Dylan Barnes, Ryan Kilbride, and Cameron Kerbaugh

CSci 463, Spring 2016, Dr. Hassan Reza, Ph.D.

DC# Converter Project Design

DC# Converter Project Design Table of Contents

[Revision History 3](#_Toc448091112)

[1 Introduction 4](#_Toc448091113)

[1.1 Goals and Objectives 4](#_Toc448091114)

[1.2 Project Overview and Scope 4](#_Toc448091115)

[1.2.1 Core Features 4](#_Toc448091116)

[1.2.2 Additional Features 4](#_Toc448091117)

[1.3 Software Context 4](#_Toc448091118)

[1.4 Major Constraints 4](#_Toc448091119)

[2 Data Design 4](#_Toc448091120)

[2.1 Internal Software Data Structure 4](#_Toc448091121)

[2.2 Global Data Structure 4](#_Toc448091122)

[3 Architectural and Component-Level Design 4](#_Toc448091123)

[3.1 System Structure 4](#_Toc448091124)

[3.2 Primary Classes 4](#_Toc448091125)

[3.2.1 DCS\_CSV 4](#_Toc448091126)

[3.2.2 DCS\_JSON 4](#_Toc448091127)

[3.2.3 DCS\_XML 4](#_Toc448091128)

[3.2.4 Form1 4](#_Toc448091129)

[3.3 Auto-Generated Classes 4](#_Toc448091131)

[4 Interface Design 5](#_Toc448091132)

[4.1 Command-Line Interface 5](#_Toc448091133)

[4.2 User Interface 5](#_Toc448091134)

[4.3 Library 5](#_Toc448091135)

[5 Restrictions, Limitations, and Constraints 5](#_Toc448091136)

[6 Testing 5](#_Toc448091137)

[6.1 Types of Testing 5](#_Toc448091138)

[6.1.1 White Box Testing 5](#_Toc448091139)

[6.1.2 Black Box Testing 5](#_Toc448091140)

[6.1.3 Feature Testing 5](#_Toc448091141)

[6.2 Performance Bounds 5](#_Toc448091142)

[6.3 Critical Systems 5](#_Toc448091143)

[6.4 Test Cases 5](#_Toc448091144)

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| VERSION | TEAM MEMBER | DESCRIPTION | DATE |
| 0.1.0a | Dylan Barnes | Initial Document Layout and Introduction | 4/04/2016 |
|  |  |  |  |
| 0.1.1 | Ryan Kilbride | Fault tolerance, under Restrictions, Limitations, and Constraints | 4/12/2016 |
|  |  |  |  |
| 0.1.1b | Cameron Kerbaugh | Added content to Introduction | 4/12/2016 |
| 0.1.2 | Dylan Barnes | Added Sub-Clause 2.1  Added Sub-Clause 2.2  Added Sub-Clause 2.3 | 4/18/2016 |
| 0.1.2b | Dylan Barnes | Added Sub-Clause 6.1  Added Sub-Clause 6.2  Added Sub-Clause 6.3  Added Sub-Clause 6.4  Added Table 6-4 | 4/20/2016 |

# Introduction

This Project Design Document is intended to provide an overview and description of DC# Converter’s low-level design, looking into how each individual component is structured as well as how the overall software package is connected. Design information contained in this document includes: data structure used, software architecture chosen, user interface design details, and test cases along with their intended results.

## Goals and Objectives

DC# Converter aims to provide a portable, easy-to-use, multi-function conversion tool for converting between several popular data exchange formats, including XML, JSON, and CSV. It is a component of the larger DC# Software Suite, an open source software suite for managing and converting data files of various formats.

In order to provide the aforementioned portability and ease of use, both a GUI and command line functionality must be implemented. The GUI will also provide feedback to the user about the data contained in converted files,

## Project Overview and Scope

### Core Features

### Additional Features

## Software Context

## Major Constraints

# Data Design

## Internal Software Data Structure

The internal data structure of DC# Manager can be broken down into two subcategories: front-end and back-end.

The front-end work involves everything the user will interact with. This primarily involves the interfaces, which varies between a GUI, command-line interface, and library. The GUI must be designed simplistically, as it will be the primary source of interaction for the standard non-technical user. The GUI will also allow for additional functionality that may not be feasible in a command-line interface. However, the command-line interface will still attempt to mirror the functionality of the GUI, as this will be the method that other applications use to interact with DC# Converter. Together, these interfaces serve as a way to access the core and additional features of DC# Converter.

The back-end work involves the functionality, such as parsing and converting data. The functionality is relating to the features listed in sub-clause 1.2. In addition to this, research will have to be performed to determine if it is viable to use pre-existing C# libraries to assist in the conversion of data. If external libraries are required, it will be our goal to ensure the number of libraries required is kept to a minimum to decrease the number of dependencies.

