

2014-04-28

# Software Engineering in Small Projects: The Most Essential Processes

Luis T. Nunez

*University of Miami*, [lncautus-um@yahoo.com](mailto:lncautus-um@yahoo.com)

Follow this and additional works at: [https://scholarlyrepository.miami.edu/oa\\_theses](https://scholarlyrepository.miami.edu/oa_theses)

---

## Recommended Citation

Nunez, Luis T., "Software Engineering in Small Projects: The Most Essential Processes" (2014). *Open Access Theses*. 478.  
[https://scholarlyrepository.miami.edu/oa\\_theses/478](https://scholarlyrepository.miami.edu/oa_theses/478)

This Open access is brought to you for free and open access by the Electronic Theses and Dissertations at Scholarly Repository. It has been accepted for inclusion in Open Access Theses by an authorized administrator of Scholarly Repository. For more information, please contact [repository.library@miami.edu](mailto:repository.library@miami.edu).

UNIVERSITY OF MIAMI

SOFTWARE ENGINEERING IN SMALL PROJECTS: THE MOST ESSENTIAL  
PROCESSES

By

Luis Teófilo Núñez Degwitz

A THESIS

Submitted to the Faculty  
of the University of Miami  
in partial fulfillment of the requirements for  
the degree of Master of Science

Coral Gables, Florida

May 2014

©2014  
Luis Teófilo Núñez Degwitz  
All Rights Reserved

UNIVERSITY OF MIAMI

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

SOFTWARE ENGINEERING IN SMALL PROJECTS: THE MOST ESSENTIAL  
PROCESSES

Luis Teófilo Núñez Degwitz

Approved:

\_\_\_\_\_  
M. Brian Blake, Ph.D.  
Vice Provost for Academic Affairs,  
Department of Computer Science

\_\_\_\_\_  
Dilip Sarkar, Ph.D.  
Associate Prof. of Computer Science

\_\_\_\_\_  
Iman Saleh Moustafa, Ph.D.  
Assistant Scientist, the Graduate School

\_\_\_\_\_  
M. Brian Blake, Ph.D.  
Dean of the Graduate School

NUNEZ DEGWITZ, LUIS TEOFILO

(M.S., Computer Science)

Software Engineering in Small Projects:  
The Most Essential Processes

(May 2014)

Abstract of a thesis at the University of Miami.

Thesis supervised by Professor M. Brian Blake.

No. of pages in text. ( 71)

Although frequently viewed as bureaucratic and inefficient, some software engineering processes, in particular unit testing, may prove to be not only useful but indispensable for small projects. Software-related small businesses or “startups” often do not know which software engineering processes and tools are most effective or even those that are absolutely required. In addition, they usually have significant time constraints and limited resources. As a result, it is very common for startup businesses to overlook and omit the use of many vital processes and/or tools, without realizing that such omissions could negatively impact their project, financially, at present and many years into the future.

This thesis surveys and evaluates relevant business processes for software engineering in small enterprises including requirements engineering, infrastructure selection, and testing alternatives. Consequently, this work provides important decision support guidelines when selecting software processes that will ultimately result in robust, reliable, scalable, and maintainable software. This thesis is the first step towards developing repeatable techniques for selecting an appropriate set of processes and tools to be used for new, small-scale software projects. Within this work is a focused experiment that demonstrates how to effectively leverage unit testing techniques for small projects. The results of this model are evaluated within a software development experiment where an

existing software product (that did not initially consider formal software engineering techniques) is redeveloped to incorporate unit testing paradigms. The outcomes of this experiment include the evaluation and comparison of software quality and an assessment of level of effort to produce the existing product as it relates to the unit testing-enhanced product.

From the experimentation it was found that even though the unit tested code has approximately twice the code lines as the version without unit testing, the total time to develop the unit tested version was only 33% greater than the untested version. In addition the qualitative analysis showed that the tested version was superior in terms of reliability, maintainability, and scalability.

## ACKNOWLEDGEMENTS

I would like to give special thanks to Professor Brian Blake who almost every two weeks dedicated a considerable amount of his very busy time to oversee the progress of this work and to review the resulting document.

I would also like to give special thanks to Dr. Dilip Sarkar, who apart from being my academic advisor for over 2 years, was my professor in a parallel computing course. In that course, Professor Sarkar did not limit himself to teach how to efficiently develop an application for a parallel processing environment, but also how to think about optimization in general for any kind of environment.

In addition, I would like to thank Dr. Iman Saleh Moustafa since besides from accepting the responsibility of being part of my thesis committee also spent considerable time listening about the progress of this work and giving accurate feedback and recommendations to help with the work.

And last, but not least I would like to recognize Dr. Burt Rosenberg, from whom in the last few years I have learned so many things that I won't even attempt to enumerate them.

## TABLE OF CONTENTS

	Page
LIST OF FIGURES .....	v
LIST OF TABLES .....	vi
 Chapter	
1 INTRODUCTION .....	1
2 REQUIREMENTS ENGINEERING .....	3
3 SOFTWARE PROCESS METHODOLOGY .....	9
4 IMPLEMENTATION PLATFORM .....	13
4.1 Choice of programming language .....	13
4.2 Choce of operating system and database management systems .....	17
5 SOFTWARE TESTING .....	18
5.1 About unit testing .....	18
5.2 Advantages of unit testing .....	19
5.3 Disadvantages of unit testing .....	20
5.4 Limitations of unit testing .....	20
5.5 Properties of good tests .....	20
6 CASE STUDY AND EXPERIMENT: UNIT TESTING.....	22
6.1 Experiment 1: The re-implementation of Find-it .....	22
6.1.1 Hypothesis .....	22
6.1.2 The evaluation method .....	22
6.1.3 The procedure .....	27
6.1.4 Quantitative results .....	29
6.1.5 Qualitative results .....	32
6.2 Experiment 2: Adding unit testing to existing software .....	34
6.2.1 Hypothesis .....	34
6.2.2 Validation .....	34
6.2.3 Description of the experiment .....	34
6.2.4 Results .....	36
7 DISCUSSION AND CONCLUSION .....	44
 REFERENCES.....	 48
APPENDIX A.....	51



## LIST OF FIGURES

Figure 2.1: Sample use case .....	6
Figure 3.1: The Cockburn Scale .....	11
Figure 4.1: Programming language selection .....	17
Figure 5.1: Simplified process for performing unit tests .....	21
Figure 6.1: Quick-see architecture .....	24
Figure 6.2: Quick-see class diagram .....	25
Figure 6.3: Find-it arcitechture .....	26
Figure 6.4: List of PHP files for Find-it .....	27
Figure 6.5: Comparative line count .....	30
Figure 6.6: Percentage of total coding hours .....	31
Figure 6.7: Sample user interphase .....	35

LIST OF TABLES

Table 6.1: Quantitative results for experiment 1 ..... 29

# Chapter 1

## Introduction

Although contemporary business owners sometimes suggest that formal engineering processes can be an inefficient practice, software engineering processes, in particular Unit Testing, continue to be both useful and indispensable for small projects. Many software-related startup businesses are not aware of which software engineering processes and tools are really needed and why they are important. Small businesses tend to have limited time and resources and, as such, it is very common that these organizations overlook and omit the use of many vital processes and/or tools. These omissions could prove fatal, or at a minimum very costly, to the underlying projects.

The intention of this thesis is to demonstrate the need and advantages of a set of software engineering-related processes, particularly in small businesses. This work provides guidance to small companies on how to apply the most relevant best practices of software engineering in a way that is cost effective, while reducing delays to their projects. The ultimate result is robust, reliable, scalable, and maintainable software. To achieve this goal, this work will be divided in two parts. In Chapters 1 through 5, the thesis will introduce several software engineering processes core to software lifecycles, but sometimes neglected by entrepreneurs. This section of the work will be based on studying the criteria established by others for selecting software engineering processes and tools. Each of the Chapters 1 through 5 will briefly present the process required based on what it consists of and recommendations with respect to how to implement.

The second part of this work, Chapter 6, described two software development experiments and corresponding results. Both experiments describe the benefits and level of effort involved in developing software that contains Unit Testing. The objective of the experiments is to empirically evaluate the effectiveness of Unit Testing for small projects and the effort involved in introducing tests to existing software.

The first experiment incorporates unit testing into an existing Web application, called Find-it, which was initially developed without the use of testing. In this experiment, Find-it gets updated to incorporate unit testing without performing major refactoring to the existing code. This experimentation is conducted to evaluate the level of effort for introducing unit testing in similar applications and the corresponding benefits.

In the second experiment, the Find-it application is re-implemented altogether. Since Find-it was developed in a controlled environment where the thesis's author was one of the main developers, detail information is available on how it was developed and the effort that it took. Therefore, by re-implementing the application, also in a controlled environment, it is possible to perform a comparison between both of the preliminary software and resulting outcome.

# Chapter 2

## Requirements Engineering

Having a well-defined set of requirements, the Software Requirement Specification (SRS), is a critical software engineering element within an engineering project. Nothing can be more threatening for a software project than having ineffective requirements. Improper requirements are a major cause of unscheduled changes that will normally result in cost overruns and/or project overdue.

According to the study in [1], 80% of the deficiencies in delivered software may be attributed to problems in requirements engineering. Poor requirements can cause what Gilb et al. called “The evil circle principle”, that is described as follows [2]:

If requirements are unclear, incomplete or wrong, then the architecture will be equally wrong.

If the architecture is wrong, then our cost estimates will be wrong.

If the cost estimates are wrong, then people will know we are badly managed.

If the high-level requirements and architecture are wrong, then the detailed design of them will be equally wrong.

If the detailed designs are wrong, then the implementation will be wrong.

So we end up re-doing the entire project as badly as the last time, because somebody will cover up the initial failure, and we will presume that the methods we used initially were satisfactory.

Gathering, analyzing, and modeling requirements are not easy tasks, but they need to be done effectively while limiting irrelevant information. In fact, irrelevant information can have an adverse effect on a project [3]. For example, including unreasonable expectations from the stakeholders regarding the required time to develop it could influence the time estimation into set unrealistic deadlines which could later affect morale. A good SRS is nothing more than a comprehensive description of what the

stakeholders anticipate that the end product should be. Mostly, requirements are written in plain natural language like English, and with some graphic representations of user and other system interfaces, and a few other illustrations and diagrams. Of course, for it to be comprehensive it should describe in detail the desired functionality, performance and design constraints, and the external interfaces [4].

According to the IEEE guideline 830 [5], a good SRS should be:

- a) Correct;
- b) Unambiguous;
- c) Complete;
- d) Consistent;
- e) Ranked for importance and/or stability;
- f) Verifiable;
- g) Modifiable;
- h) Traceable.

Since the quality and completeness of the SRS can have a significant impact in the success of the any software project, it is imperative to give the proper time and resources to the SRS elaboration. The basic steps [2], [5]–[10] are as follows.

#### 1. Elicitation

This is a phase of the process where information is gathered from stakeholders and users regarding every aspect of what the software is desired to be, and should include the following.

- a. High level requirements (Vision & scope). This will form the basis for the introduction and overall description sections of the SRS.

- i. Opportunity/needs - Why this software is needed.
  - ii. Justification - What benefits the new software will bring to the stakeholders.
  - iii. Scope - What the software will do and what won't do.
  - iv. Major constraints - Thing to which the development is to be bounded.
  - v. Major functionality - Description of all the functions that the system will perform.
  - vi. Success factor - Definition of what would be considered a successful project.
  - vii. User characteristics - Background information of the intended users.
- b. Low level requirements (can be done by interviews, Observation of people working, Discussion summaries or questionnaires). The low level requirements should address most of the following areas.
- i. Individual functionality. Use Cases - Detailed description of each functionality. This is usually done in a tabular form using plain natural language. Of all the parts of an SRS, the use cases description, or an equivalent detailed description of the functionality, is the most important part of the document. It is imperative to anyone not to start a project, without having a well-defined use case description. The following is an example of a simple but effective way to describe a use case.

Name:	Log In to account.
Code:	FR LogAcc.
Actor(s):	Any active user.
Precondition:	The user has arrived to the login page.
Trigger:	The user enters a user and password in the proper fields and submits it.
Basic Path:	The user can enter a user and password to login to his account and submits the information.
Alternative Path:	After submitting username and password, the user is not authenticated and is redirected to the login page which now shows that access was denied and gives an option to try again.
Exception paths:	The system fails to connect to the database, in which case the user is informed of the problem and is invited to try again at a later time.
Other:	One username can be associated with different accounts, but the combined username and password defines the account to which the user is login in. Therefore every user and password combination must be unique.

Figure 2.1:Sample use case

- ii. Use Case Diagrams - The functionality description can be complemented with one or more Use Case Diagrams.
- iii. Business flow - This is a description of how the users perform the desired functionality.



- iv. Data, format, and information needs. - A description of how the system receives its (input) and how the output will be formatted.
- v. User interfaces - Description and or graphical representation of the user interfaces.
- vi. Interfaces with other systems - If the system is to interact with other systems, the interfaces for such an interaction need to be described.
- vii. Other constraints - performance, reliability and security.

## 2. Analysis

Once all the requirements are gathered, the data may be disorganized and requires analysis to categorize and prioritize it [6].

- a. Categorizing - An intuitive and effective method for categorizing requirements is to group them as being functional, nonfunctional, interface, or data format requirements.
- b. Prioritizing - In most cases, time and resources are limited, so software engineers must decide which requirements should be implemented and others that cannot. Moreover, software engineers must have an in-depth understanding of what to implement first. Requirements are typically assigned a priority value.

## 3. Documentation and definition – Gathered information is typically represented as software requirements specifications (SRS) Draft. SRSs follow the IEEE guidelines 830, IEEE recommended practice for software requirements specifications [5]. For Mission critical systems, more formal methods of specifications may be required where requirements written in natural language are then expressed in machine

readable languages to enable the possible use of computer-aided validation and verification [11].

4. Prototyping – Some software lifecycles suggest a process where a prototype of the system is developed. An increment prototype allows the stakeholders to validate their ideas and that requirements are understood correctly. Software engineers must evaluate if this technique is efficient for the specific application.
5. Review and validation - As the SRS is being prepared, it should regularly be reviewed and validated by the stakeholders and developers.
6. Agreement and acceptance - The final version of the SRS should be accepted by all parties. Subsequently, the SRS acts as a contract between stakeholders and developers that describes what it is that should be developed. Modification and changes after the SRS is finalized must be properly analyzed and risk to the overall project assessed. If a change is approved it should be incorporated within the SRS in a way that leaves a clear trace of what was changed.

# Chapter 3

## Software Process Methodology

A group of professionals working on a software project without following a defined methodology or software process is a *recipe for failure*. According to an article by Linda Dailey Paulson [12].

Although research by The Standish Group found factors such as executive involvement rated as more important in guaranteeing project success, having a formal project methodology rates among the top 10.

