completed 3 parts and so the earned value (EV) is 3. In, iteration 2 (i2), the developer did not progress well and so he earned only 1 part as against the planned value (PV), 5 parts. Although the developer was behind the planned schedule at the end of the second iteration, he thought that he could still finish the remaining parts. At the end of iteration 3 (i3) on day 13, the developer had earned only 1 part against the PV, 3 parts. Also, in iteration 3 (i3), the developer discovered that he needed to write 2 more new parts in order to complete the project. Since the remaining time was not realistic to finish the remaining parts, the developer decided to reschedule at the end of third iteration.

Day	Planned Time	Planned Burn Down	Planned Iteration	Cum PV	Actual Time	Cum Actual Time	Actual Burn Down	Cum EV	Backlog Δ
1	90	1410		0	90	90	1410	0	0
2	60	1350		0	60	150	1350	0	0
3	0	1350		0	30	180	1320	0	0
4	90	1260		0	0	180	1320	0	0
5	0	1260		0	0	180	1320	0	0
6	120	1140	i1: 1167 (Effort: 333)	3	120	300	1200	3	-3
7	360	780		3	120	420	1080	3	0
8	240	540	i2: 610 (Effort: 557)	8	80	500	1000	4	-1
9	0	540		8	100	600	900	5	-1
10	90	450		8	0	600	900	5	0
11	0	450		8	0	600	900	5	0
12	0	450		8	0	600	900	5	0
13	90	360	i3: 388 (Effort:	10	10	610	890	5	+2

			222)						
14	120	240		10					
15	90	150		10					
16	150	0	i4: 55 (Effort: 333)	13					

Table 13: Diary (Example case) – Planned Vs Actual

At the end of i3 on day 13, the developer earned 5 parts and also discovered that he need to write 2 more new parts. Hence the total amount of work (Backlog) to complete becomes:

Backlog of work = 13 (planned parts) – 5 (EV at i3)

= 8 + 2 (newly discovered)

= 10 parts

The total backlog is 10 parts and the developer decided to change the iteration plan such that he will complete the remaining parts in 3 iterations as follows: (i4: 4 parts, i5: 3 parts, i6: 3 parts). To re-estimate the time for each iteration, the developer has to make use of the cumulative actual time spent and number of EV earned until end of i3. The calculation is done as follows:

Re-estimate (i4) = cum actual time / items built (EV) * items to build in i4

= 610 / 5 * 4

= 488

Re-estimate (i5) = cum actual time / items built (EV) * items to build in i5

= 610 / 5 * 3

= 366

Re-estimate (i5) = cum actual time / items built (EV) * items to build in i6

= 610 / 5 * 3

= 366

The total new planned duration is: 1220 (488 + 366 + 366).

The developer starts recalculating the burndown at the end of i3 on day 13 as follows:

Recalculate burndown = (new planned duration - time remaining in the day (Day 13))

= 1220 – 80

= 1140

Once the burndown is recalculated, the calendar is scheduled and the planned time for each day is recorded in the diary. The following is the diary after re-estimate.

Day	Planned Time	Planned Burn Down	Planned Iteration	Cum PV	Actual Time	Cum Actual Time	Actual Burn Down	Cum EV	Backlog Δ
1	90	1410		0	90	90	1410	0	0
2	60	1350		0	60	150	1350	0	0
3	0	1350		0	30	180	1320	0	0
4	90	1260		0	0	180	1320	0	0
5	0	1260		0	0	180	1320	0	0
6	120	1140	i1: 1167 (Effort: 333)	3	120	300	1200	3	-3
7	360	780		3	120	420	1080	3	0
8	240	540	i2: 610 (Effort: 557)	8	80	500	1000	4	-1
9	0	540		8	100	600	900	5	-1
10	90	450		8	0	600	900	5	0
11	0	450		8	0	600	900	5	0
12	0	450		8	0	600	900	5	0
13	90	**1140**	i3: 388 (Effort: 222)	10	10	610	890	5	+2
recalculate burndown (1220 - time remaining in the day) = 1220 – 80 = 1140									

14	**120**	**1020**		**5**					
15	**90**	**930**		**5**					
16	**150**	**780**		**5**					
17	**0**	**780**		**5**					
18	**240**	**540**	**i4: 732 (Effort: 488)**	**9**					
19	**60**	**480**		**9**					
20	**0**	**480**		**9**					
21	**120**	**360**	**i4: 366 (Effort: 366)**	**12**					
22	**180**	**180**		**12**					
23	**100**	**80**		**12**					
24	**80**	**0**	**i4: 0 (Effort: 366)**	**15**					

Table 14: Diary (Example case) after Re-estimate

2.4.5 Construction:

Construction is the activity of building code given a high-level design, which involves low-level design, coding and unit testing.

In PCSE, the following are the important activities of this phase:

a.) Develop low-level design

b.) Build code and unit tests via TDD

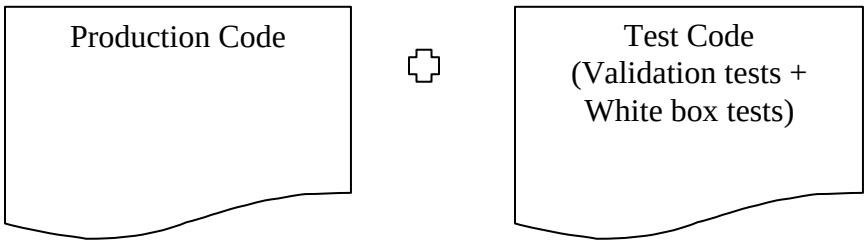


Figure 7: PCSE construction artifacts

Traditionally testing was done after the coding activity, but PCSE uses Test Driven Development (TDD) approach where first the tests are written and then code to pass them.

Design → Code → Test
Traditional Approach

Design → Test → Code
PCSE Approach

The problems of the traditional approach are:

a.) Tests are often written based on non-code artifacts, which may work for black-box tests (that tests the functionality of an application), but not necessarily for white-box tests (a method of testing software that tests internal structures or workings of an application).

b.) Tests may be written by non-coders who lack white box knowledge.

c.) Testing is done after code is written

2.4.5.1 Test Driven Development [12]:

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, and then produces code to pass that test [12]. It is a systematic approach to programming where the tests determine what code to write.

The advantages of TDD approach are:

a.) all delivered code is accompanied by tests

b.) no code goes into production untested

c.) writing tests first yield a better understanding of what the code is to do

The following figure represents a complete test driven development cycle:

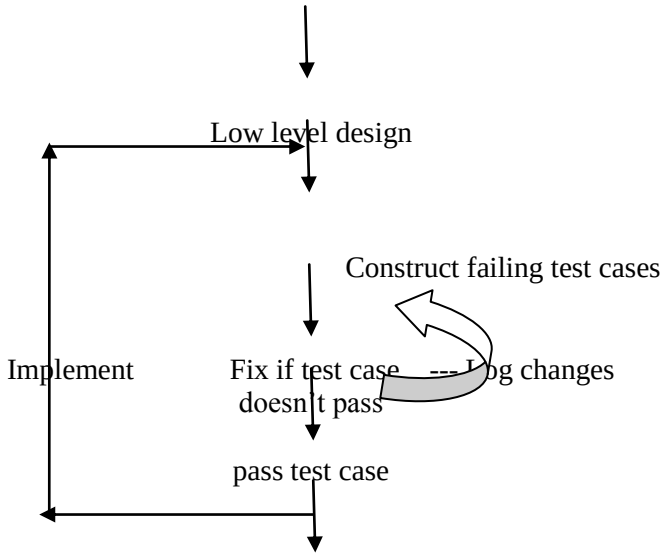


Figure 8: Test driven development cycle

Test cases can also be written using automated testing tools such as JUnit, NUnit etc.

