**Software Design Specification**

**for**

***Super Sudoku***

***Created by Microware Concepts***

***2/14/2011***

**Version 1.00**

**1.0 Introduction**

The program to be developed is called *Super Sudoku For Windows* (abbreviated by the acronym *SSFW* throughout the remainder of this document)*.* This document describes all the data, architectural, interface and component-level design for the *SSFW* software. It is based on the corresponding *SSFW Software Requirements Specification* dated 2/10/2011.

**1.1 Goals and objectives**

Sudoku is a popular logic game, providing users with varying levels of challenge. *SSFW* will provide a convenient, computer-based version of the game which will allow users to improve their game-play by easily changing their guessed input values, optionally seeing clues and correct answers for the input values, and keeping a score based on the player’s elapsed time to solve the puzzle, and the puzzle level of difficulty. The objective is to provide the user with an intuitive, easy-to-use interface which replicates very closely the basic functioning of the paper and pencil version of the game, but with the convenience of keyboard inputs. The user will have all the same challenges of the original game, with the additional benefits of having hints or answers provided for individual unfilled squares within the game grid.

**1.2 Statement of scope**

*SSFW* will be a stand-alone program for PCs running Windows XP or later operating systems. The program will offer the user a GUI interface, matching standard Windows program menu standards, and will:

1) generate Sudoku games for individual players to play

2) optionally provide hints and solutions to the player during the game

3) solve Sudoku games entered from other sources using the keyboard

4) save and load games for later play

**1.3 Software context**

*SSFW* is a stand-alone program, developed as an individual product. Although some speculation has suggested a future game line of products, there are no implied requirements or expectations for fitting *SSFW* into a larger business context.

**1.4 Major constraints**

Standard software development practices shall be utilized throughout the development of *SSFW*, utilizing a basic ‘Waterfall Model’ as the process model. No other constraints have been specified for this document.

**2.0 Data design**

Due to the uncomplicated nature of the *SSFW* program, only a few data elements and structures are needed. These are described in the following sections for Internal, Global, Temporary, and Database structures.

**2.1 Internal Software Data Structures**

A simple structure *list* is used to indicate which numbers, 1 through 9, are possible values for a particular square within the 9x9 game grid. The *list* is an array of booleans, using index slots 1 through 9, where TRUE means that the index for that slot is a possible legal value, and FALSE means that the index is not usable in the current grid location. The *list* values are set by the SLV module XXXX

**2.2 Global Data Structures**

The primary data structure for this system is a representation of the 9x9 game grid. This grid, referred to as the *gamegrid,* will be represented with a 9x9 integer-typed array to hold the contents of the game squares. Grid elements which are non-editable will contain the negative of their values, allowing the GUI grid handling functions to distinguish them when creating the display squares for the user inputs.

**2.3 Temporary Data Structures**

None identified.

**2.4 Database and Data File description**

**2.4.1 Databases**

*SSFW* does not require any formal database structure as such, and needs no database engine or support software to maintain the necessary data.

**2.4.2 Data Files**

**2.4.2.1 Game Save/Load File**

The only data file necessary is one to store a Sudoku at any point in the play process. This file will be saved at the user’s request, and reloaded as the current game, again at the user’s selection. The data which must be stored in the file are:

1) Contents of each square in the 9x9 game grid, including digits 1..9, and blank

2) Identification of whether each square is a locked, initial value of the puzzle or an editable square representing a user input square.

The file specification requires that it be in ASCII text format, readable by simple text editors. Each square requires two pieces of information– its value and its editability. Therefore, the file format will use two characters for each grid square, as *value editable*. The *value* character will be the ASCII digits ‘1'..’9', plus the digit ‘0' to designate a blank square. The *editable* character will use the ASCII ‘+’ to indicate an editable square, while the ‘-‘ will designate a non-editable square. Each row of the 9x9 grid will be represented by a CR-LF terminated line of 18 characters, with the entire grid consisting of 9 lines. For example, the following is a valid line:

0-1-2-3-4-5-6+7+8+9-

indicating the values 0 through nine appear in order on the line, with the only editable squares containing the values 6, 7, and 8.

**3.0 Architectural and Component Level Design**

The design details for *SSFW* are contained in this section.

**3.1 Program Structure**

*SSFW* is a simple procedure call structure, as illustrated in section 3.1.1.