With that being said it is also important to note that as the size or density of the methodology increases, productivity decreases at a much faster rate. Therefore adopting a methodology that is most efficient containing an appropriate density for the project is of vital importance. However, one of the problems with selecting a methodology is that there is not one methodology or software process that fits all projects without customization. It is important to evaluate some of the project's characteristics in order to select and adapt a specific methodology.

Before we continue with this topic, it is important to define the software development methodology. Software development methodologies can often be confused with software process models. The fact is that a software process model is only part of the implementation of a methodology. A methodology is concerned with practically every aspect of a software project, including but not limited to things like where should employees sit, how do they communicate, how long do they work, and so on.

Cockburn [13] describes a criteria for selecting a software development methodology for a project. In the literature, related approaches containing concise sets of guidelines for

selecting and/or adapting a methodology, as Cockburn's, are limited. His approach suggests that multiple methodologies are necessary as no one methodology can be appropriate for all projects. In his work, we propose a set of principles to follow when selecting a methodology when multiple methodologies are appropriate and necessary. Interacting methodologies must be differentiated according to staff size and system criticality (more dimensions exist, but these two serve well initially) [13]. There are four main principles [13] when selecting and differentiating software methodologies.

### **Principle 1**

“A large group, need a larger methodology.” The size of the group is defined as the number of people in the group. And the size of the methodology is defined by the number of control elements (i.e. milestones, deliverables, activities, standards, etc.)

### **Principle 2**

“A more critical system—one whose undetected defects will produce more damage—needs more publicly visible correctness (greater density) in its construction.” For this principle, Cockburn divides criticality into 4 zones. From less critical to more critical, the zones are *loss of comfort* (zone C), *loss of discretionary moneys* (zone D), *loss of essential or irreplaceable moneys* (zone E), and *finally loss of life* (zone L).

### **Principle 3**

“A relatively small increase in methodology size or density adds a relatively large amount to the project cost.” Methodology density is defined as the detail and consistency required for the controlling elements.

#### Principle 4

“The most effective form of communication (for transmitting ideas) is interactive and face-to-face, as at a whiteboard.” The software team members should be sitting in close proximity to each other to promote frequent and easy contact. This principle follows his related work in agile software development [14].

Cockburn presents a concise method for selecting a methodology based on the 4 basic principles. Based on principles 1 and 3, Cockburn presented a table to classify projects depending on criticality and size (number of people involved). This table is known as the Cockburn Scale. Below is a simplified version of the Cockburn table.

Table X. The Cockburn Scale.

	1-6	7-20	21-40	41-100	101-200	201-500	...
Life (L)	L6	L20	L40	L100	L200	L500	...
Essential money (E)	E6	E20	E40	E100	E200	E500	...
Discretionary money (D)	D6	D20	D40	D100	D200	D500	...
Comfort (C)	C6	C20	C40	C100	C200	C500	...

Figure 3.1: The Cockburn Scale

#### Recommendations for Small Businesses

Our work is focused on facilitating software engineering practices for small new ventures. As such, we assert that one dimension (i.e. criticality) should be the criteria for selecting the methodology that a project should use.

Cockburn and others [15]–[17] have discussed agile methodologies that gained popularity in the past two decades. It was stated that “Nowadays; agile methodologies are one of the most important rising methodologies in software engineering.” [16].

Unless the project's criticality is Life Threatening (L), a variant of the agile methodology can be adapted for the circumstances of small businesses. XP (Extreme Programming) [13] or one of the methodologies on the Crystal Family of methodologies [13] could be two possible effective options in these environments. To evaluate XP, Cockburn [17] states "XP was first used on D8 types of projects. Over time, people found ways to make it work successfully for more and more people. As a result, I now rate it for E14 projects." XP is based on how a collection of individual practices, such as test driven development, close customer participation, continuous refactoring, constant communication and coordination, collective code ownership, and pair programming, interact in a software development setting [15]. With respect to Crystal Clear, Cockburn [17] states that "Crystal Clear is a methodology for D6-category projects. You should be able to stretch Crystal Clear to an E8 or D10 category project with some attention to communication and testing, respectively." As a result, the agile methodology that is best suited for any project will depend in large part on the culture of the team.

If the project's criticality is life threatening, then the project may need a more sophisticated and/or denser methodology. However, some software engineers assert [18] that some agile methodologies can be extended to satisfy projects with life threatening criticality. Gary K. et al [18] affirm that:

Agile methods are flexible enough to encourage the right amount of ceremony; therefore if safety-critical systems require greater emphasis on activities, such as formal specification and requirements management, then an agile process will include these as necessary activities.

# Chapter 4

## Implementation Platform

The choice of implementation platform components, such as programming languages for the various components, operating system and environment, and pertinent database management systems, is a critical decision for some projects. For example, choosing Microsoft .Net [19] as the development platform for a web based project could be initially appealing considering the rapid development facilities that it offers. However, other factors must be understood considering the expense of the platform. Moreover, at the time of deployment, many low cost hosting servers may not support .Net. To begin this selection, it is imperative to have a well-defined SRS in order to ensure an efficient process, overall.

### 4.1 Choice of programming language

There is no single programming language for all tasks. For that reason, selecting a programming language that is right for the project is crucial [20]. Naiditch [21] states that

Too often, discussions about what language to select break down into emotional appeals for one's favorite language. Instead, a fair and rational process should be established that bases the choice of a programming language on issues related to customer needs and business goals.

The following is a list of criteria that can be used to select a programming language for a project [20]–[23].

- Ease of learning

This factor is important if you have a pre-existing development workforce and require a set of languages unfamiliar to your developers. Otherwise, we assert this factor has a relatively lower priority factor unless the time to production is very restricted.

- Ease of understanding

Software is usually written once, but read many times. In many cases, it will be read by people other than the original developers. This is of particular importance when the system is expected to have a long, useful lifetime. The programming language may play a role, but, in practice, what makes a program easier or harder to understand is closely related to the programming style and best practices that the programmer employs. In general, for a large program, object oriented language facilitate the understanding and reusability, while procedural language are less favored. In these studies, we believe that for very small programs, the opposite could be true.

- Speed of development

Libraries and tools dictate the speed at which you can create production code. For example, contemporary Integrated Development Environment (IDE) with integrated development libraries facilitates the development of Graphical User Interfaces (GUI's). Therefore, selecting a language that has libraries and tools that are adequate for the required tasks could improve speed of development.

- Help with enforcement of correct code

Strongly typed languages prevent runtime errors. Java, [24] C#, and ADA are examples of strongly typed languages. C also requires that all variables have



strong type declarations, but by the use of pointers this requirement can be somewhat overlooked.

- Performance of compiled code

Contemporary compilers for most modern programming languages are optimized to produce efficient code, so this factor is increasingly less relevant. Performance is more heavily determined by architectural issues or algorithmic decisions. Language selection is usually less relevant. There are few exceptions, such as when applications that process large numbers where small variations in performance could translate significantly increased CPU time.

- Portability

Modern programming languages have associated compilers for all the major platforms. However, if the system must be portable across platforms, some languages are more appropriate than other. For example, Java was created to be highly portable across most major platforms [23]. “By far, the most successful example of a popular language that has good portability is Java, which was deliberately designed for portability. It has achieved this by standardizing not only the language, but also the platform environment (J2EE and J2SE).”

- Runtime requirements

The language selected for developing an application may require that the user installs a specific runtime environment. A user may be reluctant to download and install applications that have security and performance risks.

- Ease of collaboration

This is an important factor when the project is developed and/or maintained by teams in distributed locations. In general, object-oriented languages support collaboration more than procedural languages. Hybrid languages and web-based languages like PHP and Python could work equally well for this factor, when software teams limit the development of procedural code but instead uses effective object-oriented practices.

- Adequate match for the task

It is important to find an adequate match for the task, like with speed of development, depend on the nature of the project and the features that it requires. Such features or libraries are not equally accessible to all languages. Also, the original philosophy of the language can play an important role in the decision. “The best choice of language for a task would be according to the original philosophy, keeping in mind that Java is portable web oriented language, Perl is a powerful script language, Python is an easily coded language and C and C++ are efficient languages used in operating systems and drivers” [22]. It is important to examine the SRS and evaluate each requirement while understanding appropriateness of the various candidate languages.

Below is a table of some of the most popular programming languages used today. The author has assigned a score to each language in a scale from 1 to 5 where 5 is the highest score, according to each of the mentioned characteristics. The values presented on this table are for illustrative purpose, readers are encouraged to use their own set of values.

	Java	PHP	Python	C	C++	C#
Ease of learning	2	5	5	4	3	2
Ease of understanding	3	4	4	2	3	3
Speed of development	2	4	4	4	2	2
Enforcement of correct code	4	2	2	3	3	3
Performance	4	2	2	5	5	3
Supported platforms	5	4	4	4	4	4
Portability	5	4	4	3	3	3
Ease of collaboration	5	5	5	3	5	5
Runtime requirements	4	3	3	5	5	5

Figure 4.1: Programming language selection

#### **4.2 Choice of operating system and database management systems**

The selection of the operating system and database management system that a project requires could be a long and complex task that extends beyond the scope of this work. If some quick decisions are required, the author suggests that the requirements should be closely evaluated. The cost of acquisition, cost of maintenance, and availability for deployment are important factors to be considered.

# Chapter 5

## Software Testing

Another important process is a systematic way to incrementally test during the development lifecycle. Failure to do it properly will increase the probability of encountering a number of expensive, and sometimes recurring, complications. There are several forms of software testing. This thesis focuses on unit testing, integration testing, system testing, and acceptance testing. Integration, system, and acceptance testing are perhaps the most prevalent. However, unit testing is more unique, since it is perceived as inefficient by many programmers. First, Elims et al. [25] suggest that the perceived additional time required to use unit testing is far greater than reality.

We examine the available data from three safety-related, industrial software projects that have made use of unit testing. Using this information we argue that the perceived costs of unit testing may be exaggerated and that the likely benefits in terms of defect detection are quite high in relation to those costs.

Properly performed unit testing actually saves time and resources over the lifetime of the project. However, unit testing must be implemented properly and early.

### **5.1 About unit testing**

There are many definitions of Unit testing but a simple and concise one is the following “Testing of individual hardware or software units or groups of related units” [26]. Unit testing is a key component for any software that is intended to be sustained while also being manageable for an extended time. However, as mentioned, unit testing should be taken into consideration at the start of any project, as it is difficult to incorporate later in the project. On the other hand, if unit testing is incorporated into the

development of each unit then the effort and expense is minimized as the integration time of new code is reduced (i.e. reduction in time to debug new code).

## 5.2 Advantages of unit testing

- Higher confidence in the code and less time spent on the debugger.

Beck [27] suggests, in context of the xUnit family of automated testing frameworks, that “It wasn't until I had been automating tests for several years that I noticed that I didn't use the debugger.” .

- Early detection of problems.

Normally unit testing is done either before writing the unit of code, or immediately after. So when a test fails at that stage, either the code for that unit has a bug or the test has a bug. In either case both codes are revised and the software bug is removed.

- Helps prevent the introduction of new bugs during code modifications.

If a modification of the program makes a test fail, the failing test tells us where the problem may be before the code is updated.

- Facilitates integration testing.

By using unit testing, all the individual units are more reliable, so the integration testing becomes less complex thus reducing development time.

- Complements documentation.

Unit testing suites constitute a form of documentation in itself, since the test reveal characteristics that the units should implement.

- Supports design.

If unit testing is done before coding, then the tests become an additional model of the design.

- Greatly reduces the cost of defects in the long run [27].

“...The Defect Cost Increase (DCI). DCI states that the sooner you test after the creation of an error, the greater your chance of finding the error and the less it costs to find and fix the error...”

### **5.3 Disadvantages of unit testing**

A major disadvantage of unit testing is the time investment. The lines of code increase proportionately and require maintenance.

### **5.4 Limitations of unit testing**

- It only tests individual units.

It does not test the integration between the units. It also does not test performance.

- A unit test failure does not prove that the code is faulty.

The fault may be in the test itself.

- Unit testing is very difficult and ineffective with GUIs.

There are frameworks and tools that help unit test GUIs and WEB pages, but in general it is difficult to do it.

### **5.5 Properties of good tests**

An excerpt from a common testing book [28] suggests

Unit tests are very powerful magic, and if used badly can cause an enormous amount of damage to a project by wasting your time. If unit tests aren't written and implemented properly, you can easily waste so much time maintaining and debugging the tests themselves that the production code—and the whole project—suffers. We can't let that happen; remember, the whole reason you're doing unit testing in the first place is to make your life

easier! Fortunately, there are only a few simple guidelines that you need to follow to keep trouble from brewing on your project.

Good tests have the following properties, which makes them A-TRIP:

Automatic  
Thorough  
Repeatable  
Independent  
Professional

As more and more people and companies recognize that unit testing is essential for the sustainability of a software project, unit testing is moving to the requirements phase of the project. The following is an illustration of the typical activities involved in proper unit testing. The illustration was taken from [25].

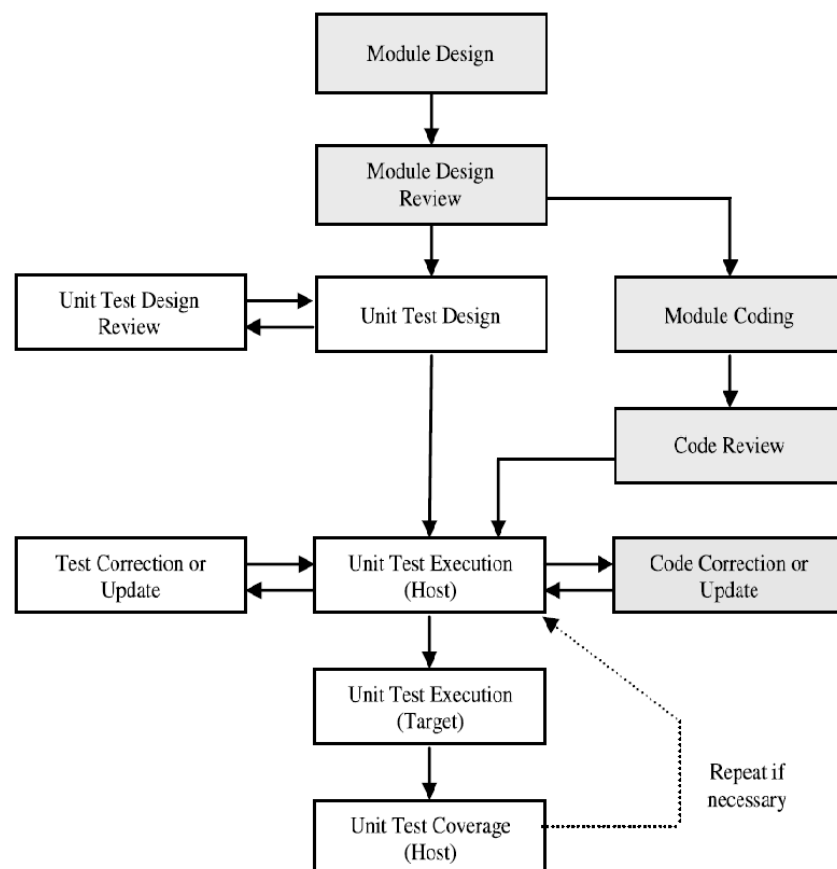


Figure 5.1: Simplified process for performing unit tests, shaded boxes show associated activities that must be completed before or in conjunction with unit testing. [25]

# Chapter 6

## Case Study and Experimentation: Unit Testing

### 6.1 Experiment 1: The re-implementation of Find-it

Find-it, an application that was developed by this author and a group of 4 other team members over the course of a semester was redeveloped into a new product of identical functionality called *Quick-seek*. One of the reasons for selecting Find-it for this experiment was that the author was one of the developers and therefore maintained detailed records related to the time it took to develop each piece. Another reason for choosing Find-it was that it is complex enough to be relevant, but simple enough to be developed in one semester, which was the maximum time available to complete the task of re-developing it in another programming language. In addition, and perhaps most importantly, Find-it was developed to incorporate several software engineering methods, but Unit Testing was not incorporated.