## Global Data Structure

DC# Converter does not require a global data structure. The GUI and command-line applications are initialized on a per-instance basis, while the library is dependent upon how users implement the classes. DC# Converter may be viewed as a pipe and filter system, where files are fed into the functions of the converter, and a new file comes out the other end. No data is stored within the application itself.

## File Creation

It is worth noting that every successful conversion results in the creation of a new file. If the user is interacting with DC# Converter through the GUI, a file explorer will be displayed allowing the user to easily specify a directory and file name. On the other hand, if a command-line interface is used, the user will be expected to provide the full output file path before performing a conversion. The size of the output file is dependent on the size of the input file, as well as the output file type. This variance in size is due to the syntax of the file type. For example, converting a CSV file to XML will result in a larger file than if you were to convert that same CSV file to JSON. As one would expect, the larger the input file is, the longer conversion may take.

# Architectural and Component-Level Design

## System Structure

## Primary Classes

The DCS Converter is broken into three data-oriented modules and one TODO. The data-oriented classes, DCS\_CSV, DCS\_JSON, and DCS\_XML, TODO

### DCS\_CSV

#### Global Variables

#### Class Signatures

### DCS\_JSON

### DCS\_XML

### Form1

#### Global Variables

#### Class Signatures and SQL

## Auto-Generated Classes

In addition to the manually created classes listed above, there are numerous classes auto-generated by Visual Studio in order to properly format the interface as well as properly compile the application. For the sake of brevity, these will not be explained in further details within this document.

# Interface Design

## Command-Line Interface

## User Interface

## Library

# Restrictions, Limitations, and Constraints

## Limitations to Parsing

### Reliability

### Robustness

In the event that a file cannot successfully be parsed, the system gracefully handles internal errors. Any exceptions thrown internally are caught, and the user is notified via a message box.

# Testing

## Types of Testing

### White Box Testing

When a class is being implemented, the developer of that class will test to ensure each functional component is working properly. The developer of the functionality is fully responsible for debugging their own code, as debugging another developer’s code can be time consuming.

### Black Box Testing

Black Box testing involves a majority of the testing. This testing is done once all functionality exists and the components have been assembled. Tests will be performed throughout the entirety of the application, ensure every possible situation that could be applied to the application succeeds without error.

### Feature Testing

The features will be tested through the use of broad test cases. The reader may find these test cases in sub-clause 6.4, *Test Cases*.

## Performance Bounds

The local front-end performance is negligible, as the majority of the interface is auto-generated on the back-end, leaving little need for optimization. Performance will be obtained by increasing conversion times on the back-end, which may be done by manually modifying existing code or incorporating a new open-source parsing library.

Performance is being kept in mind throughout the development of DC# Converter. Improved performance is typically achieved through minimal use of looping, as well as minimal use of print statements within said loops. Current parsing times are sufficient for that of a conversion software, but further optimization may be looked into following the initial release.

## Critical Systems

The most critical functionality of DC# Converter is accurate data parsing. If a data file is not parsed correctly, the output will be incorrect as well. Ensuring the integrity of the original data is critical for user acceptance.

Throughout the development of DC# Converter, the accuracy of the parsing and corresponding output is consistently tested and optimized. The accuracy of the parsing is done manually as opposed to programmatically. The accuracy is typically done by comparing the input file to the output file, and ensuring that not only the syntax is correct but that no data was lost in the conversion process.

There are future plans to create programmatic unit tests. However, due to the accelerated schedule of the initial release, manual testing has been preferred, as programming unit tests would require an extensive amount of additional work time.

## Test Cases

Table 6-4 lists all currently planned test cases. Test cases are assumed to be performed on the graphic user interface, unless otherwise specified.

|  |  |
| --- | --- |
| FEATURE | CASES |
| *Input/Output File Selection* | User shall be able to select an input file through a file explorer. |
|  | User shall be able to select an output file through a file explorer. |
|  | Input and output file types shall be restricted to the file types available for input and output. |
| *Output File Type* | User shall be able to select an output file type. |
|  | The output file type restricts the output file type selection in the file explorer. |
|  | The user shall not have to specify an output file type if the command line is used, the output file type will be interpreted. |
| *Output File Options* | Additional options shall be provided to the user for the output file. |
|  | The provided options shall be dependent upon the selected output file type. Thus, the options differ between file types. |
| *Data Grid View* | The parsed data shall be displayed after conversion is performed. |
|  | The data will be displayed in a data grid, allowing users to easily view the content that will be output to a new file. |
| *Command-Line* | Users shall be able to access core functionality through the command line. |
| *C# Library* | Users shall be able to utilize all functionality through the use of a .dll library. |
| *Data Integrity* | The data integrity shall be verified throughout development. |
|  | Data integrity shall be validated manually. |
| *Syntax Validation* | The syntax out the output files shall be verified throughout development. |
|  | Syntax shall be validated manually. |
| *Performance* | Time all actions to ensure command-line conversion runs at a consistent, acceptable speed. |
|  | Time all actions to ensure the GUI conversion process runs at a consistent, acceptable speed. |

Table 6-4: Test Cases