**SolidFire Chess Service Test Strategy**

**(A SolidFire/NetApp employment interviewee Software Assignment)**

**Author/Candidate: Dan Doran, Boulder CO.**

**Contact:** [**dancdoran@gmail.com**](mailto:dancdoran@gmail.com) **, 303-709-6746**

**Date: 11/16/2016**

**Index:**

* [**Test Environment Requirements**](#test_env_requirements)
* [**Test Program Requirements and Objectives**](#test_program_requirements)
* [**Testing Scope**](#section_1_testing_scope)
* [**Problems Found with Assignment/Requirements Document**](#section_2_spec_problems)
* [**General Testing Strategy**](#section_3_gen_testing_strategy)
* [**Testing Implementation**](#section_4_testing_implementation)
* [**How To Run The Tests**](#section_5_how_to_run_tests)
* [**Bugs Found in SolidFire Chess Service API ‘MakeMove’ method using tests**](#section_6_API_bugs_found)
* [**Known Deficiencies with Testing Implementation**](#section_7_known_deficiencies)
* [**Programmer’s Notes**](#section_8_programmers_notes)
* [**Summary**](#section_9_summary)

**TEST ENVIRONMENT REQUIREMENTS**

The supplied test program requires Linux Ubuntu v14.04 or newer, using Python v3.1 or newer. It was tested using the Windows 10 Bash Ubuntu shell on my windows laptop. The “requests” python module should be the only one that doesn’t come with the standard python v3.4 installation and may require a pip installation.

Test programs also runs under windows (cmd terminal) with python v3.1 or newer plus “requests” module installed.

No effort was made to be backward-compatible with Python 2.x versions.

**TEST PROGRAM REQUIREMENTS and OBJECTIVES:**

1. **Provide reasonably complete test coverage program(s) to test the SolidFire Chess Service (SFCS) API ‘MakeMove’ method.**
2. **Create & return a human-readable test report upon test completion.**
3. **Include a log file of all API requests and responses**
4. **Include this report detailing testing strategies, known deficiencies, and found API bugs**
5. **Include a README file for folks in a hurry.**

**SECTION 1: TESTING SCOPE**

**SFCS Items “In-Scope” for the test program:**

* **Supplied SFCS API Specification insufficiencies or ambiguities**
* **SFCS API “MakeMove” requests and responses**
* **Game Flow – Verify White goes first and play alternates afterward. Verify playerState both on requests and responses.**
* **Out-of-bounds moves & boardState locs & types, both as illegal requests and checking for bogus response values.**
* **Chess Piece Move Legality, including castling, pawn promotion (w/ & wo capture) and en passant.**
* **Response Game States – In-Progress (blank “”), check, checkmate, stalemate**
* **API Specified Error Conditions – Player Error, Move Error, Board Error, Unknown API Errors**
* **Platform support - Tested on both windows and Ubuntu Linux v14.04**

**Items “Out-of-Scope” for this test program:**

* **Full Board examination – The test program doesn’t verify game situations such as ‘check’ & ‘checkmate’ or validate piece move legality during test case generation. It remains up to the tester to stage these situations, anticipate the API response gameState, and investigate if the response is not what was expected to determine if it is a test case issue or a response API bug.**
* **No Full-Game start-to-end system testing. Insufficient time to code up realistic full games which would require starting with a new game and supplying all of the required moves to get to an endgame state. Tests are run with suites of pre-coded boardStates + a “move” in an API request, checking for critical Piece strings or expected error code in the response. (*Programmer’s Note – If this were a “real” assignment I would also write an interface between one or more free chess playing programs and the SFCS API, log the request and response outputs, recording the number of moves, outcomes, and scanning for returned errors, abnormal completions, and suspiciously short games. We should be able to run many thousands of complete games/day and also obtain realistic performance metrics if we instrumented for cpu time/cycles and memory stats.)***
* **No post-“check” follow-up validation to ensure the check situation has been legally removed by the opposing player, or “checkmate” conceded.**
* **No “Threefold Repetition” testing for endgame**
* **No “Fifty Move Rule” or “Insufficient Material Rule” testing for endgame**
* **No Performance Testing. No Memory, disk space or runtime requirements specified or tested.**
* **No Reliability testing of the SFCS system – No checks for memory leaks or other resource threats for long term non-stop use of the SFCI.**
* **No Load testing of API for multiple simultaneous users.**
* **No Security testing of the API**
* **Network Functionality. No tests to see if network dropped while SFCS was in use.**
* **No GUI support or testing. Ascii result report and API log file only. (I do however have some ascii chess board art in the test case generation program, ‘RequestGen.py’)**