#### 6.1.1 Hypothesis

This thesis claims that if Unit Testing is performed concurrently with development it should not significantly increase the initial developing cost of the project, and that in time the benefits of a self-testing application will exceed the higher cost.



### **6.1.2 The evaluation method**

By redeveloping Find-it, but this time doing it with a test-driven development frame of mind from the initial phases, this experiment intends to demonstrate that even though more than twice the code was written than it was with the original system, the new software, Quick-seeK, did not take much more time or resources to build it, and the quality of the new product is far superior in terms of robustness and maintainability. Quick-seeK was intentionally written in a different programming language and using a different architecture to ensure that rewriting it wouldn't be easier than writing it for the first time.

Quick-seek architecture

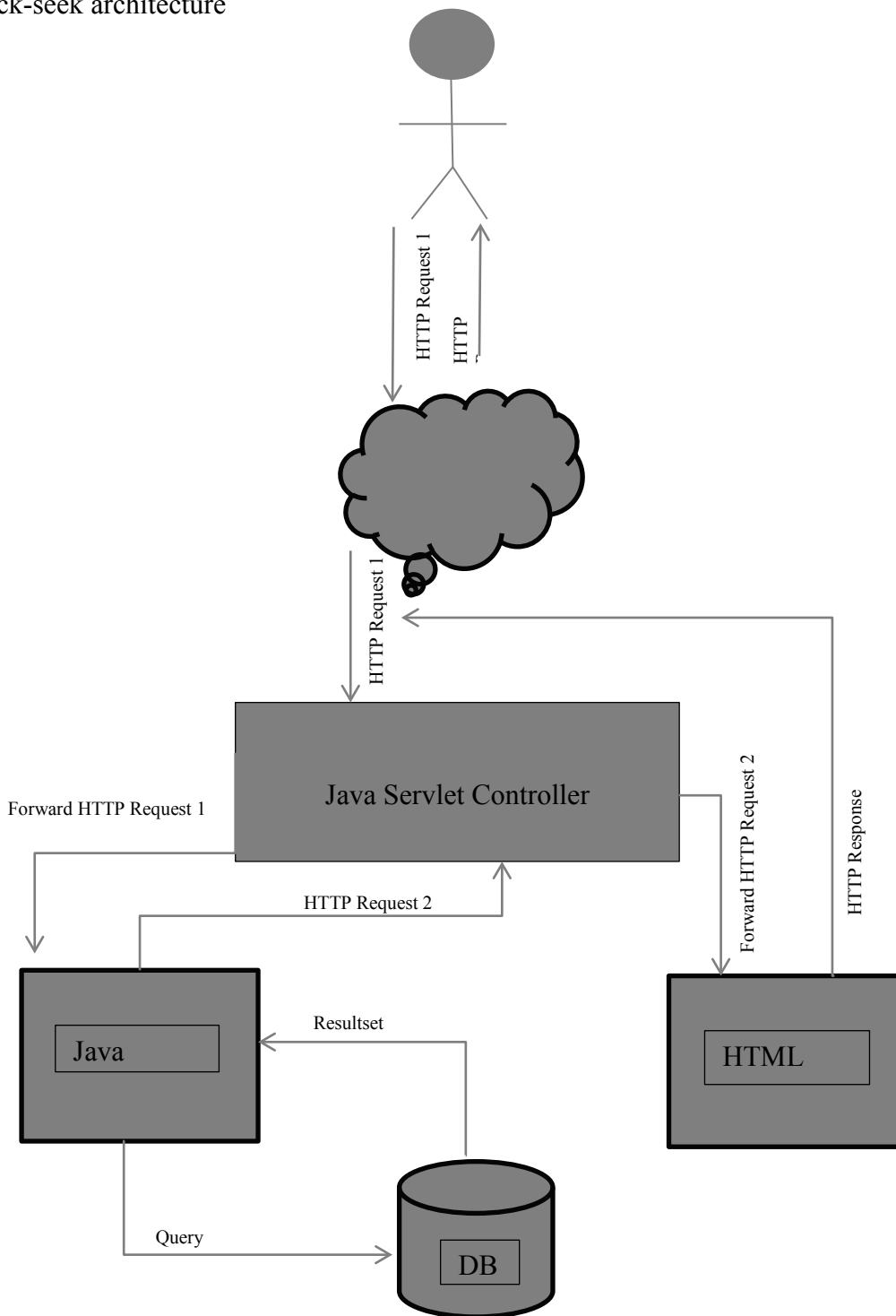


Figure 6.1

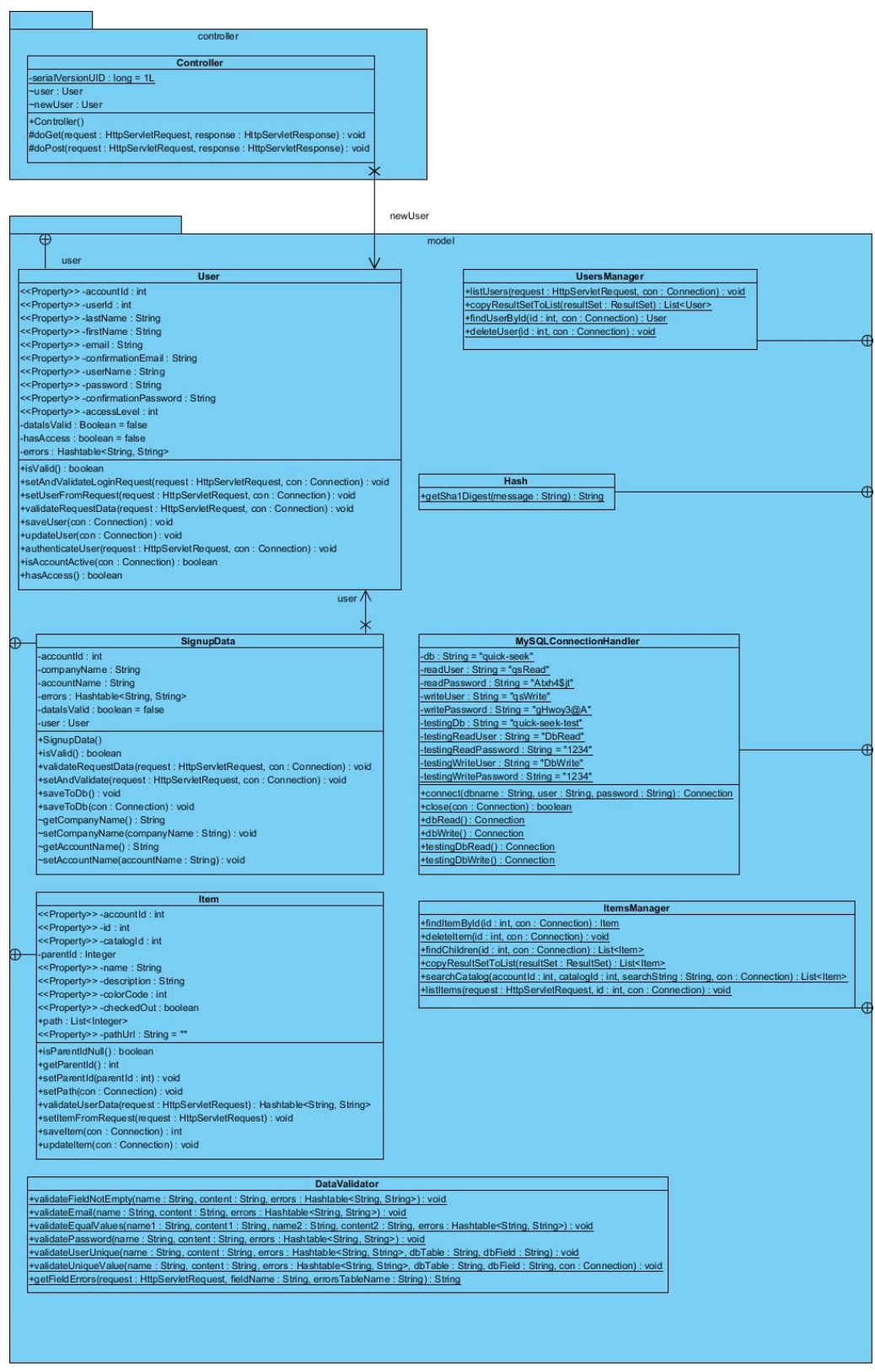


Figure 6.2: Quick-Seek class diagram

Find-it architecture

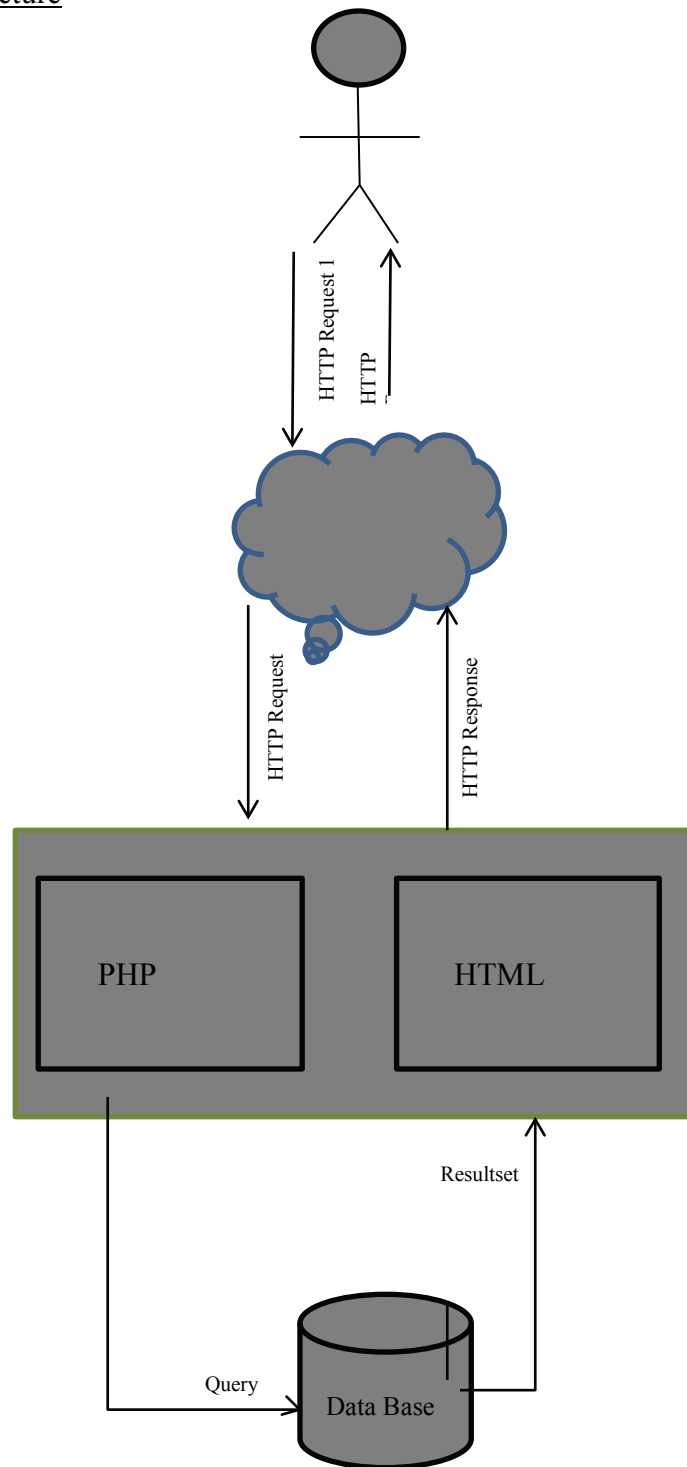


Figure 6.3

Find it does not have a class diagram, since its logic was written as a group of files with procedural PHP code, and therefore it has no packages or classes. The PHP files are enumerated as follows.

ProcessAddItem.php

ProcessEditItem.php

ProcessDeleteItem.php

ProcessSearch.php

ProcessAddCatalog.php

ProcessEditCatalog.php

ProcessDeleteCatalog.php

ProcessAddUser.php

ProcessEditUser.php

ProcessDeleteUser.php

ProcessSignup.php

ProcessAuthentication.php

Restrict\_Access.php

Figure 6.4: List of PHP files for Find-it

### 6.1.3 The procedure

The first step in the experiment was to take the SRS of Find-it and do some minor adaptations to it to use it as the SRS for Quick-seek. The author decided to use the same SRS instead of creating a completely new one, to make sure that Quick-seek, the new product, would be functionally the same as Find-it and therefore comparable.

The second step was to set up a developing environment similar to the one that was used to develop Find-it, but adapted to the new language to be used. Among the most important things included in the developing environment were a version control system, (i.e. SVN), Eclipse as the IDE, and MySQL as the DBMS. The environment also included a Java JDK, Tomcat as the Java container, and JUnit for unit testing. In addition to that an Endeavour Server, which is a freeware web based Project Management System was installed. Due to the small nature of the project, the Endeavor was used very lightly, only to define the major iterations planned and to occasionally check if the development process was on track.

Following the advice presented in this document, since the project is very small, a very simple methodology was adopted, a simplification of Extreme Programming (XP). This was the most significant change in comparison with what was done with Find-it, since Extreme Programming requires developing a unit test for every method or unit of code.

In selecting the implementation platform for Quick-see it was decided to keep everything equal to what was used in Find-it except for the programming language. The decision of changing the programming language was to ensure that the code would be completely written from scratch as opposed to retyping some of the code that was already done. Very soon it became obvious that the change of the language was unnecessary with that respect to the fact that this time the application was going to be built using a test driven development (TDD) methodology. Therefore, it would best be developed with an object oriented model, while Find-it was mainly procedural and therefore very different.

Nonetheless, switching to Java added to the purpose of simulating a completely new project.