2.4.6 Review:

Review is a phase where the developer examines the test code for coverage. This is the phase where the developer ensures enough black box tests and white box tests are constructed. If the test code is not covered, then the developer can add tests appropriately and fix it via TDD.

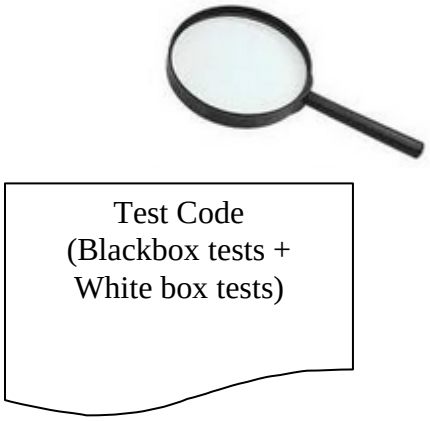


Figure 9: Review Test Code

The better the quality, the lower the cost. Test code reviews help ensure better quality and standards. It will not only improve the quality of the test code, but it can also expose any assumptions made by the developer and help in generating more test cases. Some of the most common problems to look for during a test code review are:

a) Creating test code that tests functionality already tested in a different test method. For example testing a "get" along with a “set” and also having a separate test method for testing the "get".
b) Testing the same functionality dropped in two different releases by rewriting existing test cases just for the new release. Existing test code should be written in a way that it works across multiple releases.
c) Re-writing functions that have part of the same functionality implemented by other methods should be avoided. For example, if functionality involves opening a file, reading a value from it, connecting to a database and getting values out and if methods exist for each of these tasks then they should be reused instead of creating a new method that does all three tasks.
d) Verifications should be performed for all possible fields of the object being tested. This should include core fields and also audit fields such as created, updated date etc. [13]

The advantages of performing a test code review are:

a) Tests reveal the intention behind the code much better than the code itself. That means it's easier to discover logical bugs in the code by reading a test.
b) Tests are declarative by default - the developer declares what the code is supposed to be accomplishing.
c) Tests are faster and shorter to read and understand. [14]

2.4.7 Refactor:

Code refactoring is the process of changing a computer program's source code without modifying its external functional behavior in order to improve some of the nonfunctional attributes of the software.

Also the internal structure of the software is improved. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility [15].

The purpose of refactoring is

a.) To make software easier to understand

b.) To help find bugs

c.) To prepare software for next iteration

d.) To speed development process

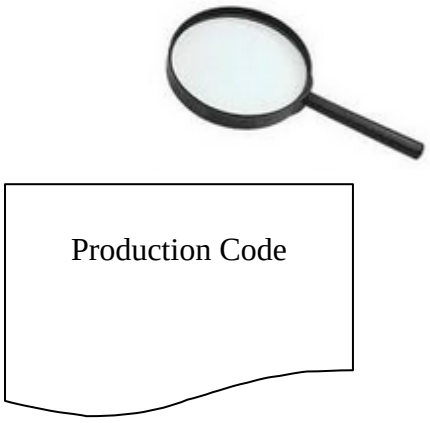


Figure 10: Refactor production code

Refactoring is usually motivated by noticing a code smell. For example the method at hand may be very long, or it may be a near duplicate of another nearby method. Once recognized, such problems can be addressed by refactoring the source code, or transforming it into a new form that behaves the same as before but that no longer "smells". We can also add/modify tests appropriately and fix it via TDD, if we notice code smell in the production code.

The following are some examples of code smell that can be addressed through refactoring [1]:

a.) ***Data clumps:*** member fields that clump together but are not part of the same class.

b.) ***Primitive obsession:*** characterized by the use of primitives in place of class methods

c.) ***Switch statements:*** often duplicated code that can be replaced by polymorphism

d.) ***Parallel inheritance hierarchies:*** duplicated code in subclasses that share a common ancestor.

e.) ***Duplicated code***

f.) ***Long method***

g.) ***Large class***

h.) ***Long parameter list***

i.) ***Divergent change:*** one type of change requires changing one subset of modules; another type of change requires changing another subset.

j.) ***Shotgun surgery:*** a change requires a lot of little changes to a lot of different classes.

k.) ***Feature envy:*** a method in a class seems not to belong.

l.) ***Lazy class:*** a class that has little meaning in the context of the software

m.) ***Speculative generality:*** methods (often stubs) that are placeholders for future features.

n.) ***Temporary field:*** a variable that is used only under certain circumstances and is reused later under other circumstances.

o.) ***Message chains:*** object that requests an object for another object.

p.) ***Middle man:*** a class that is just a "pass-through" method with little logic

q.) ***Inappropriate intimacy:*** violation of private parts.

r.) ***Alternate class with different interfaces:*** two methods that do the same thing, but have different interfaces.

s.) ***Incomplete library classes:*** a framework that doesn't do everything you need.

t.) ***Data class:*** classes that have getters and setters, but no real function.

u.) ***Refused bequest:*** a subclass that over-rides most of the functionality provided by its super class.

v.) ***Comments:*** text that explains bad code (vs fixing the code).

2.4.8 Integration:

This is a phase where the software component undergoes regression tests before its actual integration.

Regression testing is any type of software testing that seeks to uncover software errors by partially retesting a modified program. The intent of regression testing is to provide a general assurance that no additional errors were introduced in the process of fixing other problems. Regression testing is commonly used to test the system efficiently by systematically selecting the appropriate minimum suite of tests needed to adequately cover the affected change. Common methods of regression testing include rerunning previously run tests and checking whether previously fixed faults have re-emerged. "One of the main reasons for regression testing is that it's often extremely difficult for a programmer to figure out how a change in one part of the software will echo in other parts of the software" [16].

The following methodology is followed in PCSE to ensure all test cases introduced in one particular iteration do not alter the behavior of the overall system:

For i in regression tests

If not passed(i)

Declare i invalid & discard

Defer i to backlog

Fix i this iteration

2.4.9 Post Mortem:

This is a phase to prepare for the next iteration. The major activities addressed in PCSE during this phase are:

a.) Baseline the production source code in version control

b.) Baseline the test code in version control

c.) Revisit estimation if necessary

d.) Revisit iteration map if necessary

e.) Revisit backlog to add/remove scenarios etc.

2.4.10 Code Complete:

Code complete is to mark the end of the development. Some of the major activities carried out during this phase are:

1.) Final testing of the system.

2.) User documentation.

3.) Deployment.

4.) Training.

5.) Documentation of findings, information to improve future efforts.

2.5 Introduction to Django

Django is an open source web application framework, written in Python, which follows the model-view-controller architectural pattern**.** Django's primary goal is to ease the creation of complex, database-driven websites. Django emphasizes reusability and "pluggability" of components, rapid development, and the principle of don't repeat yourself. Python is used throughout, even for settings, files, and data models. Django also provides an optional administrative create, read, update and delete interface that is generated dynamically through introspection and configured via admin models [17].

The core Django **MVC** framework consists of an object-relational mapper which mediates between data models (defined as Python classes) and a relational database ("**M**odel"); a system for processing requests with a web templating system ("**V**iew") and a regular-expression-based URL dispatcher ("**C**ontroller").Also included in the core framework are:

- A lightweight, standalone web server for development and testing.
- A form serialization and validation system which can translate between HTML forms and values suitable for storage in the database.
- A caching framework which can use any of several cache methods.

- Support for middleware classes which can intervene at various stages of request processing and carry out custom functions.
- An internal dispatcher system which allows components of an application to communicate events to each other via pre-defined signals.
- An internationalization system, including translations of Django's own components into a variety of languages.
- A serialization system which can produce and read XML and/or JSON representations of Django model instances.
- A system for extending the capabilities of the template engine.
- An interface to Python's built-in unit test framework.

2.5.1 Design the model

Although we can use Django without a database, it comes with an object-relational mapper in which we describe our database layout in Python code [18].