**SECTION 2 - SFCS API SPECIFICATION DOCUMENT PROBLEMS FOUND:**

1. **PROBLEM:** [**Link in specification to Algebraic Test Notation**](http://renaissanceknights.org/IL%20Scholastic/Handouts/Handouts%20PDFs/AlgebraicChessNotationintroduction.pdf) **did \*not\* include En-Passant notation. So all legal chess moves are not covered.**

**POSSIBLE RESOLUTION: I found this Algebraic Chess Notation guide to be more useful, containing notation for all legal moves. Perhaps it should be used in the specification document:** [**http://www.chesshouse.com/how\_to\_read\_and\_write\_chess\_notation\_a/166.htm**](http://www.chesshouse.com/how_to_read_and_write_chess_notation_a/166.htm)

1. **PROBLEM: How do we begin a game? No methods in API to specify who we wish to invite for a game, how that contact is made, responded to (positively or negatively), How to select who goes first (who is white player), and how to get a setup initial board without typing an entire valid starting boardState value.**

**POSSIBLE RESOLUTION: Possibly covered under other methods.**

1. **PROBLEM: No way in API Spec to concede a game when it’s as good as lost.**

**POSSIBLE RESOLUTION: Add a “concede” option to legal “move” values.**

1. **PROBLEM: How do we end a game? No methods in API to specify what we wish to do next (Request another game? Leave? Send a Congratulation message? Send an Insult and then leave?).**

**POSSIBLE RESOLUTION: Add more “move” values in addition to “concede” in the MakeMove method. E.g. “play\_another”, “quit”, or “message: <insult\_goes\_here>”**

**SECTION 3: GENERAL TESTING STRATEGY**

***Programmer’s Note – My experience with software testing is that the testing framework functionality and details are more important to get right than immediately having sufficient amounts of test coverage cases. The ability to quickly add cases to increase coverage more than compensates for the initial framework time up front, especially for products under continuous development. So I spent a lot of time trying to build a somewhat generic testing framework, and a second program to interactively create additional cases in less than five minutes and automatically add them to the library of functional and expected error tests. This was what consumed the bulk of my development time.***

**Since we only have one API method to test (MakeMove), testing will concentrate on what happens the API MakeMove method is given validly formatted JSON-RPC requests with different values of the three params - “boardState”, “move” & “playerState”.**

**The possible legal permutations of the “boardState” list are as good as infinite, overflowing a 64-bit int. So only the “type” and “loc” bounds of the individual Pieces can be reasonably covered for valid and legal numbers of “types” (e.g. not more than 1 kings per side), and all “locs” being both unique and within the a1-h8 chess board bounds. Error tests can be written or modified for bogus “type”s and out-of-bound “loc”s.**

**Simple piece “move” permutations are not nearly as infinite. All piece types could in theory be tested exhaustively in all 8 directions, some legal, some not. Knights have 4 directions \* 2 turn options. Move distances could be either 1 (minimum move except knight) to the max move distance limit (1 for Kings, 1 for pawns, 2 in start row, 7 for Queens, Rooks & Bishops, and NA for Knights). For our purposes we’d assume all intermediate legal values should work and only test the maximum move in all directions. So, 6 piece types \* 8 directions \* 2 colors \* 2 distances (min & max) = 184 tests give or take to cover the breadth of legal piece moves.**