The graphic design for Find-it was taken from a free HTML template from FreeCSS.com. It was decided to use the same template for Quick-peek since changing it did not add additional value to the experiment. However, the CSS was modified to change the colors of the UI just so that the two products could be easily differentiated.

#### 6.1.4 Quantitative results

To obtain the following figures we used Code Analyzer, a Java application developed by Mark Teel [29].

	<b>Find-it</b>	<b>Quick- seek</b>
Total Files	52	66
Total Lines	4,116	8,758
Code Lines	3,229	6,075
Functional code lines	3,229	2,803
Test Code Lines	0	3,272
Documentation Lines	524	1,246
Whitespace Lines	368	1,501
Total Coding Hours	146	194

Table 6.1: Quantitative results for experiment 1

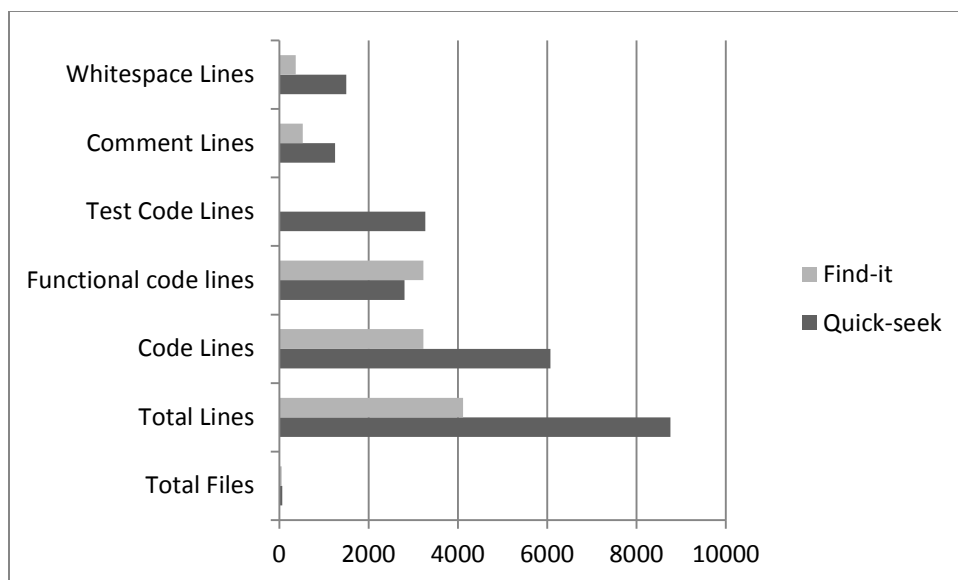


Figure 6.5: Comparative line count

The results show that the redeveloped software has more than twice the lines of code than the Find-it application. Also, more than half of the lines of code for Quick-seek are test code lines. The results were expected based on related work. At first, it appears that it will be hard to justify unit testing, since apparently the overhead caused by it is very significant. However, the average time to develop a line of code without unit testing is not the same as the average time to develop a line with unit testing. On the average, it takes considerable less time to develop a line of code that is tested. Why? There are a several reasons, but two are most relevant: First, even the best programmers usually make mistakes when they write code. As a result, when developing without frequent testing it is very common to spend time debugging the code before it can adequately execute for the first time. Such was the case when Find-it was developed, where many hours were spent debugging. That initial debugging time is part of the total time to develop those lines of code. If the code is unit tested each time a small unit is written then this method eliminates that initial debugging. As a result, by disregarding the time to write the tests, each line of unit tested functional code is written faster on the average than untested



functional code lines. Second, the larger the size of the functional code, the longer it is required to conceptualize the tests. Test lines also require thinking; however, since they are so closely related to the code to be tested, if they are written very close to the time of writing the functional unit of code, then they require considerably less amounts of time than a line of functional code. As to be expected, this difference becomes more apparent if the developer is very experienced in unit testing.

It is necessary to point out that even though it is faster to develop a line of unit tested code, the difference is not enough to compensate for twice the code. As a result, the total time to develop an application using test driven development will undoubtedly still be larger than developing without it. For example, in our experiment, it took 49 more hours to develop Quick-see, which represents only a 33% increase in developing time.

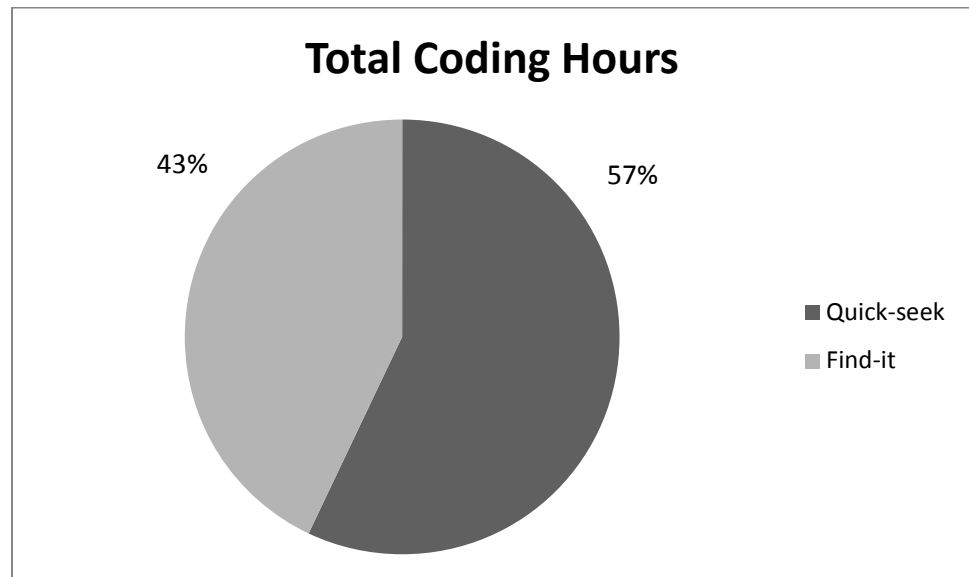


Figure 6.6: Percentage of total coding hours

Nonetheless, time to develop cannot be used in isolation to decide whether or not it is justified to use unit testing. Other metrics might lead to a more holistic conclusion, since

the cost of a software project does not end at the moment when the developers write the last line of the first version.

### **6.1.5 Qualitative results**

In comparing Find-it, the original non-tested application, with Quick-see, it is initially observable that the two products that are functionally identical; however, as the code is analyzed in terms of reliability, maintainability and scalability, it is clear that the quality of the two products is very different.

Regarding reliability, it is notable that with the exception of the HTML code, the controller servlet, and simple accessor methods, every method of the application has an automated test for it. As a result, every time a method or class is added or modified, all of the other classes and methods are tested, which significantly raises the reliability of the product. These tests are simple tests that verify that each small unit of the software is working as anticipated. However, in other cases, some very complex methods may require more exhaustive testing. Because of the built-in suite of automated unit test, Quick-see requires very little integration, and acceptance testing before it was approved with confidence for production. Find-it on the other hand has no built-in testing capabilities which creates uncertainty regarding the reliability of the code. Due to the lack of automated testing, Find-it required extensive manual integration and acceptance testing before it is approved as a production-ready product. During the several iterations of testing, often the product had to be sent back for debugging a defect. These testing and debugging iterations obviously add to the initial cost of development.

Regarding maintainability the situation is similar. Since Quickseek has a built-in suite of automated tests that exercise every method in the application and that can easily be run after any change is made to the software; if Quick-seek is put into production and later a new feature is needed or a change in functionality is required, a developer can work on the software with assurance that if his additions cause new error within the existing code, then it is probable that the test suite will detect it and point out where the problem is. As a result, the updated version can be put into production with confidence with little additional manual testing. On the other hand, the same cannot be said of Find-it which does not have any automated testing to exercise all code. A developer making a change or correcting a bug could very easily introduce a new bug that without extensive manual testing would have a high probability of being unnoticed before the new version goes into production. This kind of situation occurs frequently in commercial products that had not been developed with unit testing, and it is difficult to estimate the related cost. On a similar note, it is worth noting that the main developer of Find-it has used the product for personal purposes almost on a daily basis since it was first published. Since that time, many bugs have been discovered and corrected. If Find-it had been a commercial product, the cost of a customer encountering a problem could have a serious impact on the business, not to mention the additional cost.

When it comes to scalability, again it can be seen that Quick-seek has a great advantage over Find-it. If new features are required, that call for a major design change, the unit test built into Quick-seek makes the refactoring process straightforward and less risky.

## **6.2 Experiment 2: Adding unit testing to existing software**

### **6.2.1 Hypothesis**

If unit testing is skipped, during software development there is a chance that problems will start to show up later on in the life of the software. Unit testing should be incorporated at the start of a new project, particularly for small businesses, as it takes longer to incorporate unit tests later in the project. Alternatively, if unit testing is built into the implementation of each unit, the effort is reduced and less time is spent debugging the new code.

### **6.2.2 Validation**

To validate these assertions, we conducted an experiment where we incorporated unit testing to *Find-it*, an application that was implemented without applying unit testing. The goal is to introduce unit testing without a major refactoring effort. Find-it, is a web based filing management system written in non-object-oriented PHP interfacing with a MySQL database. The experiment was then evaluated quantitatively by tracking the time that it took to create unit tests for most of the PHP scripts that comprise the main logic of the software and comparing that time with the time that took to build the functional code. In addition, an analytical evaluation was conducted by identifying the factors that presented complications to the process.

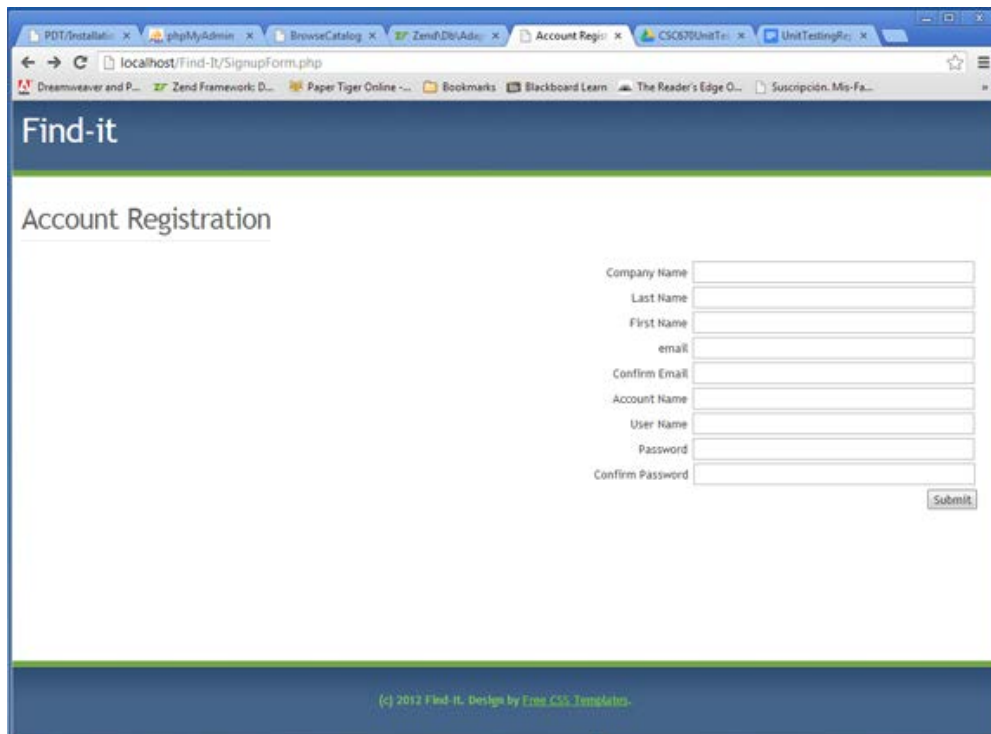
### **6.2.3 Description of the experiment**

After some analysis regarding how to best implement unit testing to Find-it, the initial impression was that it would not be possible to effectively incorporate unit testing into the application without massively refactoring the software. However, after further examination, it is observed that certain aspects of unit testing could be achieved but it

requires some compromise with respect to various characteristics of quality unit testing. The process is also observed to be very time consuming. In particular, developing independent tests were difficult to create from existing code. The analytical results of the experiment are the identification of areas in the original code that made unit test incorporation intractable and the introduction of mitigating solutions (i.e. “workaround solutions”) in this environment.

In the experiment, three major types of problems were discovered and will be described in more detail in the next few paragraphs. It is important to note that this was an isolated experiment performed on a single system. However, useful insights were discovered around the importance of considering unit testing at design time.

Find-It, the system used for the experiment, is made of a collection of HTML forms like the following one.



The image shows a web browser window displaying the 'Find-it' Account Registration form. The browser's address bar shows 'localhost/Find-It/SignupForm.php'. The page has a blue header with the 'Find-it' logo. Below the header, the title 'Account Registration' is displayed. The form contains several input fields: Company Name, Last Name, First Name, email, Confirm Email, Account Name, User Name, Password, and Confirm Password. A 'Submit' button is located at the bottom right of the form. At the bottom of the page, there is a footer that reads '(c) 2012 Find-It. Design by Free CSS Templates.'

Figure 6.7: Sample user interface

In most cases the HTML has embedded PHP code that is standardly executed on server side where final HTML code is sent to the client browser. In many cases this embedded PHP code consists of a line that instructs the parser to include the lines of a specified PHP file before it gets executed, as shown in the piece of code bellow.

```
<?php  
require_once('scripts/account_registration.php');  
?>
```

The previous code fragment is inserted at the beginning of the HTML form; and as a result the code in the file “account\_registration.php” gets inserted into the HTML document before the form gets executed by the server.

In this particular case these inserted files, which contain pure PHP code, is what we wanted to unit test. But as mentioned earlier, doing so still present some challenges.

#### **6.2.4 Results**

On the quantitative side it was observed that it took 153 hours to code the unit tests for Find-it, while it only took 146 hours to develop the functional code. In other words, it took longer to build the tests than it took to build the functional code. This result by it self are not enough to validate our hypothesis since it does not tell us anything about the time that it would have taken if the tests were coded incrementally wile developing the functional code. However, combining this result with the results of experiment 1 we get that the total time to code Find-it plus the time to code it’s unit tests was 299 hours. Therefore, it took considerably longer than the 194 hours that it took to develop Quick-see, a functionally identical application that was developed using test driven development. This combined results do in fact validate our hypothesis.

On the analytical side the following was observed. The two main types of difficulties encountered originate from the fact that in most cases the PHP files have no class declarations or functions that can be called from a test function. The most promising solution is to dynamically include the whole file into the test function. As a result, there are several challenges.