The data-model syntax offers many rich ways of representing our models. Here's a quick example, which might be saved in the file mysite/news/models.py:

```
class Reporter(models.Model):
    full_name=models.CharField(max_length=70)
    def__unicode__(self):
        returnself.full_name


class Article(models.Model):
```

```
    pub_date=models.DateTimeField()
    headline=models.CharField(max_length=200)
    content=models.TextField()
    reporter=models.ForeignKey(Reporter)
    def __unicode__(self):
        return self.headline
```

2.5.2 Install it

Next, run the Django command-line utility to create the database tables automatically:

```
manage.py syncdb
```

The syncdb command looks at all our available models and creates tables in your database for whichever tables don't already exist [18].

2.5.3 Design our URLs

A clean, elegant URL scheme is an important detail in a high-quality Web application. To design URLs for an app, we create a Python module called a *URLconf*. A table of contents for our app, it contains a simple mapping between URL patterns and Python callback functions. URLconfs also serve to decouple URLs from Python code [18].

Here's what a URLconf might look like for the Reporter/Article example above:

```
From django.conf.urls import patterns,url,include
```

```
urlpatterns=patterns('',

(r'^articles/(\d{4})/$','news.views.year_archive'),

(r'^articles/(\d{4})/(\d{2})/$','news.views.month_archive'),

(r'^articles/(\d{4})/(\d{2})/(\d+)/$','news.views.article_detail'),

)
```

The code above maps URLs, as simple regular expressions, to the location of Python callback functions ("views"). The regular expressions use parenthesis to "capture" values from the URLs. When a user requests a page, Django runs through each pattern, in order, and stops at the first one that matches the requested URL. (If none of them matches, Django calls a special-case 404 view.) This is blazingly fast, because the regular expressions are compiled at load time.

Once one of the regexes matches, Django imports and calls the given view, which is a simple Python function. Each view gets passed a request object -- which contains request metadata -- and the values captured in the regex.

For example, if a user requested the URL "/articles/2005/05/39323/", Django would call the functionnews.views.article_detail(request, '2005', '05', '39323').

2.5.4 Write our views

Each view is responsible for doing one of two things: Returning an HttpResponse object containing the content for the requested page, or raising an exception such as Http404. The rest is up to us. Generally, a view retrieves data according to the parameters, loads a template and renders the template with the retrieved data [18]. Here's an example view for year_archive from above:

```
def year_archive(request,year):
```

```
    a_list=Article.objects.filter(pub_date__year=year)

    return render_to_response('news/year_archive.html',{'year':year,'article_list':a_list})
```

This example uses Django's *template system*, which has several powerful features but strives to stay simple enough for non-programmers to use.

2.5.5 Design our templates

The code above loads the news/year_archive.html template. Django has a template search path, which allows us to minimize redundancy among templates. In our Django settings, we specify a list of directories to check for templates. If a template doesn't exist in the first directory, it checks the second, and so on [18].

Let's say the news/year_archive.html template was found. Here's what that might look like:

```
{%extends"base.html"%}

{%block title%}Articles for {{year}}{%endblock%}

{%block content%}

<h1>Articles for {{year}}</h1>

{%for article in article_list%}

<p>{{article.headline}}</p>

<p>By{{article.reporter.full_name}}</p>

<p>Published {{article.pub_date|date:"F j, Y"}}</p>

{%endfor%}
```

{%endblock%}

Variables are surrounded by double-curly braces. {{ article.headline }} means "Output the value of the article's headline attribute." But dots aren't used only for attribute lookup: They also can do dictionary-key lookup, index lookup and function calls.

Note {{ article.pub_date|date:"F j, Y" }} uses a Unix-style "pipe" (the "|" character). This is called a template filter, and it's a way to filter the value of a variable. In this case, the date filter formats a Python datetime object in the given format (as found in PHP's date function; yes, there is one good idea in PHP).

We can chain together as many filters as we'd like. We can write custom filters. We can write custom template tags, which run custom Python code behind the scenes.

Finally, Django uses the concept of "template inheritance": That's what the {% extends "base.html" %} does. It means "First load the template called 'base', which has defined a bunch of blocks, and fill the blocks with the following blocks." In short, that lets us dramatically cut down on redundancy in templates: each template has to define only what's unique to that template.

Here's what the "base.html" template might look like:

<html>

<head>

<title>**{%block**title**%}{%endblock%}**</title>

</head>

<body>

<imgsrc="sitelogo.gif"alt="Logo"/>

{%blockcontent**%}{%endblock%}**

</body>

</html>

It defines the look-and-feel of the site (with the site's logo), and provides "holes" for child templates to fill. This makes a site redesign as easy as changing a single file -- the base template. It also lets us create multiple versions of a site, with different base templates, while reusing child templates. Django's creators have used this technique to create strikingly different cell-phone editions of sites -- simply by creating a new base template.

Note that we don't have to use Django's template system if we prefer another system. While Django's template system is particularly well-integrated with Django's model layer, nothing forces us to use it. For that matter, we don't have to use Django's database API, either. We can use another database abstraction layer, we can read XML files, we can read files off disk, or anything we want. Each piece of Django -- models, views, templates -- is decoupled from the next [18].

3. Changes in PCSE process and artifacts

3.1 Objective

The goal of this thesis work was to apply PCSE process to develop a web application in a web development environment. In the past, PCSE had been used to develop only conventional applications. All the phases of PCSE were designed in a way to suit the development of conventional applications. The main challenge of the thesis was to adapt PCSE phases and artifacts to accommodate the unique aspects of web application development. The PCSE lifecycle was adapted to fit the non-homogenous artifacts that resulted during the development of the web application. The different modifications made to the PCSE process are explained in the next section.

3.2 Changes made to PCSE

The changes that were made to the different phases of the PCSE lifecycle were as follows:

- **Analysis –** Analysis is the process of breaking a complex topic into smaller parts to gain a better understanding of it. This stage includes identifying the desired behavior of the system. The artifact produced is usually user scenarios for each task of the system. Since the web application had raw user needs, the requirements were written as user stories on index cards. Each requirement was written on a separate index card and was represented as an individual feature.

- **Architecture -** The main motive of this phase was to develop a high-level design and to identify major components sufficient to begin scoping the effort required by the project. The

artifacts produced were CRC cards which is the usual PCSE artifact for architecture. But, the traditional component types defined by PCSE were not enough to accommodate the non-homogenous artifacts of the web application. So, three new component types model, template and view were defined to fit these artifacts.

- **Planning –** The planning and estimation method used by PCSE is the proxy based estimation method. It depends on the historical data for predicting the size and effort of the development of new components. Since we were dealing with a web application for the first time, there was no historical data to aid the planning phase. So, a new estimation method was used. The method used was story point based estimation. The size of the features was calculated relative to a base feature and the effort was calculated relative to the effort of the base feature. The templates (web pages and style sheets) were estimated relative to the size of the existing templates.

- **Construction –** Construction in PCSE is done by Test-Driven Development (TDD). The construction for the web application was done by following TDD. Different methods were employed to test the different component types. The web pages were tested by the use of Django's built-in test client class.

The details of the changes made to the PCSE process and the different artifacts produced are discussed in detail in the next chapter.

4. Web Application in Django using PCSE

The following sections illustrate the objective of this thesis by using PCSE to develop the web application "NEED-A-NERD" that entailed building Python code, Django template code, and XHTML. To understand how well PCSE could be applied to develop web based applications, a web based application was implemented using the PCSE process. The web application called "NEED-A-NERD" was developed to be used by the faculty, staff and students of the Computer Science and Software Engineering department at Auburn University for their on-campus job search. The web based interface would let faculty and staff do a variety of functions including adding new jobs to the site, editing their existing job postings, search for students based on skill set they expect, download students' resumes and so on. The students can do a lot of functions including applying for specific jobs, search for jobs, update their resume and profile information and so on. There will also be an administrator who will handle all the maintenance functions.