**3.1.1 Architecture diagram**

A pictorial representation of the architecture is presented below:

**GEN**

**Puzzle Generator**

**GUI**

**User Interface**

**SLV**

**Puzzle Solver**

**FIL**

**File Handler**

The logical modules shown above are assigned the following functionality:

**GUI-** Implements the user interactions for the menus, grid display and input. Sends file load or save requests to FIL. Controls when GEN generates a new puzzle at the user’s request. Selects when a puzzle is solved by SLV at the user’s request. Provides for display and input of grid square data via the keyboard. Provides menu driven interface.

**GEN-** Generates new puzzles randomly by progressively filling in values resulting in a completed, valid Sudoku puzzle, then blanking out some of those values to create an unsolved puzzle.

**SLV-** Solves a puzzle that contains some blanks remaining, by progressively trying all possible valid solution values in the blanks, then backtracking if an invalid solution is encountered.

**FIL-** Handles the file read and write functions to load and save a game grid into the global puzzle grid matrix.

**3.1.2 Architecture Alternatives**

Due to the simplicity of this project, no alternative architectures were considered. The chosen architecture meets the specifications in a straight-forward, understandable manner.

**3.2 Functional Descriptions**

This section describes the functionality for the four main modules: GEN, SLV, FIL, and GUI, in subsections 3.2.1 through 3.2.4, respectively.

**3.2.1 Description for GEN (Generate Puzzle)**

GEN is the functional component which generates a valid Sudoku puzzle. It starts with an empty grid, and randomly fills in legal values, until a complete, legal puzzle is created. It then randomly blanks out some of the squares, and confirms that the puzzle is still uniquely solvable.

All valid Sudoku puzzles have two requirements, as defined by the original game definition. The first requirement is that they have a unique solution. By blanking out too many squares from a correct solution, or by unknowingly blanking out critical squares which determine the uniqueness of the puzzle, it is possible to construct an invalid puzzle which has multiple solutions. To prevent this error from happening, GEN will check the puzzle by using the SLV module to solve it after each addition of blanks, ensuring that the solution is still unique. (SLV returns TRUE only if there is a single solution to the given grid.)

The second Sudoku requirement is that puzzle blanks should be symmetric about either the vertical, the horizontal, or the diagonal axis. Therefore, when blanking squares, a randomly chosen square will be mirrored according to the randomly selected symmetry type, thus adding a pair of blanked squares at each iteration, until the chosen number of blanks has been added.

**3.2.1.1 Processing narrative (PSPEC) for GEN**

The high-level pseudocode for GEN follows:

Fill in initial random grid values

Solve the grid and save the results as the solution

Randomly blank out the desired number of squares to obtain puzzle, checking to ensure that the resulting blanked puzzle has a unique solution

**3.2.1.2 GEN Interface description**

GEN takes as input a 9x9 integer array, and produces in it a new, solvable array with blanked elements, along with a copy of the solved grid containing the correctly filled grid values. A desired number of blanks within the puzzle is input as well.

**3.2.1.3 GEN processing details**

**3.2.1.3.1 GEN Interface details**

The primary function for GEN has the following pseudocode interface:  *GenerateGrid( puzzlegrid (type grid, output);*

*solutiongrid (type grid, output);*

*numberblanks (type integer, input);): boolean*

**3.2.1.3.2 GEN Algorithmic model**

The following pseudocode describes the GEN algorithm:

Randomly fill in the first row and column of *puzzlegrid*

Call *SolveGrid* to solve *puzzlegrid*

Store solved *gamegrid* as the correct solution in *solutiongrid*

Totalblanks = 0

tries = 0;

Randomly select symmetry of vertical, horizontal, diagonal

REPEAT

SaveCopy = puzzlegrid // in case undo needed

RandomlyBlank(puzzlegrid, symmetry)

// blanks 2 squares according to symmetry chosen

ok = SolveGrid(*puzzlegrid)* // will it solve uniquely?

if not ok then

puzzlegrid = SaveCopy

inc(tries);

UNITL (NumBlankSquares(puzzlegrid) >= numberblanks) or (tries > 1000);

After the above executes, there should be a uniquely solvable puzzlegrid, with approximately the desired number of blanks included, found by iteratively trying different blanked out squares. If a blanking selection ever results in a multi-solution grid, that blanking attempt is undone with the line: puzzlegrid = savecopy, and another attempt is made, until the maximum number of retries is met, or the desired number of blanks is met or exceeded.