**1. The function may directly send output to the user without storing a result value**

There may be some scripts for which the functionality is just to display (echo) some string or value like in the following script. In those cases, there is no easy way to automatically test the value of the string that gets echoed. When this occurs, the easiest solution may be to do some minor refactoring to the code. Look at the following example.

```
accessLevelOptions.php
<?php
    $sql = 'SELECT accessLevel FROM `users`
        WHERE `userId`=' . $_GET['userId'] . '
        ORDER BY accessLevel';
    $result = $dbRead->fetchAll($sql, $_GET['userId']);
    $admin = 10;
    $user = 20;
    $ruser = 30;
    echo "<option";
    if($result[0]['accessLevel']==$admin){
        echo " selected=selected";}
    echo ' value=' . $admin . ">Administrator</option>";
    echo "<option";
    if($result[0]['accessLevel']==$user){
```

```

    echo " selected=selected";}
echo ' value='. $user . ">User</option>";
echo "<option>";
if($result[0]['accessLevel']==$user){
    echo " selected=selected";}
    echo ' value='. $user . ">Restricted User</option>";
?>

```

In this case, minor refactoring allows the data to be first stored on a variable which we can then use to test (compare with) some known value as was done in the following refactored code.

```

accessLevelOptions.php (Refactored code)
<?php
$sql = 'SELECT accessLevel FROM `users`
      WHERE `userId`= ?
      ORDER BY accessLevel';
$result = $dbRead->fetchRow($sql,$_GET['userId']);
$admin = 10;
$user = 20;
$ruser = 30;
$usersList = "<option>";
if($result['accessLevel']==$admin){
    $usersList = $usersList . " selected=selected";
}
$usersList = $usersList . ' value='. $admin . ">Administrator</option>";
$usersList = $usersList . "<option>";
if($result['accessLevel']==$user){
    $usersList = $usersList . " selected=selected";
}
$usersList = $usersList . ' value='. $user . ">User</option>";
$usersList = $usersList . "<option>";
if($result['accessLevel']==$ruser){
    $usersList = $usersList . " selected=selected";
}

```



```

}
$usersList = $usersList . ' value=' . $user . ">Restricted User</option>";

```

## 2. Performing more than one functionality in the same unit of code

Many scripts perform two or more functionalities synchronously. This eliminates straightforward methods for testing independent functionality separately. Such is the case of the following script.

```

process_delete_catalog.php
<?php
require_once('library.php');
// Check if the catalog has children
$sql = 'SELECT id FROM `containers`
      WHERE parentId = 0 AND dbId = ?';
$result = $dbRead->fetchAll($sql,$_GET['dbId']);
$size_of_result = sizeof($result);
// Delete the catalog if it has no children
if ($size_of_result == 0){
    // The container has no childs
    $dbWrite->delete('databases','dbID='.$_GET['dbId']);
    header('Location: ../DBList.php');
    break;
}

// There are childs, do not delete
header('Location: ../DBList.php?msg=CatalogNotEmpty');
?>

```

As we can see from the highlighted sections of this short script, there are two distinguishable functionalities.

The first functionality that can be found is in the code in “library.php”, which creates a connection with the database and instantiates two objects \$dbRead and \$dbWrite. These two objects contain the methods for reading and writing to the database.

The other functionality deletes a catalog if it has no children. Since the script will run in entirety before an intervention can be asserted, again there are limited approaches to run the script and test functionality without the necessity of using the real database. Using the real database is not desirable for many reasons. For one, the test would be dependent on a successful connection and on specific data being present on it. Secondly, there is a risk of deleting a record, and even leaving that database in an unstable or incorrect state. Fortunately, in this particular case, the insertion of ‘library.php’ was done via a `require_once` and not via `require`. For that reason, it was possible to apply a rudimentary solution to solve one aspect of the problem. A `require_once(‘library.php’)` was inserted in the test before inserting ‘process\_delete\_catalog.php’ to it. Making the `require_once` inside ‘process\_delete\_catalog.php’ inactive since it was already inserted. That solution enables the instantiation of a dummy class of \$dbRead and \$dbWrite before running ‘process\_delete\_catalog.php’ and as a result avoids reading or writing to the real database, as it can be seen in the code below. However, we did not recognize a solution that avoids the initial connection to the database.

DeleteCatalogTests.php
------------------------

```

<?php
require_once 'C:\Program Files\PHP\PEAR\PHPUnit\
    Framework\TestCase.php';
//This makes the second 'require_once('..\scripts\library.php')'
that is inside // 'process_delete_catalog.php' ineffective.
require_once('..\scripts\library.php');

// Scripts that redefine $dbRead and $dbWrite
require_once('DbReadForTestDeleteCatalog.php');
require_once('DbWriteForTestDeleteCatalog.php');

class DeleteCatalogTests extends
    PHPUnit_Framework_TestCase{
    private $dbRead;
    private $dbWrite;

    protected function setUp(){
        $this->dbRead = new DbReadForTestDeleteCatalog();
        $this->dbWrite = new DbWriteForTestDeleteCatalog();
    }

    public function test_no_delete_if_catalog_not_empty(){
        $dbRead = $this->dbRead;
        $dbWrite = $this->dbWrite;
        $_GET['dbId']=1; // This will simulate a catalog with 2
            // entries
        require('..\scripts\process_delete_catalog.php');
        $this->assertfalse($dbWrite->deleteCalled == true,
            "The delete was attempted on a non empty
            catalog\n");
    }
}

```

```

public function test_delete_requested_if_catalog_empty(){
    $dbRead = $this->dbRead;
    $dbWrite = $this->dbWrite;
    $_GET['dbId']=0; //This will result on an empty result
    require('..\scripts\process_delete_catalog.php');
    $this->asserttrue($dbWrite->deleteCalled == true,
        "The delete was NOT attempted on a non
        empty catalog\n");
}

public function test_sql_for_fetch(){
    // This test protects the code from someone messing with
the
    // sql query for the fetch
    $dbRead = $this->dbRead;
    $dbWrite = $this->dbWrite;
    $_GET['dbId']=1; // This will simulate a catalog with 2
    // entries
    require('..\scripts\process_delete_catalog.php');
    $this->asserttrue($result[0]['sql'] == "SELECT id
        FROM `containers`
        WHERE parentId = 0
        AND dbId = ?",
        "The sql was not the expected one\n");
    $this->asserttrue($result[0]['id'] == $_GET['dbId'],
        "The id was not the expected one\n");
}
}

```

**An additional challenge encountered was testing code that is hard to understand**

The script that calculates the path to a container was written by the author a year prior to the experiment. The script is only 23 lines long, and the author required 2 hours to fully understand it enough to apply unit testing to it. If the unit test had been written at the inception of the script then this would have represented a significant saving of time.

To summarize the results of this experiment we have that on the quantitative side, it took 153 hours to code the unit tests for Find-it, while it only took 146 hours to develop the functional code, and the total developing time was far greater than the total developing time of Quick-see which included unit testing. This result, even though originally counterintuitive, was supported by the analytical results where a number of complications were found that substantially added to the time that it took to develop the tests.

# Chapter 7

## Discussion and Conclusion

The motivation for this work was to aid new entrepreneurs of small software project in understanding the need for some of the most essential processes of software engineering. This work is most applicable to small projects where budget and resources are usually very limited. The author assessed and included the areas of software engineering that are most essential to small projects.

Of the practices presented there is one in particular, unit testing, for which there is controversy as to whether its benefits are worth the cost of implementing it. For that reason the experimental part of this work was dedicated to unit testing and consisted of two experiments which attempt to dissipate the controversy about unit testing.

Experiment 1 was conducted to support the hypothesis that if unit testing is performed concurrently with development it should not significantly increase the initial developing cost of the project, and that in time the benefits of a self-testing application will exceed the higher cost. The results of our experiment strongly support our hypothesis. Of the total number of hours that took to code Find-it and Quick-see, only 57% of that time belongs to Quick-see which agree with our hypothesis. The qualitative analysis shows that Quick-see is more reliable, maintainable, and scalable, which suggest that the benefits do in fact outweighs the relatively small cost increase.

Experiment 2 was conducted to support the hypothesis that unit testing should be incorporated at the start of a new project, particularly for small businesses, as it takes longer to incorporate unit tests later in the project. The results of experiment 2 strongly support the hypothesis. When we added the time that it took to build Find-it with the time that it took to implement its unit tests we observed that the total was far greater than the time that it took Quick-see which already incorporated similar amount of unit testing.

Both experiments were designed with small projects in mind, but this author believes that the results are equally valid to projects of any size. The only exception would be extremely small and simple software, less than a page of code without complex logic. As size and complexity increases, so does the value of using test driven developing techniques.

Factors like choice of language or programmers experience should also not affect the results. For example, if Find-it was coded by a very inexperienced programmer and then the experiments were conducted by the same programmer, then all times should have increased proportionally for each case and would lead to the same conclusion.

The second experiment presented in Chapter 6 was conducted with worst case scenario frame of mind. For example, many of the test lines of code accounted for in the statistics as test code were to create mock objects to isolate the test from the rest of the system. Normally this would be done using existing libraries and frameworks that are freely available for that purpose.

As mentioned, related projects [25], [27], [28], [30], [31] and the state-of-the-art agrees on the value of unit testing and curiously most of the resistance to use it originates from those individuals who suffer the most from not taking advantage of it, programmers

and decision makers. Part of this problem comes from lack of proper training. Unit testing, not only should be part of the core courses of any software-related higher education program, but it should be introduced early in the program and then enforced in any programming course. In that way, when the students graduate, they will be accustomed to incorporate it to any piece of software that they write.

Small software related new ventures often cannot afford the cost associated with implementing the full range of software engineering processes and practices. However, sometimes unknowingly, they cannot afford to disregard the most essential practices. If the leaders of a software project want to improve their long-term outlook, at a minimum, they should elaborate a comprehensive SRS, choose and implement a methodology that is appropriate for the size and complexity of the project, carefully select their implementation and deployment platform, and implement procedures to efficiently and continuously test the software. Integration and acceptance testing should be performed at a minimum before every minor release. Unit testing however should be done every time a small unit of code gets added modified or deleted. Moreover, unit testing should not be seen as optional for any project that is expected to last more than a few months. In such cases it should be regarded as an integral part of the project without which the software would be incomplete. However, it should be noted that for achieving the most efficiency, the tests should be written either before each unit of functional code is written, or immediately after.

The quantitative analysis of Experiment 2 can only account for the cost involved during the development phase of the products. It would be interesting to conduct a similar experiment with a commercial software product developed without unit testing and that



has been in the market for at least one year for which there are records of the bugs corrected and the amount of resources that were expended to correct them. The results of such an experiment would be very valuable.

## References

- [1] R. P. Evans, “A Methodological Framework for Requirements Assessment,” 1994.
- [2] T. Gilb and S. Finzi, *Principles of Software Engineering Management*, vol. 4. Addison-Wesley Reading, MA, 1988.
- [3] M. Jorgensen and S. Grimstad, “The Impact of Irrelevant and Misleading Information on Software Development Effort Estimates: A Randomized Controlled Field Experiment,” *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 695–707, Sep. 2011.
- [4] K. Udupa, *Computer Supported Cooperative Work in Specifying Software Requirements*. 1992.
- [5] IEEE Computer Society, Software Engineering Standards Committee, and IEEE-SA Standards Board, *IEEE Recommended Practice for Software Requirements Specifications*. New York, NY: Institute of Electrical and Electronics Engineers, 1998.
- [6] F. Tsui and O. Karam, *Essentials of Software Engineering*, 2nd ed. Sudbury, Mass: Jones and Bartlett Publishers, 2011.
- [7] W. Maalej and A. K. Thurimella, *Managing Requirements Knowledge*. Berlin; New York: Springer, 2013.
- [8] M. Chemuturi, *Requirements Engineering and Management for Software Development Projects*. New York: Springer, 2013.
- [9] O. Vogel, *Software Architecture: A Comprehensive Framework and Guide for Practitioners*, 1st ed. New York: Springer, 2011.
- [10] R. F. Schmidt, *Software Engineering Architecture-driven Software Development*. Waltham, MA: Morgan Kaufmann, an imprint of Elsevier, 2013.
- [11] M. C. B. Alves, D. Drusinsky, J. B. Michael, and M.-T. Shing, “End-to-End Formal Specification, Validation, and Verification Process: A Case Study of Space Flight Software,” *IEEE Syst. J.*, vol. 7, no. 4, pp. 632–641, Dec. 2013.
- [12] L. D. Paulson, “Adapting Methodologies for Doing Software Right,” *IT Prof.*, vol. 3, no. 4, pp. 13–15, 2001.
- [13] A. Cockburn, “Selecting a Project’s Methodology,” *Softw. IEEE*, vol. 17, no. 4, pp. 64–71, 2000.

- [14] M. Fowler and J. Highsmith, "The Agile Manifesto," *Softw. Dev.*, vol. 9, no. 8, pp. 28–35, 2001.
- [15] T. Dingsoyr, T. Dyba, and N. B. Moe, *Agile Software Development Current Research and Future Directions*. Berlin: Springer, 2010.
- [16] A. H. Mohammad and T. Alwada'n, "Agile Software Methodologies: Strength and Weakness," *Int. J. Eng. Sci. Technol. IJEST*, 2013.
- [17] A. Cockburn, *Agile Software Development: The Cooperative Game (2nd Edition) (Agile Software Development Series)*. 2006.
- [18] K. Gary, A. Enquobahrie, L. Ibanez, P. Cheng, Z. Yaniv, K. Cleary, S. Kokoori, B. Muffih, and J. Heidenreich, "Agile Methods for Open Source Safety-critical Software," *Softw. Pract. Exp.*, vol. 41, no. 9, pp. 945–962, Aug. 2011.
- [19] D. Chappell, *Understanding .NET*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2006.
- [20] D. Spinellis, "Choosing a Programming Language," *IEEE Softw.*, vol. 23, no. 4, pp. 62–63, Aug. 2006.
- [21] D. Naiditch, "Selecting a Programming Language for Your Project," *Aerosp. Electron. Syst. Mag. IEEE*, vol. 14, no. 9, pp. 11–14, 1999.
- [22] M. Fourment and M. Gillings, "A Comparison Of Common Programming Languages Used In Bioinformatics.," *BMC Bioinformatics*, vol. 9(1), p. 82, 2008.
- [23] C. Britton, "Choosing a Programming Language," *skyscrapr.net*, Jan. 2008.
- [24] D. Parsons, *Foundational Java*. Dordrecht: Springer, 2012.
- [25] M. Ellims, J. Bridges, and D. C. Ince, "The Economics of Unit Testing," *Empir. Softw. Eng.*, vol. 11, no. 1, pp. 5–31, Mar. 2006.
- [26] IEEE Computer Society and Standards Coordinating Committee, *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York, NY: Institute of Electrical and Electronics Engineers, 1991.
- [27] K. Beck, *JUnit Pocket Guide*. O'Reilly Media. Kindle Edition., 2009.
- [28] D. Hunt, A., & Thomas, *Pragmatic Unit Testing: In Java with JUnit*. Raleigh, N.C.: Pragmatic Bookshelf., 2004.
- [29] M. Teel, *Code Analyzer*. .

- [30] G. Miller and J. Cashion, “Agile Development at Ultimate Software,” 19-Mar-2014.
- [31] K. Auer, *Extreme Programming Applied: Playing to Win*. Boston: Addison-Wesley, 2002.