The application was developed in Django, which is a web application framework written in Python. It was constructed using TDD (Test-Driven Development). The Eclipse IDE was used for development purposes. A plugin for the Eclipse IDE called PyDev was used to develop the Python code. PyDev also has a unit-testing framework for testing Python code which was used for testing purposes. A custom MySQL database was used as the backend instead of the built-in SQLite3 database. HTML and CSS were used in constructing the web pages.

4.1 Analysis

Analysis is the phase where the requirements of the system are identified. Analysis was done with the help of user stories. A user story, also called as scenario, expresses one very specific need that a user has. It usually consists of a few simple sentences. The user stories were written as plain text in English. Any user would be able to read a user story and immediately understand what it means. To put it simply, a user story is a raw user need. It is something that the user needs to do on the interface. User stories were written on index cards since they are easy to work with. Each requirement was written on an index card.

The following are a few important facts about user stories [20].

- **Users –** Since the users are faculty, staff and students of the Computer Science Department, members of the PCSE research team (includes faculty and students) played a major role in building user stories.
- **Estimated size –** Each user story was given an estimate of how much effort it would take to implement. The way we estimated was to assign user story points to each card, a relative indication of how long it will take a programmer to implement the story. For example, if the programmer has determined that it takes an average of three hours to implement a story point, the number of hours to implement a user story will be roughly three times the number of story points.
- **Priority –** Requirements were prioritized according to the importance of the feature described by the user story. We assigned priorities to each user story and implemented the most important user stories in the first iteration, then the next set of prioritized user stories in the next iteration and so on. If the priority of a story changed over time, we moved it to the most appropriate iteration.

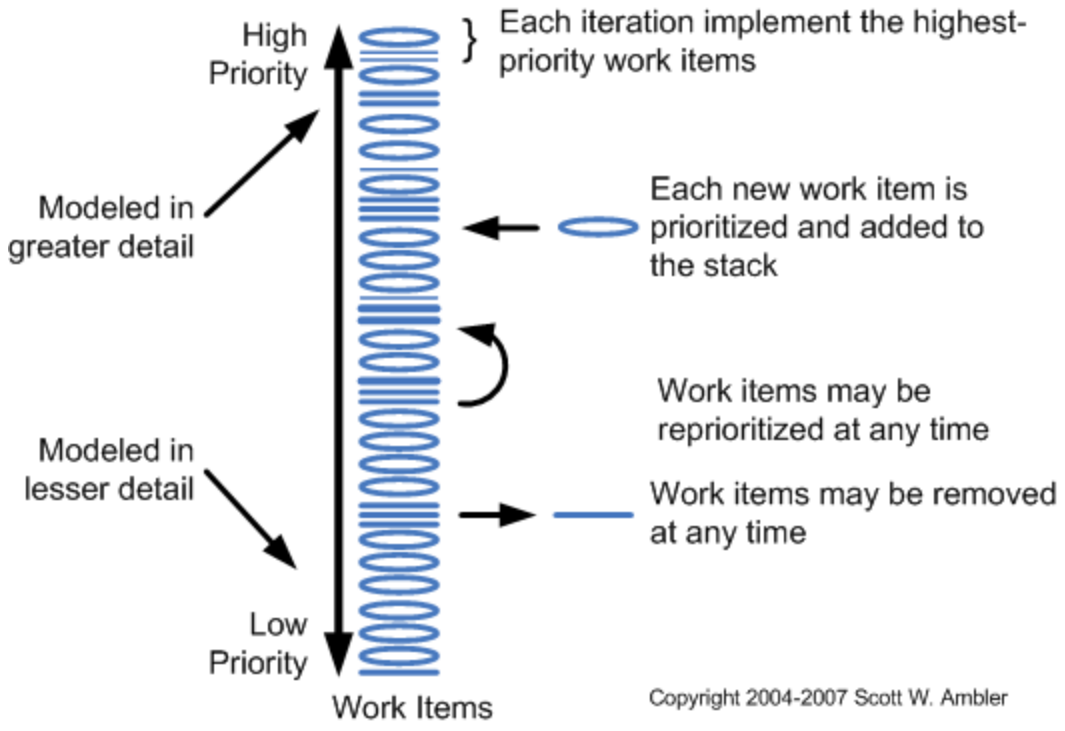


Figure 11: User story prioritization[20]

- **Unique identifier –** Each card also included a unique identifier for the user story. The reason this was done was to maintain some sort of traceability between the user story and other artifacts.

Figure 12 illustrates a sample user story.

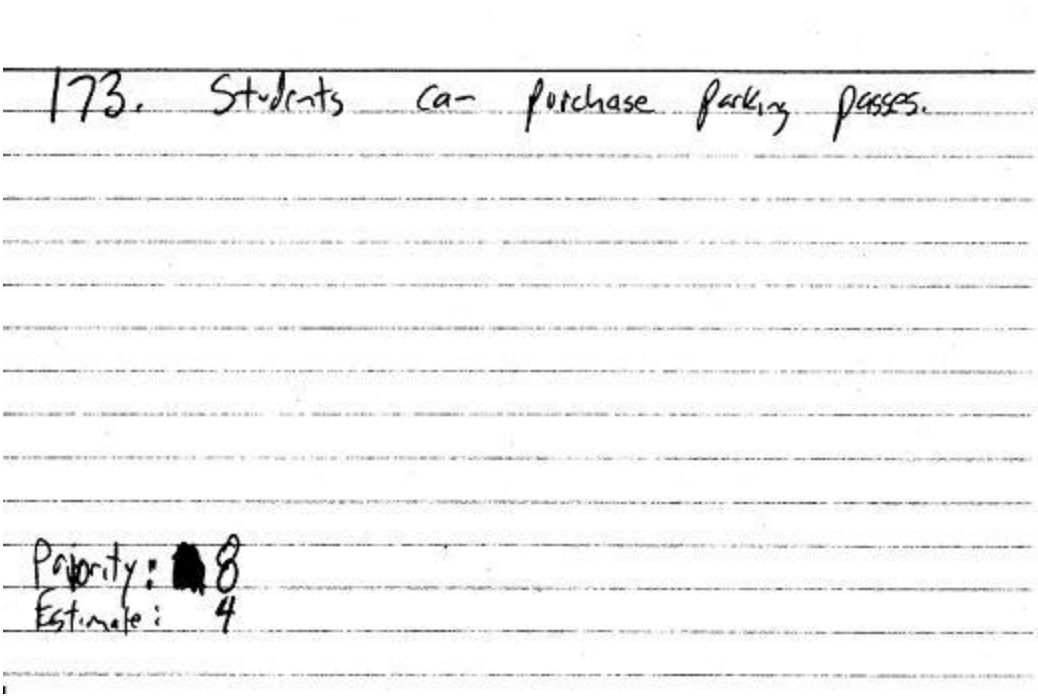


Figure 12: Example user story on index card

User stories that generated during the analysis phase were

- Students should be able to post their resumes.
- Students should be able to update their profile information.
- Employers should be able to post new jobs
- Employers should be able to search for students based on skill set
- Admin should be able to put the site on maintenance mode
- System should be able to expire resumes on a periodic basis
- System must be able to update the recently posted jobs on the site.

4.2 Architecture

Architecture is the phase where the components of the system are identified with the help of CRC cards. Each user story is taken separately and the components that are needed to implement the user story

are identified. In other words, each user story is mapped onto the corresponding components that need to be built to complete the task proposed by the user story. Traditionally, since PCSE has been used to develop conventional applications, the four different components that PCSE defines are:

- Data
- Calculation
- I/O and
- Logic

Since we were dealing with a web based application, the components did not quite fit into any of these four categories. We decided to define three new component types for the web application to be developed in Django. These three new component types were:

- Model
- Template and
- View

4.2.1 Model

A model in Django is a Python class that wraps a database table. Django models are persistent. Every time we create a new instance of the model, it becomes an object (a tuple in the database table). Models can inherit from other models just as classes do. Models can also use fields from other models as references (foreign keys). In other words, models can collaborate with other models. Similar to classes, models have member variables which become columns in the database table. They have member functions which can be used to access and manipulate the data from the database table. Figure 13 gives the CRC cards for some of the models in the web application developed.