**Adding capture permutations makes the test space explode. Limited capture moves for each piece will have to suffice.**

**Move errors beyond basic direction tests (e.g. beyond range, through own or opponent pieces, off of move axis, locs occupied by same color pieces, out-of-bound locs, etc.) also make the test space explode. Sampling will have to suffice foe coverage here.**

**Castling moves should be tested for both directions and player colors. (4 total).**

**En Passant and pawn promotion are tested with their unique Algebraic Test Notation. They should be tested both with and without captures.**

**Both “playerState” values (“w” & “b”) will be tested, along with empty and bogus values.**

**SECTION 4: TESTING STRATEGY IMPLEMENTATION**

1. **“Test\_Boards” repository. These are starting API Request boardState lists to stage & facilitate all kinds of tests. Some of these boardStates are realistic for normal play, while others are contrived to trigger appropriate API Errors. Having starting boards we can use for more than one test speeds test case development by not needing to populate an entire setup board every time. Any newly-created boards are optionally automatically added to this repository. The following sub-dirs/categories should cover all of our testing requirements:**
   1. **“Test\_Boards/Start\_Game” - Category that includes full & partial starting chess board setups, all legal.**
   2. **“Test\_Boards/Invalid\_Boards” – Boards that should return API board Errors. This includes boards with too many of any given piece, less than two Kings, more than one piece on a square, out-of-bounds locs and unknown pieces.**
   3. **“Test\_Boards/Piece\_Moves” – Setup for legal and illegal moves for each piece, either from a starting position or advanced positions. Castling, pawn-promotion & En Passant setups also live here.**
   4. **“Test\_Boards/Piece\_Captures” – Setups for captures from each piece type.**
   5. **“Test\_Boards/End\_Game” – Setups for check, checkmate & stalemate API response gameStates. Piece move can include a capture or not.**
2. **Test Case Definition File Repository. This is where our functional (expected passing) tests and Expected Error tests live. Either in ‘expPassTestDir’ or ‘expFailTesstDir’. All new tests created will be put in one of these two directories.**
3. **‘RequestGen.py’ test case rapid-generation script. It does the following:**
   1. **Prompts tester whether they want to create a functional test (expected to return a full API response), or an expected error test (expected to return an API error code). The contents of the produced test case definition file and the test repository location will be determined by this choice.**
   2. **Prompts for test description and name, which becomes the root file name in the test case repository.**
   3. **Prompts user whether they want to browse and use an existing starting board from the Test\_Boards repository, or wish to create a new starting board.**
   4. **For the new-board case, works interactively with tester to populate an empty board and provides opportunity to save the new board to the Test\_Boards repository.**
   5. **Prompts tester for color playing, type of move, which piece to move, and the destination square for the move. Captured pieces are inferred.**
   6. **Prompts tester for either the expected response API error code for expected error cases, or for the expected reponse playerState and gameState values for functional tests.**
   7. **For functional tests, infers a list of request board pieces that should be \*missing\* in the response board – piece moved from its starting loc (rook+king original locs in castling), plus any captured piece.**
   8. **For functional tests, infers a list of “new” pieces expected in the response – the moved piece in its new loc for most moves, a new queen object for pawn-promotion, and rook+king new locs for castling.**
   9. **Writes the new completed test definition file to the test case repository.**
4. **‘Run\_SFCI\_Tests.py’ script. This is the test script intended to be run either interactively from the command line or in regression-test batch mode from a cron job or code submission request. Default is to run all tests in the test case repository, but single tests or a file containing a list of test cases can be provided. It does the following actions and checks:**
   1. **Gathers a list of all cases expected to run, and runs them sequentially (no multi-processing/multi-threading support).**
   2. **Submits each test case request to the SolidFire Chess Service API URL.**
   3. **Performs checks on the API response to determine if the case passed or failed (in an unexpected manner for expected fail cases).**
   4. **For expected fail cases, the only check is for the expected response error code.**
   5. **For functional tests, the API response is checked for:**
      1. **Expected response playerState (opposite of request player)**
      2. **Expected response gameState, based on what test was trying to accomplish.**
      3. **Verifies any moved pieces or captured pieces are no longer on the response board.**
      4. **Verifies the moved piece is in its expected new loc. Does two checks for castling moves, and makes sure the pawn was changed to queen for pawn promotion moves.**
      5. **Verifies all other pieces came back unchanged and no new pieces were introduced.**
   6. **Writes a Date/Time-stamped summary report and API log file for the test run.**