## APPENDIX A

### QUICK-SEEK SRS

February 24, 2013

#### **Developer**

Luis Teófilo Nunez Degwitz

**Note:** This is an adaptation of the original SRS for a project named Find-it originally developed by

Luis Nunez - Nick Shuman - Sana Khan - Jeff Milinazzo -Sean Meadows - Kin Chan

The original SRS of the Find-it project was intentionally used, with some minor adaptations, in order to develop similar products so that they can effectively be compared to one another.

To avoid page numbering conflicts with this thesis page numbering, the table of content of the SRS and the page numbers of it have been removed.

## **1. Introduction**

### **1.1. Purpose**

The purpose of this document is to describe a filing management software that is to be developed to efficiently solve the following problem.

It is very common for most people to receive a variety of paper documents that need to be filed to be referenced on a future date. The problem is how to efficiently file them in a way that they can be easily retrieved in the, possibly distant, future. We normally put them in folders on file drawers sorted by some criteria. But which criterion is the right standard for all kinds of documents? If after some time you do not remember which criterion you used to file a document, you may have a hard time finding it. Every time that we need to file a new document we have to first find if we have already have created a place where this document belongs, and if not, then we need to think about what will be the proper way to file it. Without an efficient system, this could be time consuming, and if done wrong could lead to the misplacement of the document. If other people need to find a file they may not be able to easily guess the criterion that you used to file it. This problem can be extended to storing any object anywhere, for example, the things that we put in a box in the garage and then we forget where we put them.

### **1.2. Scope of the project**

To create a web based database system to catalog items that are to be filed or stored away.

- a. The user should be able to store the name, description, categories, location and other attributes of the documents or objects to be filed so that they can be easily searched and located at any time.
- b. The system should be web based and should work efficiently from a

smartphone.

- c. Each file (container) should be able to be assigned to multiple categories.
- d. It should make the process of filing and retrieving documents easier, not harder.
- e. It would probably be written as Java Server Pages or PHP, connected to a MySQL or hsqldb database. The HTML should probably use CSS to allow for different screen sizes.

### **1.3. Definitions, acronyms, and abbreviations**

**Catalog:** a collection of containers and items.

**Container:** an item that contains other containers or items

**Item:** Information related to an object that is or will be cataloged by the system. When an item contains other items, then we may call it a container but they are basically the same kind of entity in the database.

## **Requirements Document**

### **Vision of the solution: Feasibility Study**

This project's main concern is to develop a web based system to help properly organize and catalog physical documents or objects that are going to be filed for a possible later retrieval. For the purpose of this document the name of the system will be Quick-see. The users should be able to store something into a location or container, and then document in the system where that particular item is. Later, the user can look up that specific item, and our system will display exactly where it is that the object was stored. Also, the database should be able to store descriptions and various characteristics of the stored object.

There has been plenty of other work in this area with filing systems such as The Paper Tiger. This system allows for users to catalog where they have put documents inside a file cabinet. Then, the user can search for that document, and The Paper Tiger will tell them where it is within the cabinet. Quick-see can act as marketable competition to The Paper Tiger because we plan to overcome certain limitations that Paper-Tiger has.

One of the characteristics that make Quick-see different is that we are enabling the user to add an infinite amount of containers within containers in the database. For example, if a user has a file cabinet that would be one container (say Cabinet 1). If there

is a box within the file cabinet that would be an additional container (say Car Information Box). Then, there can be a folder within the box, which will be a third container (say Insurance Info). Finally, the user can add an item (say BMW 1999 insurance) into the folder (resulting in the path Cabinet 1 ---> Car Information Box ---> Insurance Info---> BMW 1999 insurance). Another unique feature of Quick-seeK is that it will incorporate a color coding mechanism to help minimize some human errors that can cause the misplacing of an item within a database.

As well as The Paper Tiger, there exists other kind of filing systems like the one called DigitalDrawer, which aims to virtually keep track of a company's paper documents. The DigitalDrawer software scans in the paper document, and saves the file to the computer. This differs from our software because we do not aim to make a user go paperless. Instead, Quick-seeK will be just a digital database that maps to physical objects. Going paperless (essentially item-less) would be impractical for many uses.

## Analysis of requirements

The software that is being proposed for this project, tentatively titled Quick-seeK, hopes to make the lives of its users easier by providing a system that allows them to know the location of an item quickly and precisely.

This web based program hopes to allow the user to store the name, description, category, location and other attributes of the item to be filed so that it can be easily searched and located at any time. For instance, the user does not need to remember what container the item was filed in, they only need to do a search regarding any one of the tags/keywords that are associated with that item to be able to find it. This program will make the process of filing and recovering items easier.

## Risk assessment and Reduction plan

**Risk 1:** Unrealistic schedule, greater workload than anticipated

**Description:** It may be the case that the features promised for the project may have been due to an overzealous calculation, and realistically the functionality of the final version 1 of the product may have to be more modest, since the amount of work to be done may exceed the realistic time frame of the project.

**Priority:** High. If proper and accurate planning is not done, then there is little hope for the product being completed on time.

**Probability of happening:** Moderate. This project calls for a lot of time to be put into it in a productive manner. The developer must learn some new skills necessary to complete the project. **Impact:** High. If proper and accurate planning of a schedule is not done then the project may not be completed on time.

**Mitigation:** A system of keeping track of the work and the tasks that need to be completed has been implemented. The functions that are essential will be done first and any additional features will be implemented if there is time.

**Contingency:** If it becomes apparent that time is running out to complete the essential features of the software, overtime will need to be scheduled.

## **Non-Functional Requirements**

### **A. Product Requirements**

#### **1. Efficiency Requirements**

- a. The application should take no more than 3 seconds to load each page.
- b. The application should inform the user (by something such as a loading bar, spinning beach ball, etc.) during all functions that may take more than 3 second of processing time.
- c. Each database should be able to be accessed concurrently by no less than 5 users.
- d. The application should not require any processing on the user's computer that could not be accomplished by a 2.0 GHz single-core Pentium 4 CPU with 1 GB of RAM running Windows XP Home on a 7200RPM HDD.

#### **2. Reliability Requirements**

- a. It should be available 24 hours a day, 7 days a week, all year long. The only downtime should be for critical system upgrades or bug fixes, and these should be attempted to be completed during off-peak hours to minimize loss of use.
- b. The program works consistently mainly for Google Chrome, all operating mayor systems, and geographic locations. We will define "consistently" as the program is operating as it is intended to 95% of the time.
- c. The system will be less expensive, and more specialized than other existing systems such as The Paper Tiger. Quick-seeek will have unique features such as the ability to add color coding to each item.

#### **3. Portability Requirements**

- a. Although the system was created for Google Chrome it should also be compatible with Mozilla Firefox, Internet Explorer, Apple Safari, as well as general mobile web browsers. However, users are recommended to access the system with Google Chrome whenever possible.
- b. The minimum screen resolution that should be supported is 1024x768, as at least 98% of all web users are using a screen resolution equal to or higher than this.
- c. It should be coded with CSS support for all different supported browsers as well as a default basic styling option that will be supported by all browsers.

### **B. Organizational Requirements**

#### **1. Delivery Requirements**

- a. The system should be web based and should work efficiently from a desktop and a smartphone.

#### **2. Implementation Requirements**



a. It should be written in HTML with server side processing using Java, PHP or similar, connected to database. The HTML should use CSS to allow for different screen sizes.

### **3. Standards Requirements**

a. All web pages should follow the same HTML and CSS based template.

## **C. External Requirements**

### **1. Interoperability Requirements**

a. As this is a web-based application with multiple browsers supported, it should be OS-independent and able to share data seamlessly between different browsers.

b. It should take into consideration that in future versions the system is expected to be multi-lingual and multi-zone ready. It must be easy to add versions for other languages or zones like Spanish-Venezuela, Portuguese-Brazil.

### **2. Legislative Requirements**

a. The application is regulated under the MIT License

b. The application contains no warranty

### **3. Ethical Requirements**

a. The organization's data protection officer must certify that all data is maintained according to data protection legislation before the system is put into operation.

## **Use cases with complete textual description**

### **List of Use Cases:**

1. Sign-up for new account. In future version needs to give instant access after email verification.
2. Login to account
3. Log-out of account (every page will have this, except for home)
4. List of catalogs. Show all catalogs created on a specific account
5. Search through a catalog by keywords to find an item. This is a search field. (every page after List of Catalogs Page)
6. Search through all catalogs to find an item.
7. Browse through the database going through selected items
8. Add, Modify, or Delete users
9. Add, Modify, or Delete a catalog
10. Add, Modify, or Delete an item in a catalog

11. View item information (limited view)
12. Add, Modify or Delete Labels (Future version)
13. Associate one or more labels to an item (Future version)
14. Find all items associated with a label (Future version)
15. Check-out or check-in an item from the filing system (Future version)
16. List item history (Future version)
17. Color code an item
18. Move an item and all its children to another location (Future version)
19. List items alphabetically

**Name:** Sign-up for new account.

**Code:** FR NewAcc

**Actor(s):** Anyone.

**Precondition:** Someone has arrived to the homepage of the system.

**Trigger:** Activate the account registration link.

**Basic Path:** The user fills out a form to obtain an account.

**Alternative Path:** User cancels the request and returns to the home page.

**Exception paths:** There is a problem with the connection to the Database, and the user gets a message encouraging him to try later.

**Other:**

**Name:** Log In to account.

**Code:** FR LogAcc

**Actor(s):** Any active user.

**Precondition:** The user has arrived to the login page.

**Trigger:** The user enters a user and password in the proper fields and submits it.

**Basic Path:** The user can enter a user and password to login to his account and submits the information.

**Alternative Path:** After submitting username and password, the user is not authenticated and is redirected to the login page which now shows that access was denied and gives an option to try again.

**Exception paths:** The system fails to connect to the database, in which case the user is informed of the problem and is invited to try again at a later time.

**Other:** One username can be associated with only one account. Therefore every user name must be unique.

**Name:** Log-out of account

**Code:** FR LogOut

**Actor(s):** Administrator\_User, Standard\_User, Restricted\_User

**Precondition:** A user is logged-in to the system, and is viewing any of the pages of the system.

**Trigger:** User activates a logout link or the session times out due to inactivity.

**Basic Path:** The user is sent to the homepage and his session is invalidated.

**Alternative Path:** The user navigates to a page outside of the system in which case the session remains open until it times out.

**Exception paths:**

**Other:** All pages except for the home or any external page will have a logout option

**Name:** List all catalogs

**Code:** FR LisCat

**Actor(s):** Administrator\_User, Standard\_User, Restricted\_User

**Precondition:** The user has recently logged in from the homepage or has navigated to the Catalogs List page.

**Trigger:** Navigating to the Catalogs List page.

**Basic Path:** A list of all the catalogs available to the account is displayed.

**Alternative Path:**

**Exception paths:** There is a problem with the connection to the Database, and the user gets a message encouraging him to try or contact support if the problem persists.

**Other:**

**Name:** Search by keyword through a catalog to find an item.

**Code:** FR SrchKW.

**Actor(s):** Administrator\_User, Standard\_User, Restricted\_User.

**Precondition:** A catalog must have been selected previously and the user is viewing any of the pages related to that catalog.

**Trigger:** User submits the keywords.

**Basic Path:** User enters some keywords and requests the search by submitting them. A list of all items containing all of the entered keywords is displayed along with each item's description. Sorted by item name. Each item can be selected to show its details or to be edited which may include deleting it.

**Alternative Path:**

**Exception paths:** If no items are found, a message that no items were found is displayed at the result page. Alternatively, the user can be taken back to the previous page but a noticeable message indicating that no items were found should be displayed.

**Other:**

**Name:** Search by keyword through the whole account to find an item.

**Code:** FR SrchAc.

**Actor(s):** Administrator\_User, Standard\_User.

**Precondition:** The user must be logged-in and viewing the list of catalogs.

**Trigger:** User submits the keywords to search.

**Basic Path:** Similar to FR SrchKW but the search is not limited to a catalog, instead it searches the whole account. User enters some keywords and requests the search by

submitting them. A list of all items containing all of the entered keywords is displayed along with each item's description. Each item can be selected to show its details or to be edited which may include deleting it.

**Alternative Path:**

**Exception paths:** If no items are found, a message that no items were found is displayed at the result page. Alternatively, the user can be taken back to the previous page but a noticeable message indicating that no items were found should be displayed.

**Other:**

**Name:** Browse through a catalog.

**Code:** FR BroCat.

**Actor(s):** Administrator\_User, Standard\_User, Restricted\_User.

**Precondition:** The user must be logged-in and viewing the list of catalogs.

**Trigger:** The user clicks on the link of a catalog

**Basic Path:** The user navigates to any location or item by following the links. When a user clicks on a catalog or location then a list of all the locations/items contained in that location is displayed.

**Alternative Path:** If the location does not have any descendants then the detail view of the location is presented.

**Exception paths:**

**Other:** When the item has other items within it (descendants), the descendants are listed in alphabetical order.

**Name:** Add, Modify, or delete users and access rights (Manage users)

**Actor(s):** Administrator\_User

**Code:** FR MngUsr

**Precondition:** The user has to be logged-in and has to belong to the Administrators group. The user also has to be viewing a page that has a way to access the Users Management function.

**Trigger:** Click on a link or button to go to the Users Management function

**Basic Path:** On the first version the functionality will be simple. The user is first presented with a list of all the users registered for the account. Selecting any of the listed users will display a form that allows the administrator to edit or delete the information of the selected user including its password and access level. While viewing the list of users (previous window), the admin would also have an option to create a new user. If the create new user option is selected a form to enter the data for the new user is presented. When the data is submitted it must first be validated. Of special importance is to validate that the user is unique within the account. The administrator can give the new user one of 3 types of access levels that is predefined.

**Alternative Path:**

**Exception paths:**

**Other:** On a future version the admin should be able to create groups which would define which pages can be accessed by the members of the group. Then when a user is created or edited, the admin can assign one or more groups to it. In order to create the groups, each page should be registered in a table of the database.

**Name:** Add, Modify, or delete a Catalog in the account (Manage Catalogs)

**Code:** FR MngCat

**Actor(s):** Administrator\_User

**Precondition:** The user has to be logged-in and has to belong to the Administrators group. The user also has to be viewing the list of catalogs page.

**Trigger:** Request to edit a catalog or to create a new catalog.

**Basic Path:** When the admin request to edit a catalog a form to modify the fields of the catalog is presented. The form also has an option to delete the catalog. A similar form is presented if the user had requested to create a new catalog, but in this case the form is empty and it does not have a delete catalog option.