**3.2.1.3.3 GEN Restrictions/limitations**

No restrictions or limitations have been identified.

**3.2.1.3.4 GEN Local data structures**

Only simple counters and booleans are defined for local use.

**3.2.1.3.5 GEN Performance issues**

The maximum number of tries may need to be adjusted according to testing results, based on the typical speed of the solution algorithm, and the typical wait time needed to find a properly blanked puzzle. The value may need to be increased, if puzzles with enough blanks are not found, or reduced, if the time to generate a puzzle becomes unwieldy.

**3.2.1.3.6 GEN Design constraints**

The algorithm should use helper functions to randomly select a symmetry, to randomly blank a pair of squares (RandomBlank), and to count the current number of blanks in the current grid (NumBlankSquares).

**3.2.2 Description for SLV (Solve Puzzle)**

SLV is the functional component which solves a Sudoku puzzle. It starts with an valid puzzle grid, and progressively fills in squares with legal values, backtracking and retrying values as needed, until a solution is found if one exists. The main SLV function, SolvePuzzle returns TRUE only if there is a single solution to the given grid.

*SolvePuzzle* utilizes a recursive algorithm to implement its backtracking capability. This requires it to pass its input puzzlegrid by value (not reference) so that each instantiation of the function has its own local copy of the grid to manipulate. If it is found that the current instantiation cannot result in a correct puzzle solution, backtracking is handled by trying a different value in a blank square, and recursing again, until all possible values have been tried.

**3.2.2.1 Processing narrative (PSPEC) for SLV**

The high-level pseudocode for SLV follows:

Repeat until solution found or more than one solution found

Choose a set of values to try from the square

with fewest number of possibles

For each value in possibles list

Set the grid location to the value

Recurse with current grid

Record whether successful solution resulted

**3.2.2.2 SLV Interface description**

SLV takes as input a 9x9 integer array having some filled and some unfilled squares, and stops when it has tried all possible solutions for the given set of blanks. It returns TRUE if there is exactly one solution for the puzzle it was given.

**3.2.2.3 SLV processing details**

**3.2.2.3.1 SLV Interface details**

The primary function for SLV has the following pseudocode interface:  *SolveGrid( puzzlegrid (type grid, input)): boolean*

**3.2.2.3.2 SLV Algorithmic model**

The following pseudocode describes the SLV algorithm. **See design constraints in section 3.2.2.3.5 for descriptions of helper functions used below.**:

//NOTE- Requires global numsolns set to 0 on first entry to detect non unique

// solutions when count becomes > 1

FillSingleChoices(puzzlegrid); // fills all open slots having just 1 possible value

IF IsSolved(puzzlegrid) then // all filled in

begin

if (numsolns > 0) then // bad situation, already have one, so not unique

begin

stoplooking := TRUE;

result := FALSE;

end

else // have not found solution yet, so can return true

begin

numsolns++

Solutiongrid = puzzlegrid //save for final solution if no more found

result := TRUE;

end;

end //IF the puzzle has been solved

ELSE // puzzle not yet solved, must find solution

begin

if (not FindFewestChoices(puzzlegrid, row, col, list, numchoices)) then

// no choices left, and not solved, FAIL this try

begin

result := FALSE;

end

else // there are choices, so loop to try recursing on each in r,c

begin

i := 1;

done := FALSE;

got\_one := FALSE;

// try all possibilities,

// making sure the answer is unique

while (not done) and (i<= numchoices) do

begin

// get a random choice in list of possibles

choice := PickOneTrue(list);

list[choice] := FALSE; // erase that choice possibility

Puzzlegrid[row,column] := choice; // plug that choice in

solved := (solvegrid(puzzlegrid));

IF stoplooking then // check boolean condition- no solution

begin

// need quit FAIL

done := TRUE;

got\_one := FALSE;

end

else // continue looping, check for other DONE situation

begin //not told to stop

got\_one := got\_one or solved; // remember solution

i++;

end;

// return TRUE really got a solution

result := got\_one;

end;

end;

**3.2.2.3.3 SLV Restrictions/limitations**

No restrictions or limitations have been identified.