Component Name:	UserProfile
Design Approach:	Object-oriented
Parent Component:	None
Component Type:	Model
Collaborators:	Users
Operations:	__unicode__,getUserType()

Component Name:	Jobs
Design Approach:	Object-oriented
Parent Component:	None
Component Type:	Model
Collaborators:	UserProfile
Operations:	__unicode__,getJobID(),getDescription(),getCreationDate(),getContactInfo()

Figure 13: CRC cards for Models

4.2.2 Template

Templates in Django are the web pages and the stylesheets that go along with the content of the pages. In our web applications, all web pages were designed using HTML and CSS. So, the templates are basically the HTML pages and the CSS documents that guide the styles of the HTML pages.

Component Name:	applyjob_complete.html
Design Approach:	Functional
Parent Component:	None
Component Type:	Template
Collaborators:	None
Operations:	applyjob_complete.html

Component Name:	comment_posted.html
Design Approach:	Functional
Parent Component:	None
Component Type:	Template
Collaborators:	None
Operations:	comment_posted.html

Component Name:	contact_student.html
Design Approach:	Functional
Parent Component:	None
Component Type:	Template
Collaborators:	None
Operations:	contact_student.html

Figure 14: CRC cards for templates

A few of the CRC cards developed for templates in the web application can be seen in Figure 14. Each HTML page and each CSS document is denoted by a single CRC card.

4.2.3 View

Each view is a standalone Python method which retrieves the data to be displayed on a particular web page. A view in Django is the component that controls the data that will be displayed on a web page. The view decides on the contextual data that would be displayed to different types of users. Generally, a view retrieves the data according to the parameters, loads a template and renders the template after populating it with the retrieved data.

Component Name:	studentjobs
Design Approach:	Functional
Parent Component:	None
Component Type:	View
Collaborators:	Jobs,studentjobindex.html
Operations:	studentjobs

Component Name:	jobDetail
Design Approach:	Functional
Parent Component:	None
Component Type:	View
Collaborators:	Jobs,JobHits,jobdetails.html,404.html
Operations:	jobDetail

Component Name:	jobDelete
Design Approach:	Functional
Parent Component:	None
Component Type:	View
Collaborators:	Jobs,jobsDeleted.html,404.html,accessdenied.html
Operations:	jobDelete

Figure 15: CRC cards for views

Figure 15 shows the CRC card for a Django view. Each Django view collaborates with Django models that the view accesses and retrieves the data from. Also, the view collaborates with the templates that it loads the retrieved data with. So each Django view is represented as a separated CRC card in PCSE architecture.

4.3 Project Plan

The estimation method originally used in PCSE was based on proxies that represent historical data. This method could not be used to estimate the web application since it required that all proxies represent homogeneous components.

For the web application developed, the estimation method used was based on story points rather than estimation line-of-code-based size. In simple terms, a story point is a measure of complexity. This is to differentiate it from hourly estimates, which are measures of effort. Story points are a relative measure. We defined a user story having a story point value of one (which was the least complex user story) as the base reference for all other stories [22]. We then estimated every other user story relative to this one. For example, if another user story is thrice as complex as the base story, then we give it a value of three points. The difference between complexity and effort is that complexity is a property of the story whereas the effort depends both on the story and the person implementing it.

We took all the user stories from the analysis phase. We took a base story point -- the least complex user story -- and assigned it a value of one point. Likewise, we gave a story point value to every other user story relative to this one. A few changes in story points were made as the project went on. In that case, the user stories were just re-estimated and then developed.

4.4 Iteration Plan

4.4.1 Select feature set

In PCSE, the project is split into several iterations. The idea is to have a working product at periodic intervals. During each iteration, we select a set of user stories or features. We call this set of features, a feature set. This will be the set of features that have to be completed at the end of that particular iteration.

4.4.2 Set iteration goal

The iteration goal is a plan or a set of tasks intended to be achieved by the end of the iteration. Some of the iteration goals were:

- A list of models that need to be developed
- A list of Django views that need to be developed during the iteration
- The list of templates or web pages that need to be constructed
- A list of changes that must be incorporated in the product.

4.4.3 Schedule work

After we know the effort for each iteration, the work can be scheduled using a calendar and tracked using the burn down chart and diary. The calendar, burn-down chart, and dairy are important artifacts in PCSE in order to schedule and track the effort. The detail on how the effort was calculated for each iteration is explained in the next section. Figure 16 shows the Work Breakdown Structure for the iterations in the project.

Work Breakdown Structure

Iteration Number	Planned Effort (minutes)	Cumulative Planned Effort	Planned Velocity	Cumulative Planned Velocity	Planned Completion Date	Actual Completion Date	Actual Effort (minutes)	Cumulative Actual Effort
1	1200	1200	6	6	2/29/2012	2/29/2012	3185	3185
2	4250	5450	12	18	3/16/2012	3/14/2012	2710	5895
3	3839	9289	17	35	3/26/2012	3/26/2012	2870	8765
4	3946	13235	20	55	4/8/2012	4/7/2012	3520	12285
5	3096	16331	18	73	4/21/2012	4/22/2012	3240	15525
6	3738	20069	21	94	5/12/2012	5/12/2012	3650	19175

Figure 16: Work Breakdown Structure for the project

The following figure shows a part of the calendar of the entire project. It includes the calendar for the second and third iterations.

Calendar

Day #	Date	Available Minutes	Cumulative Minutes	Planned Velocity	Cumulative Planned Velocity	
62	3/4/2012	90	3130	1	6	end iteration 1
63	3/5/2012	350	3480			start iteration 2
64	3/6/2012	420	3900	2	8	
65	3/7/2012	250	4150			
66	3/8/2012	220	4370	2	10	
67	3/9/2012	260	4630	2	12	
68	3/10/2012	300	4930			
69	3/11/2012	220	5150	2	14	
70	3/12/2012	200	5350	2	16	
71	3/13/2012	210	5560			
72	3/14/2012	280	5840	2	18	end iteration 2
73	3/15/2012	260	6100			start iteration 3
74	3/16/2012	280	6380		18	
75	3/17/2012	310	6690	3		
76	3/18/2012	270	6960		24	
77	3/19/2012	170	7130	3		
78	3/20/2012	280	7410		28	
79	3/21/2012	320	7730	4		
80	3/22/2012	220	7950		31	
81	3/23/2012	220	8170	3		
82	3/24/2012	260	8430	2	33	
83	3/25/2012	190	8620			
84	3/26/2012	90	8710	2	35	end iteration 3

Figure 17: Calendar for Iteration 2 and 3

The diary for the iterations 2 and 3 of the project is shown in the following Figure 18. It shows the entire breakdown for the corresponding iterations.

