VeriHandy

System Design Document

#### Table of Contents

Page

## 1. Introduction

* 1. Purpose of This Document
  2. References

1. System Architecture

2.1 Architectural Design

2.2 Decomposition Description

3. Persistent Data Design

3.1 Database Descriptions

3.2 File Descriptions

1. Requirements Matrix

Appendix A – Agreement Between Customer and Contractor

Appendix B – Peer Review Sign-off

Appendix C – Document Contributions

1. **Introduction**

1.1 Purpose of This Document

This document serves as an overview of the design for the VeriHandy application, whose requirements are laid out in the System Requirements Document. Includes information about the architecture and database, as well as notes about how design will be carried out, and the relationships between parts of the architecture.

1.2 References

* VeriHandy System Design Document
* VeriHandy UI Design Document
* Laravel PHP Framework – found at <http://laravel.com/>, upon request of Shawn Squire

1. **System Architecture**

2.1 Architectural Design

Figure 2.1.1 Basic Diagram showing the Client-Server Architecture in addition to some specifics of how the client-server relationship works for VeriHandy

VeriHandy Users

VeriHandy Server

Users use web pages to communicate with server and utilize VeriHandy

Server sends information using Laravel and displays using HTML web pages

The VeriHandy system uses a basic Client-Server Architecture as shown in Figure 2.1.1. The current system will utilize the Laravel PHP Framework to assist this architecture and make implementing and testing the product easier.

The current architecture is responsible for communicating a variety of information between the user and the server. This information includes unique usernames, passwords, location information, contact information, and job requesting information. This can break down the design into a few main parts:

* Users
  + Interact with the server through a UI outlined in the VeriHandy UI Document
  + Information is requested by users, such as job listings and account information, from the server
  + Information is given by users, such as their ID, password, search parameters, and job information
* Server
  + Stores jobs, job information, user account information, and reviews
  + Communicates requests and receives to and from Users through the UI outlined in VeriHandy UI Document

2.2 Decomposition Description

Illustrate and describe the decomposition into components of the system that you presented in Section 2.1 (e.g., functions, objects, scripts, files). This is the view of the system as you know it at this point in time. Provide diagrams as follows:

Other – If you are using PHP or another language that cannot be adequately described using a structural decomposition diagram and/or class diagrams, create custom illustrations. Show implementation components of the system (e.g., scripts, files) and their relationships.

Regardless of the type of diagram(s) that you use, refer the reader to the diagram(s) and describe what it is intended to communicate. Give a brief description of each of the components. If you are using a pre-defined pattern (e.g., Model-View-Controller), explain this to the reader. [Two to three substantial paragraphs]

1. **Persistent Data Design**
   1. Database Descriptions

Describe the database(s) used by the system. Include a diagram of the schema. You may use an entity-relationship (ER) diagram, or you may create your own diagram type (e.g., a collection of tables). For each field in a database, give its name, data type (e.g., int, double), size (e.g., strings), and description of what it represents. Basically, give all of the information that a programmer must have to implement the database. If no databases are used, simply state so. [Length is whatever it takes]

* 1. File Descriptions

Describe the file(s) used by the system. Include a diagram of the file structure. For each field in a file, give its name, data type (e.g., int, double), size (e.g., strings), and description of what it represents. Basically, give all of the information that a programmer must have to implement the file. If no files are used, simply state so. Supplement your description with a sample file(s). [Length is whatever it takes]

1. **Requirements Matrix**

Use a tabular format to show which system components (e.g., functions and/or methods) satisfy each of the functional requirements from the SRS. Refer to the functional requirements by use case number and name.

**Appendix A – Agreement Between Customer and Contractor**

Place on a separate page. Describe what the customer and your team are agreeing to when all sign off on this document. [One paragraph] Include a statement that explains the procedure to be used in case there are future changes to the document. [One paragraph] Provide lines for typed names, signatures, and dates for each team member and the customer. Provide space for customer comments.

**Appendix B – Team Review Sign-off**

Place on a separate page. Provide a brief paragraph stating that all members of the team have reviewed the document and agree on its content and format. Provide lines for typed names, signatures, dates, and comments for each team member. The comment areas are to be used to state any minor points regarding the document that members may not agree with. Note that there cannot be any major points of contention.

**Appendix C – Document Contributions**

Identify how each member contributed to the creation of this document. Include what sections each member worked on and an estimate of the percentage of work they contributed. Remember that each team member must contribute to the writing (includes diagrams) for each document produced.

**Unified Modeling Language (UML)**

**Class Diagrams**

Reference: **UML Distilled**, 2nd edition, Martin Fowler and Kendall Scott, 2000

Note: The information below has been modified slightly to meet the purposes of CMSC 345

Symbols:

Class – Represented by a box as follows:

*Class Name*

*Operations*

*Attributes*

*Class Name*

Navigability – Represented by a solid line with an arrowhead at one or both ends (unidirectional or bidirectional association, respectively). Indicates the direction(s) of an association between classes. (Below, A can navigate to B, but not the reverse.)

A

B

Generalization – Represented by a solid line with a hollow arrowhead at one end. Indicates that one class is a generalization of another. (Below, B is a generalization of A.)

A

B

Composition – Represented by a solid line with a solid diamond at one end. Indicates that an instance of one class is “owned” by a single instance of another. (Below, an instance of B is owned by a single instance of class A. Note: if an instance of A is deallocated, the associated instance(s) of B are also deallocated.)

B

A

Dependency – Represented by a dashed line with an arrowhead at one end. Indicates that one class depends on the interface to another class. (Below, an instance of A is dependent on the interface to an instance of B.)

A

B

Multiplicity – Indicates how many instances of one class type are associated with another class. (Below, 1 to 5 instances of class A are associated with class B.)

B

A

1**. .** 5

Note: We will not be using any other symbols in our class diagrams.

Format for class attributes:

*visibility name* : *type* = *defaultValue*

where *visibility* = + for public, # for protected, - for private

*name* = attribute name

*type* = data type

*defaultValue*  = default value

Format for class operations:

v*isibility name* (*parameter-list*) : *return-type*

where *visibility* = + for public, # for protected, - for private

*name* = operation name

*parameter-list*  = comma-separated parameters with the syntax

*direction name* : *type* = *defaultValue*

where *direction*  = in for input

out for output

inout for input/output

*name* = parameter name

*type* = parameter type

*defaultValue* = default value

*return-type* = return type