**3.2.2.3.4 SLV Local data structures**

The puzzlegrid passed as a parameter must be by value, not by reference, so that a local copy is maintained for the recursive algorithm.

The global variable *numsolns* is a counter used only by this function, and must be set to 0 before the initial call to the recursive *SolvePuzzle* (not within the function)*.* If numsolns ever becomes greater than 1, the solver will return FALSE meaning there was not a unique solution to the puzzle.

*SolvePuzzle* also maintains a local *list* (array) of the possible values that can be tried in the current puzzle, at the current square. Its purpose is to try each value in the list of possibles, so that the recursive call can be made to check whether the chosen value leads to a solution.

**3.2.2.3.5 SLV Performance issues**

The time it will take for SolvePuzzle to determine whether a unique solution exists for a given puzzle must be tested with various input puzzles. The algorithm necessarily tries all possible solutions, but fortunately the number of possible number variations for any given square are relatively small, even for the most difficult puzzles. It is not anticipated that time to solve will exceed a reasonable limit of 20 to 30 seconds.

**3.2.2.3.6 SLV Design constraints**

The algorithm will use helper functions as follows:

FillSingleChoices(puzzlegrid) -

finds single choice blanks and fills with the value

IsSolved(puzzlegrid) -

checks grid, returns TRUE if there are no blank squares

FindFewestChoices(puzzlegrid : grid- input;

row,column: integer- output;

list: boolean array- output;

numchoices: integer- output): boolean;

finds the BLANK square at grid position *row, column* having the fewest number of choices available > 0. (NOTE: normally, when finding a solution all the single choices will have been filled in, so minimum found here should be 2 choices). Return FALSE if any blank square has a min of 0, meaning there are NO possible choices for some blank and the current solution path is unsolvable. *Numchoices* is the count of how many possibles there are at *row,column,* and *list* contains the *numchoices* number of values for the choices to be tried. *List* is a boolean array, mapped to the index values; TRUE at an index means it is a possible value, FALSE is not a valid choice.

PickOneTrue(l: list): integer;

return a randomly chosen index for one of the TRUE occurences in list, signifying a possible value within the list.

**3.2.3 Description for FIL (File Handler)**

The FIL module has the responsibility to read and write puzzle grid files in ASCII format. Its file structure is defined in section 2.4.2.1 describing game files.

**3.2.3.1 Processing narrative (PSPEC) for FIL**

The high-level pseudocode for FIL follows:

*Read File:*

Open file and attempt to read 9 lines of text data into parameter grid

*Write File*

Open file and attempt to write 9 lines of text data from the parameter grid

**3.2.3.2 FIL Interface description**

FIL contains a *ReadFile* and a *WriteFile* function. *ReadFile* takes as input a 9x9 integer grid array, and a filename, and returns TRUE if successful in writing the named file. Likewise, *WriteFile* takes the same parameters, and returns TRUE is the new file is successfully written.

**3.2.3.3 FIL processing details**

**3.2.3.3.1 Interface description**

The two functions for FIL have the following pseudocode interface:

*ReadFile( puzzlegrid (type grid, output);*

*filename (type String, input);) : boolean;*

and

*WriteFile( puzzlegrid (type grid, input);*

*filename (type String, input);) : boolean;*

**3.2.3.3.2 Algorithmic model**

The following pseudocode describes the FIL read and write algorithms:

*ReadFile:*

Open file designated by *FileName* for READ access; report

any file errors on screen and return FALSE if any errors

For i=1 to 9

Read one line from file into Buffer string;

For k = 1 to 9

If Buffer [2\*(i-1)+1] == ‘-‘ then

puzzlegrid[i-1,j-1] = - (Buffer[2\*(i-1)]-48)

ELSE

puzzlegrid[i-1,j-1] = (Buffer[2\*(i-1)]-48)

Close file

Return TRUE

*WriteFile:*

Open file designated by *FileName* for WRITE access; report

any file errors on screen and return FALSE if any errors

For i=1 to 9

For k = 1 to 9

Buffer[2\*(k-1)]=ABS(puzzlegrid[i-1,k-1]+48)

If puzzlegrid[i-1,k-1] < 0 then

Buffer[2\*(k-1)+1]=’-‘

Else

Buffer[2\*(k-1)+1]=’+‘

Write Buffer to file

Close file

Return TRUE

**3.2.3.3.3 FIL Restrictions/limitations**

No restrictions or limitations have been identified.