**SECTION 5: HOW TO RUN THE TESTS**

1. **How to create a new expected error or functional test case (optional)**
   1. **Usage: python RequestGen.py**
   2. **Interactively follow prompts.**
2. **How to Run Test Cases**
   1. **USAGE: python Run\_SFCS\_Tests.py [-l Path\_to\_list\_of\_testcase\_files] [-s Path\_to\_single\_test\_file]**

**Path\_to\_list\_of\_testcase\_files: optional ascii file with one testcase name on each line**

**Path\_to\_single\_test\_file: optional single test case definition file**

**(Default is to run all tests in the test case directories.)**

* 1. **Read Summary in generated Test\_Results\_<date>\_<time>.report file**
  2. **Optionally read the API requests & responses in the Test\_API\_<date>\_<time>.log file**

**SECTION 6: BUGS FOUND IN SF CHESS SERVICE API ‘MakeMove’ METHOD**

1. **No Support for Legal Castling Moves (‘0-0’ KingSide, ‘0-0-0’ QueenSide). API returns 'Invalid move string.', 'code': -32020’**
2. **No Support for Pawn En Passant moves (<pawn column>’x’<destination square>’(ep) e.g. exd3(ep) ). API Returns 'Invalid move string.', 'code': -32020’**
3. **Allows Black Player to move first on Starting Board**
4. **Returns Move Error (Move cannot be made., ‘code’: -32020) rather than Player Error for out-of-turn move. (I was unable to trigger a 'Invalid player error.', 'code': -32010’). It could arguably be either, but why have a playerState API error code if you can’t trigger it?**
5. **No “Stalemate” gameState returned for a board containing only two kings. Got a continue (“”) gameState response even though there is no way for either side to win. I didn’t have time to further characterize what it took to trigger a ‘stalemate’ response.**

**SECTION 7: KNOWN DEFICIENCIES IN SOLUTION**

**Test Case Definition File name and ‘testName’ entry inside file are allowed to get out-of-sync. Arguably the ‘testName’ entry line is redundant information, but it’s also good to have if you’re inspecting or editing an unexpected result from a test case.**

**Description limited to single line. Not a big deal for such a well-defined testing space**

**No consistent error return codes if program stumbles while running in batch mode. Should set up an exception try/expect hierarchy to ensure reliable unexpected failure logging.**

**No provision to mark a failing test as a known bug, preferably with an associated/linked CR. Currently requires the tester to “know” which cases fail due to known issues to prevent re-investigation of failures. This is a major shortcoming in a real test environment since low-priority bugs may go unfixed for weeks or months.**

**Optional Test Case List file (Run\_SFCS\_Tests.py –l option) must have either windows or linux path separators that agrees with the platform it’s run on. Should let them run on either platform with either path convention.**

**No multi-threading or multi-CPU support to speed up regression runs.**

**Unable to trigger “Unknown API Failure” code (-32030). Mangled request syntax triggers JSON errors, not Unknown API Failure.**

**Code could be tighter in sections. Violated python “never write it twice” maxim more than I should have. I also found myself converting between List/Tuple/Dict/int representations and their string counterparts more than I liked.**