Day #	Date	Planned Minutes	Planned Burndown	Actual minutes	Cumulative Actual Time	Actual Burndown	Planned Velocity	Cumulative Planned Velocity	Cumulative Actual Velocity	
1	3/5/2012	400	3850	350	350	3900		0	6	start iteration 2
2	3/6/2012	400	3450	420	770	3480	2	2	8	
3	3/7/2012	400	3050	250	1020	3230		2	8	
4	3/8/2012	400	2650	220	1240	3010	2	4	10	
5	3/9/2012	400	2250	260	1500	2750	2	6	12	
6	3/10/2012	400	1850	300	1800	2450		6	12	
7	3/11/2012	400	1450	220	2020	2230	2	8	14	
8	3/12/2012	500	950	200	2220	2030	2	10	16	
9	3/13/2012	500	450	210	2430	1820		10	16	
10	3/14/2012	450	0	280	2710	0	2	12	18	end iteration 2

Day #	Date	Planned Minutes	Planned Burndown	Actual minutes	Cumulative Actual Time	Actual Burndown	Planned Velocity	Cumulative Planned Velocity	Cumulative Actual Velocity	
1	3/15/2012	300	3539	260	260	3579		0	18	start iteration 3
2	3/16/2012	300	3239	280	540	3299		0	18	
3	3/17/2012	300	2939	310	850	2989	3	3	21	
4	3/18/2012	300	2639	270	1120	2719		3	21	
5	3/19/2012	300	2339	170	1290	2549	3	6	24	
6	3/20/2012	300	2039	280	1570	2269		6	24	
7	3/21/2012	300	1739	320	1890	1949	4	10	28	
8	3/22/2012	300	1439	220	2110	1729		10	28	
9	3/23/2012	400	1039	220	2330	1509	3	13	31	
10	3/24/2012	400	639	260	2590	1249	2	15	33	
11	3/25/2012	300	339	190	2780	1059		15	33	
12	3/26/2012	339	0	90	2870	0	2	17	35	end iteration 3

Figure 18: Diary for iterations 2 and 3

4.4.4 Estimation for first iteration

Since there was no historical data to aid the estimation of the product, the method used for estimation was story point estimation, as explained earlier. For the first iteration, we decided to develop a total of 4 features totaling 6 story points. To calculate effort, we had to make a guess on the amount of time that would be needed to develop a story point. Since the features were quite simple, we gave a value of 200 minutes to develop one story point. But since there was quite a bit of learning curve in the first iteration, the estimation was a bit off. At the end of the first iteration, it took about 354 minutes to develop one story point.

4.4.5 Estimation for future iterations

The estimation for subsequent iterations started to become more accurate as we had historical data to work with after the first iteration. For example, to start with the second iteration, we estimated effort by assigning a development time of 354 minutes (actual effort per story point from iteration 1) per story point. The actual effort for iterations started getting closer to the estimated effort as the project went on.

story points = 17	estimated effort per storypoint = 225.83		
	estimated effort	**actual effort**	
3rd iteration	3839	2870	actual effort per storypoint=168.82
story points = 20	estimated effort per storypoint = 197.32		
	estimated effort	**actual effort**	
4th iteration	3946.4	3520	actual effort per storypoint= 176

Figure 19: Estimated and actual effort calculations

Figure 19 shows the estimated and actual effort calculations for iterations 3 and 4. The third iteration begins with an estimated effort per story point equal to the average of the actual effort per story point from the first two iterations. The fourth iteration takes an average of actual effort from second and third iterations and so on. The story point estimation worked out well as the project went on and we had some historical data.

4.4.6 Estimating the template design and stylesheets

The Auburn University website templates and stylesheets were used to begin with. There were a few modifications made to the stylesheets and the web page templates as the project went on. To do the estimation for the stylesheets, the existing stylesheets were used as a baseline and the new styles were estimated relative to the existing ones.

Since we used the already built-in templates and stylesheets as the baseline, we estimated the size of the new templates and stylesheets rather than the effort. Again story-point based estimation was used to estimate the size of the new stylesheets and templates.

4.5 Construction

Construction was done using TDD. There were three different kinds of components, models, templates and views when developing the web application in Django. There were different ways to write tests and the production code for each of three component types.

4.5.1 Models

Since models are Python classes, the traditional method of testing classes can be applied to testing the models. Below is an example test case for a model named Jobs.

```
def test_jobsAddTest(self):

        self.job=Jobs.objects.create(title='TestJob',description='This is a test
        job',contact_info='334-332-9561')
         self.assertEquals(self.job.get_title(), 'TestJob')
```

The above test will fail if the model named two parameters to the assert function that do not match. We just create a new object and test it within the test case. What follows is a way to test if the actual data exists in the database table.

```
def test_jobsAddTest(self):

        self.job=Jobs.objects.get(pk=1)
         self.assertEquals(self.job.get_title(), 'TestJob')
```

The above test will fail if the corresponding object for the model “Jobs” does not exist in the database. If it fails, then you have to insert the tuple into the database and then run the test again to make it pass. Also, the member function get_title() has to be implemented in the model “Jobs” and it should

return the title of the Jobs object. All these have to implemented in the production code to make the test pass. In this way, each and every function and variable for the models were constructed in the production code by writing failing tests and making them pass.

4.5.2 Views

Django's test client was used to test the views of the web application. Some of the things the test client lets us do are [19]:

- Simulate GET and POST requests on a URL and observe the response -- everything from low-level HTTP (result headers and status codes) to page content.
- Test that the correct view is executed for a given URL.
- Test that a given request is rendered by a given Django template, with a template context that contains certain values.

When retrieving pages, we need to specify the *path* of the URL and not the entire address. Here is an example of how the test client works.

```
c=Client()
response = c.get('/logout/')
self.assertEquals(response.status_code, 302)
reponse.content
```

The above test case instantiates a test client and tries to make a call to the URL '/logout/'. The test client looks for URL matches in the URL configuration of the web application. If it finds a match, it passes the test and executes the corresponding view and retrieves the web page specified by the view. The status code 302 here denotes that the view returns an HttpResponseRedirect object, which is just a page redirect. A status code 200 means the view returns an HttpResponse object. The response.content prints out the content of the web page that is retrieved by the view. In our case, the response.content will contain HTML code of the web page that is being retrieved. It looks something like,

'<!DOCTYPE html...'

……

We can also make a POST request on the provided path and return a Response object, which is documented below. The key-value pairs in the data dictionary are used to submit POST data. For example:

c = Client()

response=c.post('/login/', {'username': 'prabhu', 'password': '111111'})

...will result in the evaluation of a POST request to this URL:

/login/

...with this POST data:

username=prabhu&password=111111

A status code of 200 after the execution of this post means a successful login and anything else is an incorrect username/password combination.

Similarly, we can make a GET request on the provided path and return a Response object, which is documented below. The key-value pairs in the data dictionary are used to create a GET data payload. For example:

c = Client()

response=c.get('/jobdetails/', { 'id': 7})

...will result in the evaluation of a GET request equivalent to:

/jobdetails/?id=7

Submitting files is a special case. To POST a file, we need only provide the file field name as a key, and a file handle to the file we wish to upload as a value. For example:

c=Client()

f=open(‘…/file.pdf’)

c.post(‘/addresume/’, {‘attachment’: f})

f.close()

All the URLs in the URL configuration of the web application and all the views were tested using the test client.

4.5.3 Templates

Django doesn’t provide a way to test the content of web pages directly. We can test the template ‘context’ instance that was used to render the template that produced the response content. If the rendered page used multiple templates, then context will be a list of Context objects, in the order in which they were rendered. Regardless of the number of templates used during rendering, we can retrieve context values using the [] operator [19]. Here is an example:

response = client.get('/studentprofile/')

response.context['name']

will result in

'**Prabhu**'

The content of the web page can be tested by calling the response.content as explained in the previous section. By calling the response.templates, we can get a list of template instances used to render the final content in the order they were rendered. For each template in the list, we can use template.name to get the template's file name, if the template was loaded from a file. (The name is a

string such as 'templates/index.html'.) All templates were tested for context values using this method of the test client.

4.6 Review

Review is a phase where we examine the test code for coverage. This is the phase where we ensure enough black box tests and white box tests are constructed. If the test code is not covered, then we can add tests appropriately and fix it via TDD. Code review was done at the end of each iteration to make sure some of the common bugs from previous iterations were fixed early. A review checklist is constructed in the planning phase that has some of the common mistakes that were made in earlier iterations. This review checklist is then updated during each iteration plan. The following figure shows a couple of items in the review checklist and the number of times that defect was found during the review phase.