**Alternative Path:** If the user is not an admin, an authorization error message is displayed.

**Exception paths:**

**Other:** The catalog must be empty in order to be deleted.

**Name:** Add an item to a container/item or catalog

**Code:** FR AddItem

**Actor(s):** Administrator\_User, Standard\_User

**Precondition:** The user must be viewing the root level content of a catalog or a container/item.

**Trigger:** The user requests to add an item

**Input Fields:** CatalogID or possibly ItemID. This is required to know where to insert the new item.

**Output:** A form with all the required fields to add a new item

**Basic Path:** A user requests to add an item. When the form is displayed, the user fills it in and submits it. Data from the form fields should be validated as much as possible.

**Alternative Path:** The user decides not to add any item and cancels the request.

**Exception paths:**

**What to test:** That all fields are validated, that the sql is not invoked with invalid data, that the correct sql is invoked. For integration, that the item is added correctly.

**Other:** Associating labels to the item should be handled from here.

**Name:** Modify, or delete an item in a container (Manage Item)

**Code:** FR MngItem

**Actor(s):** Administrator\_User, Standard\_User

**Precondition:** The user must be viewing the root level content of a container/item.

**Trigger:** The user requests to edit an item

**Input Fields:** ItemID. This is required to know which item is to be edited.

**Output:** A form with all the fields of an item prefilled with the data of the selected item and editable.

**Basic Path:** A user requests to edit an item. When the form is displayed, the user edits any fields for which the user wants to make changes and submits the data. Data from the form fields should be validated as much as possible.

**Alternative Path:** The user requests to delete the item, in which case a confirmation dialog is displayed and if accepted the item is deleted. If the user does not confirm the intent to delete, he is returned to the form.

**Alternative Path:** The user decides not to make any changes and cancels the edit request.

**Exception paths:**

**What to test:** That all fields are validated, that the sql is not invoked with invalid data, that the correct sql is invoked. For integration, that the item is modified correctly.

**Other:** Associating labels to the item should be handled from here.

**Name:** View item information

**Code:** FR Viele

**Actor(s):** Administrator\_User, Standard\_User, Restricted\_User

**Precondition:** The user must be viewing a list of items

**Trigger:** The user selects the item

**Input Fields:** ItemID

**Output:** A page displaying all the item's fields. The fields must be read only.

**Basic Path:** The user has one or more items listed and selects one. The system responds by showing the item's detail (List of all fields).

**Alternative Path:**

**Exception paths:**

**What to test:** Given an itemID the system creates the correct sql to select the item.

**Other:** When the details of the item are presented, an option to check-out, check-in an item should be available.

**Name:** Add a Label

**Code:** FR AddLbl

**Actor(s):** Administrator\_User, Standard\_User

**Precondition:** Add a label can be called from any place within a catalog

**Trigger:** A user request to add a label

**Input Fields:** CatalogId (Former DBId), a reference to the calling page

**Output:** A form to create a label

**Basic Path:** The user requests to add a label, a form with all the fields that the user can edit for a label is presented. The user enters the data and submits it. The user is returned to the calling page.

**Alternative Path:** The user decides not to create the new label and cancels the form, at which point he is returned to the calling page.

**Exception paths:**

**What to test:**

**Other:** Note that Add label can be called from different places and it should be able to know where to return when it finishes. This will not be implemented until version 2, unless there is time to spare.

**Name:** Modify Labels

**Actor(s):** Administrator\_User, Standard\_User

**Code:** FR ModLbl

**Precondition:** The user must be viewing the list of labels for a catalog

**Trigger:** The user requests to edit a label

**Input Fields:** Label ID

**Output:** A form with all the fields of a label that can be edited by a user

**Basic Path:** The user requests to modify a label. The system responds with a pre-filled form with all the editable fields of the label. The user makes all the modifications that he wishes and submits the data which causes the label to be modified.

**Alternative Path:** The user can request to delete the label in which case a confirmation dialog is presented. If the user confirms the deletion, the label is deleted; else the user is taken back to the unmodified form.

**Alternative Path 2:** The user can request to cancel the modification in which case he is returned to the list of labels.

**Exception paths:**

**What to test:**

**Other:** If a label is deleted, all its associations need to be deleted as well. This will not be implemented until version 2, unless there is time to spare.

**Name:** Associate one or more labels to an Item

**Code:** FR AssLbl

**Actor(s):** Administrator\_User, Standard\_User

**Precondition:** The user is editing an item.

**Trigger:** The user requests to associate a label to the current item

**Input Fields:** Item ID

**Output:** A list of all the available labels for the current catalog. The user must be able to select all the labels that he wants associated with the current item.

**Basic Path:** The user requests to associate a label. The system responds with a list of all the available labels for the current catalog showing the ones that are already associated, if any. The user can then select some more labels and or deselect whichever he wants. If the user wants to select a label that does not exist, he can request to add a new one. When the user is done selecting the labels he submits the list and returns to the form where he was editing the item.

**Alternative Path:** The user can decide not to add or change any of the existent association and cancel the request in which case he is returned to the form where he was editing the item.

**Exception paths:**

**What to test:**

**Other:** This will not be implemented until version 2, unless there is time to spare.

**Name:** Filter the results of a search by label

**Code:** FR FltLbl

**Actor(s):** All users

**Precondition:** The user must be viewing at the results of a search

**Trigger:** A user request to filter a search result by a set of selected labels

**Input Fields:** Search query

**Output:** A list of items/containers that satisfy the query and the filter

**Basic Path:** The user decides to filter the results from a search and selects one or more labels so that the results shows only items and containers that are associated with all the selected labels

**Alternative Path:** The user can decide not to filter the result, so he can cancel the request and leave the results unchanged.

**Exception paths:**

**What to test:**

**Other:** This will not be implemented until version 2, unless there is time to spear.

**Name:** Check-out or check-in an item or container from the filing system

**Code:** FR CheOut

**Actor(s):** Administrator\_User, Standard\_User

**Precondition:** The user is editing an item or container

**Trigger:** Request to check-out or check-in an item

**Input Fields:** Item ID

**Output:** Form to mark the item as checked in or out, and to record notes and who the person responsible for returning the item is.

**Basic Path: A user that is editing an item or container requests to check-out or check-in and item.** A form is presented where the user can add notes, the name of the person responsible to return the item, and an indicator as to whether the item is checked-in or checked-out.

**Alternative Path:**

**Exception paths:**

**What to test:**

**Other:** This will not be implemented until version 2, unless there is time to spear.

**Name:** Color code an item or a container

**Code:** FR - ColCod

**Actor(s):** Administrator\_Use, Standard\_User

**Precondition:** The user is adding or editing an item or container

**Trigger:** The user selects a color code for the item or container

**Input Fields:** Item Id

**Output:**

**Basic Path:** While the user is adding or editing an item or container, he can select a color code for it in one of the fields.

**Alternative Path:**

**Exception paths:**

**What to test:**

**Other:** The color code is a visual aide to help reduce errors. The way to use it is up to the users. It is recommended that the users set up a common policy for color coding the items and containers. For example, folders can be coded with alternating colors to choose from a set of three contrasting colors. When a new folder is added in between two folders, it must be of different color than the other two. In that way if someone is filing a folder in the wrong place  $\frac{2}{3}$  of the times the folder would be of the wrong color which will help the person immediately recognize the mistake.



**Name:** Move a container or item to another container

**Code:** FR MovCont

**Actor(s):** Administrator\_User

**Precondition:** The user must be editing the container or item to be relocated

**Trigger:** A user request to relocate the item

**Input Fields:** Item ID

**Output:** A list of the top level containers from which the user can navigate to the desired new location

**Basic Path:** The user requests to relocate the container or item, the system then presents a list of the top level containers from which the user can navigate to the desired new location. When the user arrives to the desired location he can chose to relocate the item to that place or cancel the request to relocate.

**Alternative Path:**

**Exception paths:**

**What to test:**

**Other:** When a container is relocated, all the items, containers and sub containers are relocated with it. In other words, the whole tree is relocated.

This will not be implemented until version 2, unless there is time to spear.

**Name:** Generate Reports

**Code:** FR GenRpt

**Actor(s):** Administrator\_User, Standard\_User, restricted\_User

**Precondition:** The user must be logged-in and must have chosen a catalog

**Trigger:** A request from the user to present a list of available reports

**Input Fields:** Catalog Id

**Output:** A list of links to available reports

**Basic Path:** The user request to generate reports. The system responds with a list of available reports. The user can select one which will then be displayed. At this point the user can chose to print the report.

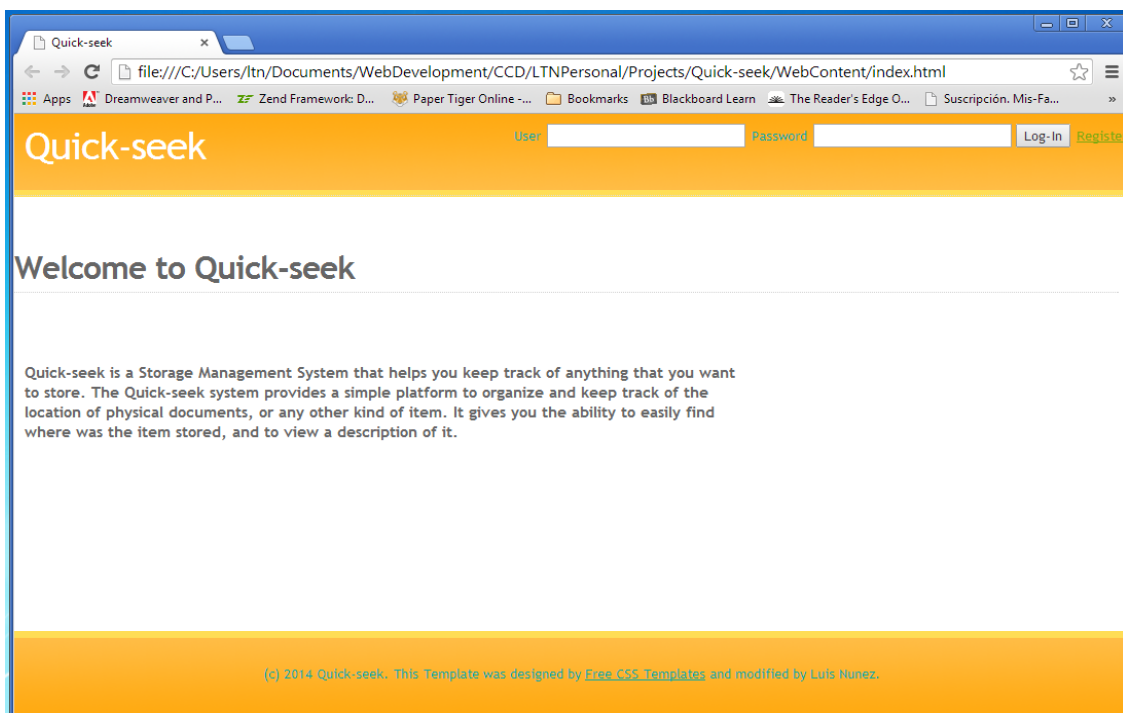
**Alternative Path:**

**Exception paths:**

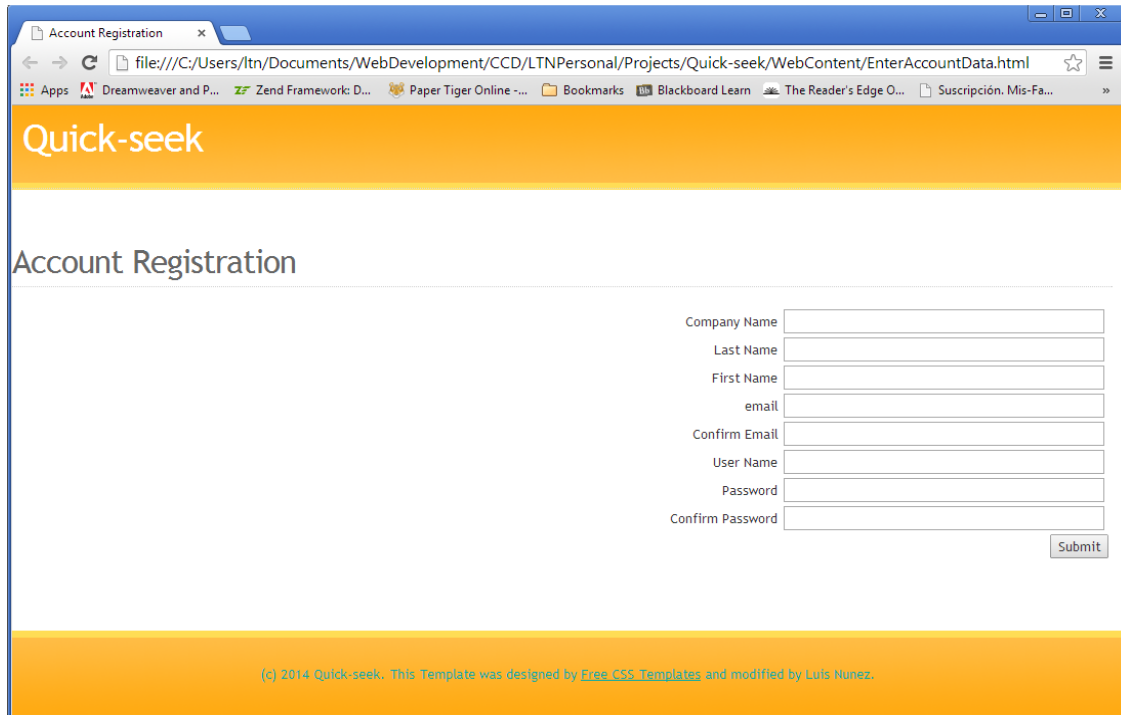
**What to test:**

**Other:** In a future version there should be an option that allows the user to define his own reports.

## Graphical User Interface



This is a screenshot of what the user will see when he first navigates on to the Quick-seek website. There is a brief description statement, explaining our system and what it is aimed to do. The user can either sign in to their previously existing account, or he can register a new account. If he wants to create a new account, he will click the “register” button and be taken to the following screen:

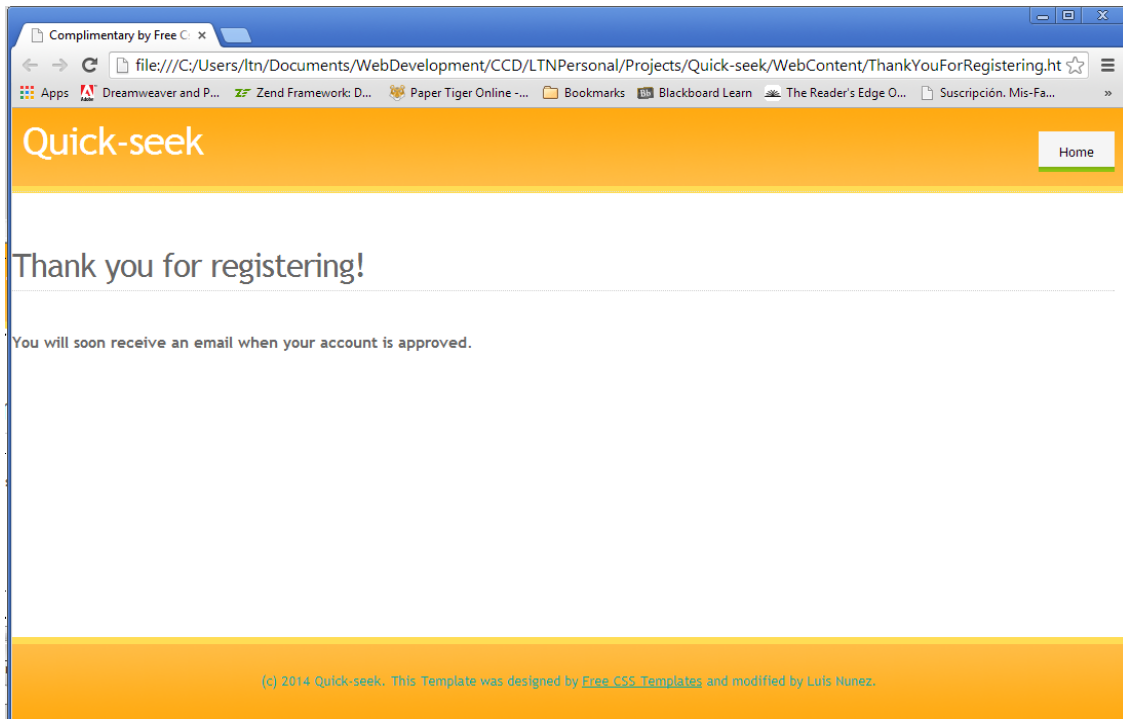


The screenshot shows a web browser window with the title 'Account Registration'. The address bar displays the file path: file:///C:/Users/ltn/Documents/WebDevelopment/CCD/LTNPersonal/Projects/Quick-seek/WebContent/EnterAccountData.html. The browser's taskbar shows several open applications: Dreamweaver and P..., Zend Framework: D..., Paper Tiger Online ..., Bookmarks, Blackboard Learn, The Reader's Edge O..., and Suscripción. Mis-Fa... The main content area features a yellow header with the 'Quick-seek' logo. Below the header, the title 'Account Registration' is displayed. The registration form consists of the following fields:

- Company Name
- Last Name
- First Name
- email
- Confirm Email
- User Name
- Password
- Confirm Password

A 'Submit' button is located at the bottom right of the form. At the bottom of the page, a footer contains the text: (c) 2014 Quick-seek. This Template was designed by [Free CSS Templates](#) and modified by Luis Nunez.

This is a screenshot of the account registration form. A user has to enter some information including a password that will be stored for future access to the account. The Company Name field will be the name of the account created. When all the information is input correctly by the user, he will hit the “submit” button and be taken to the next screen if there are no errors. Otherwise, a similar form is presented showing the errors that need to be corrected.

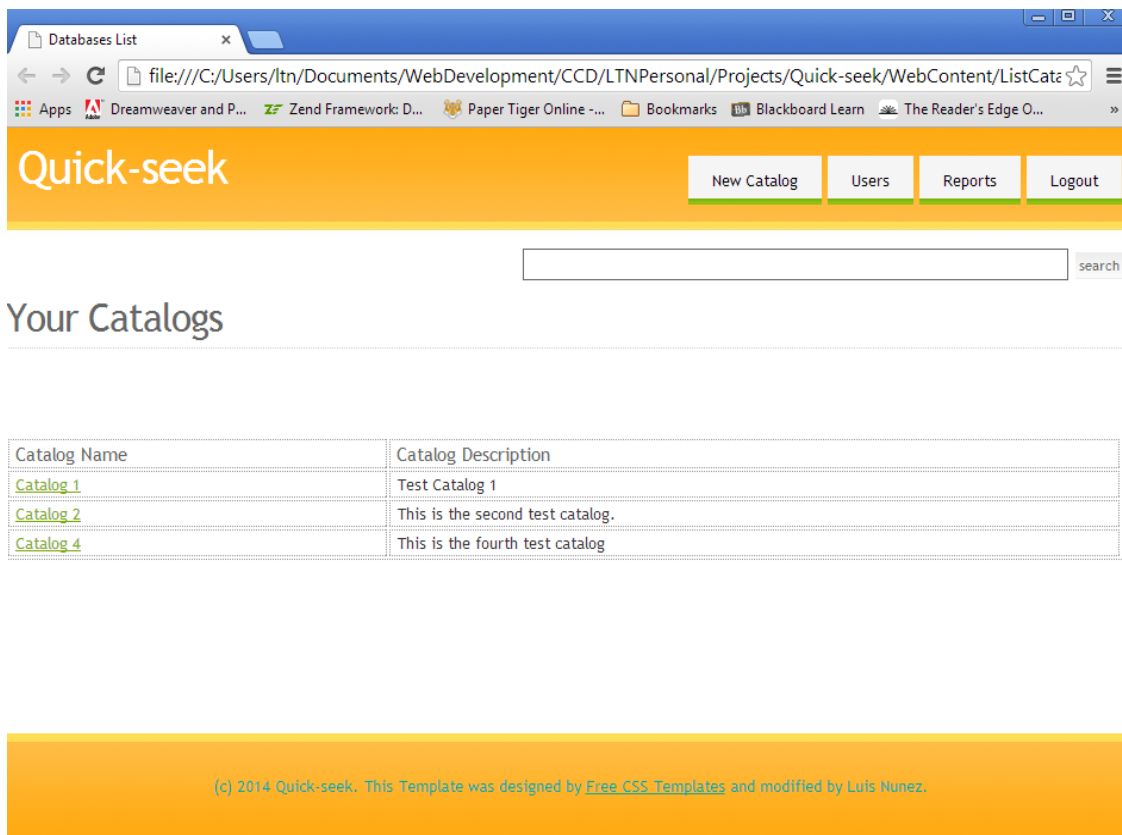


This is the screen the user will be taken to once he has submitted an account. Once it is approved, he will be sent an e-mail, and will then have the ability to start creating databases within his account.

The screenshot shows a web browser window with the following details:

- Browser tabs: Complimentary by Free C..., ListItems, ListItems
- Address bar: file:///C:/Users/ltn/Documents/WebDevelopment/CCD/LTNPersonal/Projects/Quick-seek/WebContent/AddUser.html
- Page title: Quick-seek
- Page navigation: Labels, Logout
- Form title: Add User
- Form fields:
  - Search bar (with search button)
  - Last Name
  - First Name
  - email
  - Confirm Email
  - User Name
  - Password
  - Confirm Password
  - Access Level (dropdown menu)
  - Submit button
- Footer: (c) 2014 Luis T. Nunez. This Template was designed by [Free CSS Templates](#) and modified by Luis Nunez.

This screenshot is the user registration form. This is different from the account registration form above. Here an administrator user adds additional users to an already existing account. The administrator also has the ability to select the level of access to the account that the new user will have.



The screenshot shows a web browser window with the title "Databases List". The address bar contains the file path: `file:///C:/Users/ltn/Documents/WebDevelopment/CCD/LTNPersonal/Projects/Quick-seek/WebContent/ListCats`. The browser's bookmark bar includes "Apps", "Dreamweaver and P...", "Zend Framework: D...", "Paper Tiger Online -...", "Bookmarks", "Blackboard Learn", and "The Reader's Edge O...".

The main content area features a blue header with the "Quick-seek" logo on the left and four navigation buttons on the right: "New Catalog", "Users", "Reports", and "Logout". Below the header is a search input field with a "search" button.

### Your Catalogs

Catalog Name	Catalog Description
<a href="#">Catalog 1</a>	Test Catalog 1
<a href="#">Catalog 2</a>	This is the second test catalog.
<a href="#">Catalog 4</a>	This is the fourth test catalog

(c) 2014 Quick-seek. This Template was designed by [Free CSS Templates](#) and modified by Luis Nunez.

This screenshot is the list of catalogs. Here, a user or administrator can look at a listing of all the current catalogs in the account. By clicking on one of these catalogs, the user will be taken to the list of first level Items/containers for that catalog.

The screenshot shows a web browser window with the title "Databases List". The address bar shows a file path: "file:///C:/Users/ltn/Documents/WebDevelopment/CCD/LTNPersonal/Projects/Quick-seek/WebContent/ListCatz...". The browser's taskbar includes "Apps", "Dreamweaver and P...", "Zend Framework: D...", "Paper Tiger Online -...", "Bookmarks", "Blackboard Learn", and "The Reader's Edge O...".

The web application has a blue header with the "Quick-seek" logo on the left and navigation buttons for "New Catalog", "Users", "Reports", and "Logout" on the right. Below the header is a search bar with a "search" button.

The main content area is titled "Catalog 1" and contains a table with the following data:

Item Name	Item Description
<a href="#">Item 1</a>	Test Item 1
<a href="#">Item 2</a>	Test Item 2
<a href="#">Item 3</a>	Test item 3
<a href="#">Item C1.4</a>	This is Item 4 of Catalog 1

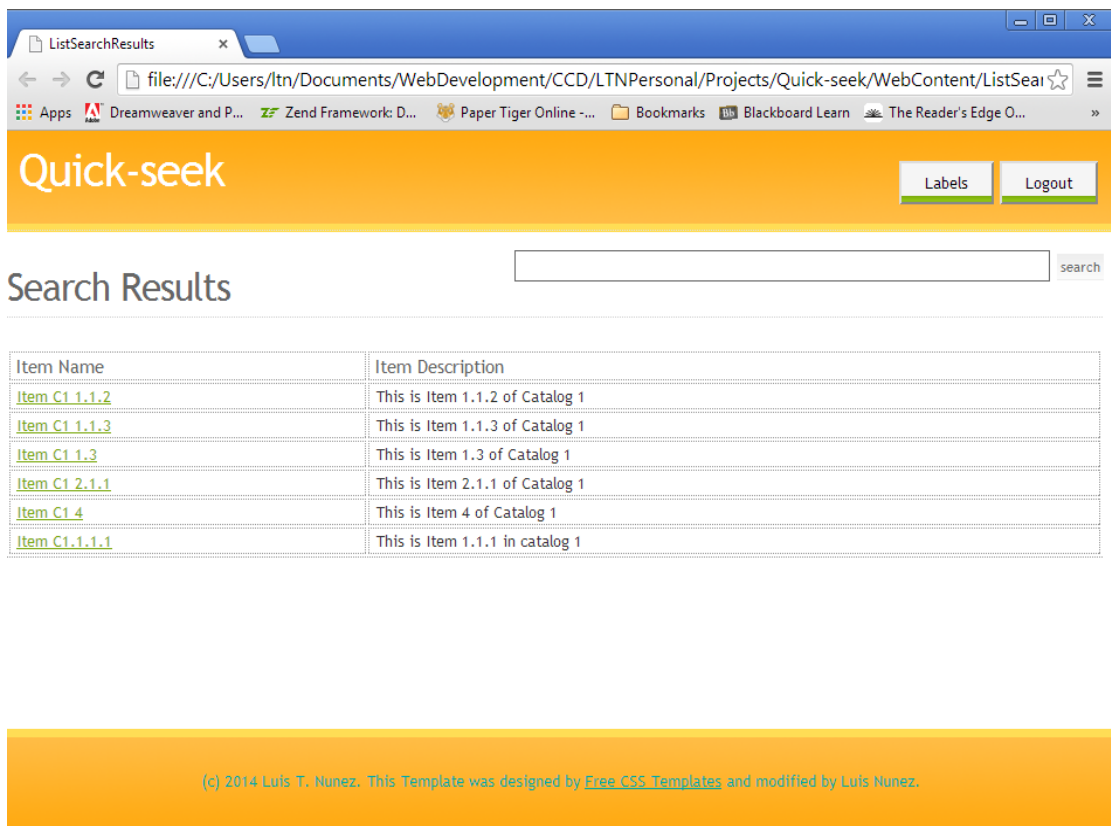
At the bottom of the page, there is a footer with the text: "(c) 2014 Quick-seek. This Template was designed by [Free CSS Templates](#) and modified by Luis Nunez."

Once a catalog is selected by the user, the list of first level items for that catalog is displayed. Also, there is a description field so the user can briefly explain each item or container. The same page is used for list items. Once the user enters into an item or container, each item (container within) will present itself along with a description next to it, just as it does in the list of catalogs.

The screenshot shows a web browser window with the title 'AddItems'. The address bar shows the file path: file:///C:/Users/ltn/Documents/WebDevelopment/CCD/LTNPersonal/Projects/Quick-seek/WebContent/AddItems. The browser's toolbar includes 'Apps', 'Dreamweaver and P...', 'Zend Framework: D...', 'Paper Tiger Online - ...', 'Bookmarks', 'Blackboard Learn', and 'The Reader's Edge O...'. The page has an orange header with the text 'Quick-seek' on the left and 'Labels' and 'Logout' buttons on the right. Below the header is a search bar with a 'search' button. The main content area is titled 'Add Item' and contains a form with the following elements: an 'Item Name' text input field, a 'Description' text area, and a 'Color' dropdown menu with options 'red', 'black', 'red', 'green', and 'blue'. A 'Save' button is located below the form. At the bottom of the page, there is a footer with the text: '(c) 2014 Luis T. Nunez. This Template was designed by [Free CSS Templates](#) and modified by Luis Nunez.'

This is the Add Item screen. It is used to store/file an item. Also it is used to modify the information if it changes (Edit Item) or there was a mistake.





The screenshot shows a web browser window with the title 'ListSearchResults'. The address bar contains the file path: 'file:///C:/Users/ltn/Documents/WebDevelopment/CCD/LTNPersonal/Projects/Quick-seek/WebContent/ListSearchResults'. The browser's toolbar shows several open tabs: 'Apps', 'Dreamweaver and P...', 'Zend Framework: D...', 'Paper Tiger Online - ...', 'Bookmarks', 'Blackboard Learn', and 'The Reader's Edge O...'. The main content area features a blue header with the 'Quick-seek' logo on the left and 'Labels' and 'Logout' buttons on the right. Below the header is a search bar with a 'search' button. The search results are displayed in a table with two columns: 'Item Name' and 'Item Description'. The table contains six rows of data. At the bottom of the page, there is a footer with the text: '(c) 2014 Luis T. Nunez. This Template was designed by [Free CSS Templates](#) and modified by Luis Nunez.'

Item Name	Item Description
<a href="#">Item C1 1.1.2</a>	This is Item 1.1.2 of Catalog 1
<a href="#">Item C1 1.1.3</a>	This is Item 1.1.3 of Catalog 1
<a href="#">Item C1 1.3</a>	This is Item 1.3 of Catalog 1
<a href="#">Item C1 2.1.1</a>	This is Item 2.1.1 of Catalog 1
<a href="#">Item C1 4</a>	This is Item 4 of Catalog 1
<a href="#">Item C1 1.1.1</a>	This is Item 1.1.1 in catalog 1

When a user enters some keywords in the search field and clicks search, he will be presented with all the items that match the search string, showing the name, a brief description, and the current availability status of each and every item.