**3.2.3.3.4 FIL Local data structures**

Only temporary file structures are defined for local use.

**3.2.3.3.5 FIL Performance issues**

FIL will be reading and writing very small files of 162 characters each (81 square, 2 characters per square), so that no performance issues are anticipated for this module.

**3.2.3.3.6 FIL Design constraints**

The FIL algorithm must use file I/O error handling (TRY/CATCH for example) to prevent program crashes due to file access problems.

**3.2.4 Description for GUI (Graphical User Interface)**

The GUI module has the responsibility for allowing the user to interact with the program. **The primary description of the screen layout and visual elements is contained in section 4.0 of this document.** This section contains the descriptions of the functional capabilities and processing provided by GUI.

**3.2.4.1 Processing narrative (PSPEC) for GUI**

The high-level pseudocode for GUI follows:

Wait for user to start a new puzzle, load a saved puzzle, or enter a puzzle.

If they wish for a new one, then one will be generated error free call GEN, display returned output..

If they choose to load one, call FIL for loading.

If they choose to enter one, the state for entering puzzles manually will begin, ending with the user clicking the submit option and errors being checked for. If errors are found the user will be prompted to correct them. Should the user attempt to exit the program during this state, before saving, they will be prompted to save or trash what they have entered.

After puzzle entrance, loading, or generation, the main game will start. Fixed values will not be editable, but others will. Navigation provided on mouse-click, or via the arrow and tab keys. Generate the list for each square when the user tabs or clicks into it, and if hints are to be displayed, show the list in the hints section.

The user can clear any non-fixed value with <space> <backspace> or <delete> or by overwriting it with a new number, this will instantly change the stored value for that slot.. If the show errors option is selected, the hints box will display a rabbit dying when the erroneous entry is submitted.

Upon submission of a completed puzzle the program will then check for errors and display a win or loss based on the result, by calling slv and checking for proper input

**3.2.4.2 GUI Interface description**

The interface for the GUI will be a standard mouse and keyboard for input and a monitor for output.

**3.2.4.3 GUI processing details**

The GUI will process the user input in a straightforward way, broken down into the following parts.

**3.2.4.3.1 Interface description**

The interface to the program will be a simple set of connections between the GUI built with a separate GUI builder and the code for the implementation of the code. Buttons will be mapped to specified actions and the correct modules and procedures for the actions will be called.

**3.2.4.3.2 Algorithmic model**

As described in the definition for the pseudocode, the gui’s processing will be very even driven, either based on waiting for input from the user or the evaluation of the input by the core program.

Wait for user to start game. Call module for input, either gen, fil or the input mode.

When puzzle is loaded/entered/generated display

Check for hints and error display

REPEAT( until submission for verification) :

Wait for user to make input, generate list for any square entered by user

Display hints if to be displayed, display errors if detected and to be displayed

Allow numbers to be input in squares, and for squares to be cleared, except the uneditable squares.

END

**3.2.4.3.3 Restrictions/limitations**

The GUI will be able to handle all interaction between the user and the game, short of them opening a .SUD file and editing the contents for their own favor.

**3.2.4.3.4 Local data structures**

Only those needed for driving the actual display of the GUI will be releveant, others will be subject to the program’s implementation. One list relevant to the highlighted square will always be stored (and if to show hints, displayed) locally. The local data structures of the GUI will be based on what GUI program is used, so long as the appearance is that of a standard windows program, it will be acceptable.

**3.2.4.3.5 Performance issues**

The GUI will be relatively lightweight, as hardly any animations are in the program, the main concern regarding Performance issues are the algorithms within the program that allow for the input or output to the GUI to be driven.

**3.2.4.3.6 Design constraints**

The GUI will be displayed as any standard windows program, and the design must fit the requirements.

**3.3 Software Interface Description**

There are no outside interfaces for the *SSFW* program

**3.3.1 External machine interfaces**

Not applicable.

**3.3.2 External system interfaces**

Not applicable.

**3.3.3 Human interface**

*SSFW* uses a standard Windows program interface, with standard GUI features. See Section 4.0 for additional GUI details.