Defect Pareto Analysis (taken from all defects to date)

Type	Description	Count of Occurrences
Documentation	Flaws in comments	
Build	Problems with environment, compiler, build activity	
Product syntax	Syntax flaws in the deliverable product	31
Product logic	Logic flaws in the deliverable product	73
Product interface	Flaws in the interface of a component of the deliverable product	
Product checking	Flaws with boundary/type checking within a component of the deliverable product	
Test syntax	Syntax flaws in the test code	11
Test logic	Logic flaws in the test code	20
Test interface	Flaws in the interface of a component of the test code	0
Test checking	Flaws with boundary/type checking within a component of the test code	
Bad Smell	Refactoring changes (please note the bad smell in the defect description)	8
	Total	143

Code Review Checklist

Item	Number of times this item detected a defect during review
Does the product satisfy all the requirements stated?	2
Is the testing complete?	3
Have temporary variables been used and cleaned?	5

Figure 20: Code review

In the review phase, the number of defects that occurred in the project listed on the review list is checked to see if it matches the historical defect count as seen in the above figure. The test code is

checked for correctness and coverage. The test code was reviewed and it was made sure that it tests the entire production code. Also, it was made sure that all the test cases in the test case pass without any bugs. In case of any missing test cases, new test cases were written and fixed through TDD.

4.7 Refactoring

Refactoring is the phase where we identify and fix bad smells in the code. This is where we change the non-functional requirements of the software. Refactoring was done at the end of each iteration after the review phase. Some of the bad smells that were found and fixed during the refactoring phase were:

- **Long method –** There were several long Django views that were written during the construction phase. All of them were broken down into smaller, reusable methods during this phase.
- **Duplicate code –** Some of the functionality was repeated in some of the views. This repeated functionality was written as a separate method and then used in the other view. This reduced code redundancy.
- **Temporary fields –** A few temporary variables were used in the views when trying to manipulate the data. These temporary variables were fixed during this phase.
- **Comments –** Comments were inserted into the code wherever necessary to make the code more understandable.

Overall, refactoring was done during each iteration and it ensured that the code was well readable and understandable.

4.8 Integration

Regression tests were done at this phase to ensure that the changes in the code during the previous iteration did not affect the functionality of the software. In other words, we check to ensure that the new or modified code did not introduce new bugs in the software. In cases where there were bugs

introduced, they were also fixed during this phase. Very few bugs were found in this phase and were again fixed via TDD.

4.9 Post Mortem

Post Mortem is the last phase of each iteration. It is done before we prepare to start the next iteration. Some of the things done during the post mortem were:

- Baseline the production code in version control on local file system and test server
- Baseline test code in version control on local file system and test server
- It was analyzed as to what went well and what went wrong during the previous iteration and what could be changed to improve the future iterations.
- One important thing done during this phase was to revisit estimation and see how well the iteration plan was done. It was used to improve the estimation and planning of the subsequent iterations.

4.10 Code complete

Code complete was the last phase in PCSE which marks the end of the development. This includes the completion of the production and test code and the final testing of the system. At the end of this phase, it is ensured that all the requirements have been satisfied by the software.

4.11 PCSE Measures

PCSE has two measures [21]:

1.) The time spent per phase (Timelog).

2.) The defects found per phase (Changelog).

The time per phase is a simple record of the clock time spent in each part of the process. The defects found per phase are a simple record of the specified data for every defect the developer find during compiling and testing.

The reason for gathering both time and defect data is that it will help the developer assess the quality of work and to help plan and manage his projects. These data will show where the developer spends his time and where he inject and fix the most defects. The data will also help the developer see how his performance changes as he modifies the process. The developer can then decide for himself how each process change affects his productivity and the quality of work products.

4.11.1 Timelog:

Timelog is an important artifact used to record the time the developer spends on each project activity.

The following are the important timelog data and their instructions:

- Record all of the time spend on the project
- Record the time in minutes
- Be as accurate as possible
- If the developer forgets to record the starting, stopping, or interruption time for an activity, promptly enter the best estimate.

Date: Enter the date when the developer starts working on a process activity.

Start time: Enter the time when the developer starts working on a process activity.

Stop time: Enter the time when the developer stops working on a process activity.

Interrupt time: Record any interruption time (e.g. phone calls, questions or other things) that was spent on the process activity. The reason for the interrupt is usually recorded in the comments section.

Delta: Enter the clock time the developer actually spent working on the process activity, less the

interruption time.

Phase: Enter the process activity (e.g. Analysis, Architecture, Project Plan, Iteration Plan etc.).

Feature Set: Enter the current iteration the developer is in.

Comments: Enter any other pertinent comments that might later remind the developer of any unusual circumstances regarding this activity.

The timelog is useful for the following:

1) Calculating the actual time
2) Evaluating what percentage of the developer's time is spent on each area
3) Determining how much time the developer takes to do a task
4) Measuring productivity
5) Understanding the interruption problem and figure out ways to address it.

The following figure shows a part of the timelog during the development of the web application.

Date	Start Time	Stop Time	Interrupt	Delta	Activity	Iteration	Comments
3/20/2011	10:00 AM	11:00 AM	15	45	Analysis	1	Requirements identified.
3/23/2011	2:00 PM	3:20 PM	25	55	Analysis	1	
4/1/2011	10:00 AM	12:00 PM	55	65	Analysis	1	Analysis ends.
4/10/2011	9:00 AM	10:00 AM	10	50	Architecture	1	Architecture begins.
4/15/2011	10:00 AM	11:00 AM	25	35	Architecture	1	Creating CRC cards
4/20/2011	2:00 PM	3:00 PM	25	25	Architecture	1	Creating CRC cards
4/25/2011	10:00 AM	10:40 AM	15	25	Architecture	1	map component to user stories
4/26/2011	8:00 AM	9:00 AM	30	30	Architecture	1	map component to user stories
8/20/2011	10:00 AM	11:00 AM	10	50	Architecture	1	
8/23/2011	10:00 AM	10:50 AM	10	40	Architecture	1	
8/26/2011	2:00 PM	2:35 PM	5	30	Architecture	1	Architecture ends.
8/30/2011	10:00 AM	11:00 AM	15	45	Planning	1	Planning begins.
9/4/2011	10:00 AM	11:00 AM	25	35	Planning	1	Assigning story points.
9/9/2011	3:00 PM	3:40 PM	10	30	Planning	1	Estimating effort
9/14/2011	10:00 AM	11:00 AM	10	50	Planning	1	Estimating effort

Figure 21: Timelog

4.11.2 Changelog:

The changelog is another important artifact that is used to hold the data on the defects the developer finds and corrects. A defect is counted every time the developer changes a program to fix a problem. The change could be one character or multiple statements.

The change log will record the following basic data:

1.) The defect type.

2.) The phase in which the defect was injected.

3.) The phase in which the defect was removed.

4.) The fix time.

5.) A brief description of the defect.

PCSE uses the PSP defect type standard [21]. Although this standard is quite simple, it should be sufficient to cover most of the developer's needs.

Documentation	**Comments, Message**
Syntax	spelling, punctuation, typos, improper programming language grammar
Build, package	change management, library, version control
Assignment	declaration, duplicate names, scope, limits
Interface	module calls and references, user formats
Checking	error messages, inadequate bound/type/format checking
Data	structure, content
Function	logic, pointers, loops, recursion, computation, functional defects
System	configuration, timing, memory
Environment	programming tools

Table 15: Defect Standard Type

The following are the important Changelog data and their instructions:

General: Record each defect separately and completely.

Type: Enter the defect type from the defect type list. Use the best judgment in selecting which type applies.

Inject: Enter the phase this defect was injected. Use the best judgment.

Remove: Enter the phase during which the developer fixes the defect. This will generally be the phase when the developer found the defect.

Inject Feature set: Enter the iteration this defect was injected.

Remove Feature set: Enter the iteration this defect was removed.