**Due to multiple program iterations there may be some dead/unreachable code remaining in the programs, or redundant sanity checks that will never be exercised. I didn’t use any static program analysis tools like Coverity or runtime code coverage tools like BullsEye.**

**Test framework is not as abstract or generic in its design as a more general test solution should be. It would take a lot of rework to make this optionally work on say, checkers or backgammon. My design would be quite different, possibly object-oriented, if the API covered more than chess.**

**Program lacks any report file archiving or management to facilitate historical trend data mining and keep the home directory from filling up with .report & .log files.**

**Test Case creation with ‘RequestGen.py’ could go faster w/ fewer keystrokes with the following improvements:**

* **Instead of repeatedly prompting tester to verify a board or move is what they intended, I should allow tester to quit (‘q’) or go back a step (‘b’) at any point.**
* **For move specification should allow tester to do the minimal input to identify the piece to move – just the color plus a single piece character when start board only contains one, or piece char + column letter for multiple move piece candidates. In the rare case where there are two or more of the same colored piece types in one column, fall back on the full loc.**
* **Should infer a ‘capture’ move from the destination square alone.**
* **Over-the-top solution would be a drag’n’drop board/move creation GUI.**

**Finally, there may well be some bugs remaining for edge cases where I didn’t create a test case file.**

**SECTION 8: PROGRAMMER’S NOTES**

1. **Things I had to learn for this Solution: I was new to JSON\_RPC web API methods & syntax. It took a fair amount of trial & error to finally get responses reliably. Also took the opportunity to set up and learn MS Visual Studio on my laptop for code writing & debugging, which consumed some time that would have otherwise been spent writing code. I alternated between Visual Studio and the Windows 10 Ubuntu Bash shell for this project.**
2. **Non-deterministic response API list ordering. I quickly found that a simple ‘diff’ of the API response would not work to determine if a case passed. The same request would generate responses with the same content, but not the same ordering. My solution for this was to make three lists – The original request boardState (needed anyway), a list of piece object(s) which should \*not\* be in the response board, and a list of new object(s) expected in the response list. Combining these three lists made an expected boardState and piece count that could be iterated & counted for comparison to the response boardstate, yielding a complete response analysis.**
3. **Testing Framework Development vs. Test Case coverage: As I stated in the General Testing Strategy section, developing a flexible test framework and a tool for rapid test case creation took priority over a simpler solution with less comprehensive response checking. Putting test case data inside the code was never considered since comprehensive testing solutions need to be data driven with the test framework code only touched for enhancements & bug fixes.**
4. **Procedural vs. Object-Oriented Design: I pondered whether the test framework solution lent itself to OOP and couldn’t come up with enough abstractions or potential inherited commonalities to justify an Object Oriented Design. I initially started on an abstract base class piece object with each piece type being a concrete class containing move methods and the like. However since the tester ultimately decided on the move destination (legal or not) for a given piece and the SFCS API determined whether it was legal, such an abstraction would make more sense on the other side of the API. I did initially use a printer class with methods to decide what combination of terminal, report file & log file to send messages to. However it had a lot of undesirable side effects like disabling regular “print()”, and having to manually insert all newline chars. Then at some point it decided to just stop sending stdout to the report files in the middle of the test run giving me stunted reports. Rather than debug stream management, I abandoned the class in favor of a few simple print functions to control the reporting streams.**

**SECTION 9: SUMMARY**

**I’d like to thank the employees at SolidFire who gave me the opportunity to audition a solution and show my experience. I was a bit rusty after a long programming sabbatical, but this exercise was a nice boot-camp experience to bring it all back and learn some new techniques and web API techniques as well. I hope you enjoy experimenting with my solution and wish all of you well with your new NetApp future. Please don’t hesitate to call (303-709-6746) or email (**[**dancdoran@gmail.com**](mailto:dancdoran@gmail.com)**) with any questions or bug reports.**

**Dan Doran**