**4.0 User interface design**

The user interface will use simple windows controls to effectively display the game to the user. The goal is to make it as crisp and clear as possible. The game will be presented in the main window area, and options will be selected through menus.

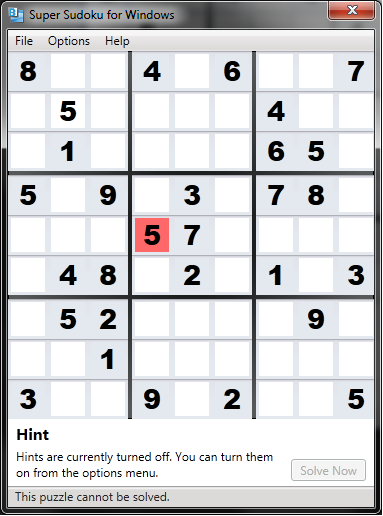
**4.1 Description of the user interface**

The main window will display a grid of 81 boxes to enter game information into. Nine sub grids are separated visually by using black lines to separate them, similar to how Sudoku games are laid out on paper. Static fields have a grey background and aren’t editable or focusable. These are only shown on the game mode, not the mode that allows the user to input a custom game mode.

At the bottom of the screen there will be a hints box, as well as a “Solve Now” button. Below this there will be a status bar giving information about the puzzle to the user.

**4.1.1 Screen images**

The GUI can be seen below. The window size is 372x505 pixels, supporting 800x600 or higher resolution.



**4.1.2 Objects and actions**

The menus will act as follows, with keyboard shortcuts available for all menu options.

1. File menu – Access main game features.
   1. New – Start a new game.  
      If a game is in progress prompt the user to save the current game if they choose too.  
      Ask the user to choose a difficulty using a dialog box with radio boxes.
   2. Load – Open a saved file.  
      If a game is in progress prompt the user to save the current game if they choose too.  
      Only show files with the extension ‘.SUD’  
      Loaded games will load the game at the same point of play as they were saved at and continue the game timer.
   3. Save – Save a game file.  
      Saves the game at the current state it’s in.  
      Automatically add the extension ‘.SUD’ to the saved file.
   4. Save Game Unsolved – Saves a copy the game without any of the fields entered by the user.  
      The game timer will be saved as 0, i.e. no time used.
   5. Enter Puzzle – Allow the user to enter a puzzle from an outside source.  
      If a game is in progress prompt the user to save the current game if they choose too.
   6. Exit – Exit the program and return to Windows.  
      If a game is in progress prompt the user to save the current game if they choose too.
2. Options menu – Access options.
   1. Always Show Hints – Shows hints on the box at the bottom of the window.  
      Will be implemented as a standard windows menu option – displaying a check box on the left if activated.
   2. Show Errors – Toggles showing errors in input to the user.  
      Will be implemented as a standard windows menu option – displaying a check box on the left if activated.
3. Help – Get help.
   1. Game Rules – Will pop up a dialogue containing the game rules.
   2. About – Displays a dialogue displaying the version of the game and any copyright information.

When the user clicks any box he or she can then enter a number, which replaces any number already there. Only numbers from 1-9 will change the value. If <DELETE>, <BACKSPACE>, or <SPACEBAR> is entered the box will be cleared of any value that was already there. Additionally, the possible values will be calculated for the current square and be optionally displayed to the user at the bottom of the screen in the hints section, based on if the menu option “Show Hints” is active. If the user enters an invalid entry and “Show Errors” is enabled, then the box will turn red and an animation of someone killing a small rabbit will be played.

The arrow keys and tab key will allow the user to move around between boxes. The tab key will wrap around when the end of the row of boxes is selected.

When the user right clicks on a square a menu will pop up with the following actions:

1. Show Hint – This will show the hint for the selected box as long as it remains selected.
2. Fill In Answer – Places the correct value in the square. Correct value is determined by solving the puzzle ignoring all values entered by the user and inputting the right value to that square only. The value could conflict with an erroneously entered value the user already inputted some where else.

At the bottom of the screen there will also be a “Solve Now” button that will solve the entire puzzle. If it is not possible to solve the puzzle at the current state, then the button will be disabled. Clicking the button will prompt the user before solving the entire puzzle.