Fix Time: Enter the time the developer took to find and fix the defect. This time can be determined by stopwatch or judgment.

Fix Reference: If the developer injected this defect while fixing another defect, record the number of the improperly fixed defect. If the developer cannot find the defect number, enter an X.

Description: Write a succinct description of the defect that is clear enough to later remind the developer about the error and help the developer to remember why he made it.

The changelog is useful for the following:

1.) Deciding where and how to improve the developer's process.
2.) Observing fix time data can help the developer evaluate the cost and schedule consequences of his personal process improvements.
3.) Analyzing defects for pattern.
4.) Preventing defects.
5.) Creating a review checklist for commonly occurring defects.

The following figure shows a part of the changelog that was obtained during the development of the web application.

Date	Type	Inject Activity	Inject Iteration	Remove Activity	Remove Iteration	Fix Time	Description
2/9/2012	Product logic	Construction	1	Construction	1	15	Error adding jobs.
2/12/2012	Test syntax	Construction	1	Construction	1	10	
2/12/2012	Test syntax	Construction	1	Construction	1	15	Error creating test database.
2/14/2012	Test logic	Construction	1	Construction	1	10	Error destroying test database
2/14/2012	Test syntax	Construction	1	Construction	1	10	
2/14/2012	Test logic	Construction	1	Construction	1	15	
2/29/2012	Test logic	Review	1	Review	1	5	Incomplete testing.

Figure 22: Changelog

5. Conclusion and Future Work

5.1 Summary

The following are a few important points that summarize the thesis:

- The web application was developed using the PCSE process. A few modifications were done in the different phases and artifacts of PCSE to fit the artifacts produced during the development process.
- The web application was developed in Django, a web application framework written in Python.
- The traditionally used proxy based estimation in PCSE process was not used here. A new story point based estimation method was used to develop the web application. This showed how different planning techniques can be used to develop different kinds of applications.
- The web application is expected to be used by the faculty, staff and students of the Computer Science and Software Engineering department at Auburn University for their on-campus job requirements.

5.2 Conclusion

This study was intended to apply PCSE in a web development environment. This thesis also gave a good measure of how well PCSE can be adapted to develop Web based applications. This study also provided a way to document the PCSE lifecycle and to explain the different activities of PCSE. The following are the major conclusions of this study:

- This work showed that although PCSE was used traditionally to develop conventional applications, it could be adapted to develop web applications.

- This work also demonstrated how PCSE phases and artifacts were adapted to accommodate the unique aspects of web application development.
- PCSE has been traditionally designed to fit homogenous components. But that was not the case with the web application developed. The web application had non-homogenous components. This thesis illustrated how PCSE was adapted to fit in these non-homogenous artifacts.
- Different planning techniques were used to estimate the size and effort of the different components developed. This showed how more than one planning technique could be employed within the same project depending on the component type.
- This work introduced a new estimation method into PCSE – the story point based estimation method which was based on complexity rather than effort.
- A new architecture technique that fits the MVC pattern of web applications was introduced.
- This work also showed how TDD could be applied in a web development environment to develop a web application.

5.3 Future Work

Although the modifications applied to PCSE worked really well during the development of the web application, there are a few more enhancements that can be done:

- First, there were several changes made to the PCSE process during each phase of development. The changes and the techniques used worked very well and they were highly efficient in the development of web application. But, different techniques could be applied during the different phases to develop more web applications in the future using PCSE and this could in turn give way to more efficient techniques.

- The web application was developed in Django. The components in Django were models, templates or views. In the future, PCSE can be used to develop more web applications in other web application frameworks. Different web application frameworks have different structure and hence, different component types. So, developing web applications in another platform would pose a different challenge and might require more changes to the techniques involved in PCSE process to accommodate the changes in artifacts.

References

[1] Software Process course notes, Fall 2010, By Dr. David A. Umphress, Computer Science and Software Engineering Department, Auburn University.

[2] "Heavyweight vs. lightweight methodologies" (2002, December 03), In Builder.com, By Jason P. Charvat, from http://www.builderau.com.au/strategy/projectmanagement/soa/Heavyweight-vs-lightweight-methodologies/0,339028292,320270383,00.htm

[3] "Django (web framework)". (2012, June 1). In Wikipedia, The Free Encyclopedia. Retrieved 18:30, June 2, 2012, from http://en.wikipedia.org/wiki/Django_(web_framework)

[4] Django documentation by 2005-2012, Django Software Foundation – https://docs.djangoproject.com/en/dev/faq/

[5] "Agile Software Development". (2012, July 5). In Wikipedia, The Free Encyclopedia. Retrieved 00:19, July 6, 2012, from http://en.wikipedia.org/wiki/Agile_software_development

[6] "Personal Software Process". (2012, June 12). In Wikipedia, The Free Encyclopedia. Retrieved 00:50, July 6, 2012, from

http://en.wikipedia.org/wiki/Personal_software_process

[7] A Discipline for Software Engineering, by Watts S. Humphrey.

[8]Haroon Altarawneh and Asim El Shiekh. 2008. A Theoretical Agile Process Framework for Web Applications Development in Small Software Firms. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*(SERA '08). IEEE Computer Society, Washington, DC, USA, 125-132.

http://www.cin.ufpe.br/~hsf/Referencial%20Teorico/A%20Theoretical%20Agile%20Process%20Framework%20for%20Web%20Applications%20Development.pdf

[9] What is Correlation Coefficient, By 1997-2005. Financial Forecast Center LLC - http://forecasts.org/cc.htm

[10] IBM Rational Unified Process. (2012, July 9). In Wikipedia, The Free Encyclopedia. Retrieved 20:30, July12, 2012, from –http://en.wikipedia.org/wiki/IBM_Rational_Unified_Process

[11]Burn down chart. (2012, July 3). In Wikipedia, The Free Encyclopedia. Retrieved 22:40, July 12, 2012, from - http://en.wikipedia.org/wiki/Burn_down_chart

[12]Test-driven development. (2012, June 28). In Wikipedia, The Free Encyclopedia. Retrieved 17:27, July 11, 2012, from - http://en.wikipedia.org/wiki/Test-driven_development

[13]Test Automation Code Review Guidelines, By Devin A. Rychetnik, Microsoft Inc. - http://msdn.microsoft.com/en-us/library/ff519670.aspx

[14] Test Reviews Vs. Code Reviews - Some Helpful Tips, By Roy Osherove - http://weblogs.asp.net/rosherove/archive/2007/03/13/test-reviews-vs-code-reviews-some-helpful-tips.aspx

[15]Code refactoring. (2012, June 12). In Wikipedia, The Free Encyclopedia. Retrieved 12:57, July 12, 2012, from -http://en.wikipedia.org/wiki/Code_refactoring

[16]Regression testing. (2012, May 28). In Wikipedia, The Free Encyclopedia. Retrieved 13:41, July 12, 2012, from - http://en.wikipedia.org/wiki/Regression_testing

[17]Django (web framework). (2012, June 27). In Wikipedia, The Free Encyclopedia. Retrieved 20:51, July 12, 2012, from -http://en.wikipedia.org/wiki/Django_(web_framework)

[18]Django at a glance- Django documentation by 2005-2012, Django Software Foundation - https://docs.djangoproject.com/en/dev/intro/overview/

[19] Testing Django Applications- Django documentation by 2005-2012, Django Software Foundation - https://docs.djangoproject.com/en/dev/topics/testing/?from=olddocs

[20] Introduction to User Stories, by 2003-2009 Scott W. Ambler - http://www.agilemodeling.com/artifacts/userStory.htm

[21] PSP – A Self-Improvement Process for Software Engineers, by Watts S. Humphrey

[22] What is Story Point Estimation, by 2007 Silver Stripe Software Private Ltd.

http://toolsforagile.com/blog/archives/155/what-is-story-point-estimation