Another state will be activated when the user is inputting a puzzle from an outside source. The window won’t contain the hint box, and will have a button that says “Entry Complete” instead of solve now. Clicking the “Entry Complete” button will allow the user to save the puzzle, and then enter regular game mode with all the entered fields greyed out and unchangeable. If the puzzle is unsolvable, a box will pop up and allow the user to edit the puzzle again.

There will also be a status bar that will contain game information at the bottom of the window. If the puzzle is ever unsolvable, the status bar will read “This puzzle cannot be solved.”

Once all the fields are entered, check for errors. If there are none, a box will show up displaying total game time. If there are errors ask the user if he/she wants to see them, and set the show errors option to on if he/she selects yes. If the user already has the show errors option on simply tell the user there are errors. This box will not be shown again till all the errors are fixed.

**4.2 Interface design rules**

The user interface will strictly follow windows design standards. Many games on windows have a unique user interface, but this game will simply use built in Windows controls to implement the game elements. Shortcuts will be provided, at no additional charge, allowing keyboard based entry and game play.

**4.3 Components available**

There are three main GUI components. The first are the menu elements, which will trigger actions in the code. The second is the game board which is defined as a grid of text boxes for user input. The last is the hints section at the bottom.

**4.4 UIDS description**

The user interface was developed with a GUI builder and will be tied into the code through simple actions.

**5.0 Restrictions, limitations, and constraints**

None identified.

**6.0 Test Plan**

The test strategy and preliminary test case specification are presented in this section. It should describe how the code will be tested in order to ensure that all SRS requirements are met.

**6.1 Classes of tests**

The types of tests to be conducted are specified, including as much detail as is possible at this stage. This should be a high-level view, a few sentences describing the contents of sections 6.1.1 and 6.1.2.

**6.1.1 White-box Tests**

Specifies what tests will be performed by the developers during the implementation of each of the modules. White-box tests involve knowledge of the internal workings of the code being created, and should be done by the persons responsible for those code sections. This section should be a list of each of the functions that will be tested at a module level, most likely using special test code that will exercise the specific functions of the code and confirm that it is returning the expected results at intermediate phases. Tests should be listed in a table for each module, numbered uniquely, and described in sufficient detail that the programmer of the module would be able to perform the test and confirm the expected results. For example:

|  |  |
| --- | --- |
| **TESTS FOR MODULE ABC** | **EXPECTED RESULTS** |
| 1.1 The xyz function will be called indpendently from a test version of *main* with zero values for all parameters. | Function xyz should return FALSE, and also display a message indicating ‘invalid input parameters: parameter a is zero’. |
| 1.2 The xyz function will be called with the range parameters set to maxint. | Function xyz should limit the range parameters to their maximum legal values of 1000, and process the data normally, returning TRUE |
| etc... |  |

A table should be provided for each of the modules.

**6.1.2 Black-box Tests**

Specifies what tests will be performed by the developers from the user-interface perspective, without knowledge of the internal workings of the code. These tests should refer to the elements of the user interface as designed and described in Section 4.0, and, as with the White-box tests, use a table to show the test instructions and the expected test results. **Be sure that the tests in this table are numbered so as to be distinguishable from the white-box tests.**

**6.2 Test Coverage Listing**

This listing should contain all of the numbered SRS requirements, alongside the one or more test numbers which are involved with each requirement. For example:

|  |  |
| --- | --- |
| **Requirement** | **Tests Involved** |
| 1.1.1 | 2.3.4 |
| 1.1.2 | 2.6.5, 1.2.1 |
| 1.2.1 | 4.2.1, 3.2.1, 5.2.2 |

**7.0 Appendices**

Presents information that supplements the design specification.

**7.1 Requirements traceability matrix**

A matrix that traces stated components and data structures to software requirements is developed. A table should be provided similar to the following, see assignment instructions for details:

:

|  |  |  |  |
| --- | --- | --- | --- |
| **Requirement** | **ABC** | **DEF** | **GHI** |
| 1.1.1 GUI buttons | **X** |  |  |
| 1.1.2 Read file |  | ReadFile | **X** |
| 1.2.1 Input check | **X** |  | **X** |

**7.2 Packaging and installation issues**

None identified.

**7.3 Design metrics to be used**

None identified.

**7.4 Supplementary information (as required)**

None identified.