Report Three: Software Validation

Corina Ciobanu Iskander Orazbekov Mir Sahin
Paul Grigoras Radu Baltean-Lugojan

December 9, 2011

Abstract

proTrade is a tennis trading environment which delivers the information a trader requires to place bets. Betting functionality, linked to the user's bank account is also provided. Testing and validation are crucial in order to deliver a reliable and secure application and have been used extensively.

Contents

1 Introduction

Tennis trading is a steadily growing market on the Betfair Exchange, with more than 70% of bets being placed in-play. In order to maintain market liquidity, exchanges must attract customers for example, by supplying them with better tools. By providing more information and better visualization techniques such a tool can help the trader improve his understanding and predict the market evolution, which should (potentially) lead to an increased profit.

For tennis in particular, the information required to predict/understand the market evolution is the score, player statistics, potentially a live video feed of the match and, of course, the market data (evolution of betting odds). Ideally, this is desired for both historical and live matches.

At the moment, no application provides all this information. A number of solutions exist which allow visualization of historical market data, but they generally lack the more specific, tennis related data. For example in Fracsoft () market data is not correlated with match data (scores, player statistics). BetAngel() provides some tennis related data and prediction, but relies on the user to input the score, by pushing buttons, which is not suited for the speed at which the market can move. Ideally all the information should be automatically provided.

proTrade means to fill this gap, by providing all the information, betting and prediction functionalities for both historical and live matches, in an entirely automated fashion.

Due to the ambitious nature of the project and the high risk associated with some user actions (e.g. placing bets with real money) in order to ensure reliability and stability testing and other validation methods have been used exstensively.

2 Testing

Testing has been used throughout all stages of the project starting with iteration 4 and more extensively with the introduction of the Continuous Integration server in iteration 5.

As suggested in [1] we have used the tests to guide our design, based on the principle that if a project is well designed it should be easy to write meaningful tests for it.

Unit testing has been used to rapidly test small portions of code while **acceptance tests** have been used to test a system feature from front to back. **TDD tests** (tests written before features) are expected to fail when they are written and should pass once the feature has been completely implemented. Once it passes, a test is transferred to the regression suite. Naturally, a failing **regression test** indicates a break in previous functionality (regression).

2.1 Unit Testing

Because of the ease of use, unit testing has been adopted early on in the development process and it has been of great use in identifying bugs and ensuring correctness of the Match API which manages match data such as score and statistics.

Since tennis scoring rules are peculiar (for example points are counted 15, 30, 40, AD instead of just 1,2,3,4) and tests are particularly easy to write we adopted a TDD aproach. For example a simple initial test was written to check the correct outcome of adding four consecutive points to one of the players. This should lead to him winning the game and the points score being reset. Furthermore the other player's score should always be zero. The code for this is presented in 1.

```
1   @Test
2   public void fortyZeroWin(){
3     int expectedPoints[] = {15,30,40};
4     for (int i=0;i<3;i++){
5       score.addPlayerTwoPoint();
6       assertGameScoreIs(0,0);
7       assertPointsScoreIs(0, expectedPoints[i]);
8     }
9     score.addPlayerTwoPoint();
10    assertGameScoreIs(0,1);
11    assertPointsScoreIs(0,0);
12  }
```

Listing 1: Initial failing test for score. The initialisation of the score object is handled in an abstract super class which also provides assertSetScoreIs() and assertGamesScoreIs().

Having first written the test and ensured it failed, we then proceeded to implementing the methods that would make it pass. We adopted this approach for the whole Match API and for the Prediction API (used to predict the evolution of a match based on current score and player statistics).

This approach helped us identify numerous bugs early on and guided us towards a better overal design of the APIs.

e.g. a function for correctly setting a set score to a certain value was not initially provided, but since while writing the tests the need for such a function became obvious, it was included and tested ** separate tests that measure progress from tests that catch regression **

2.2 Acceptance Testing

Acceptance tests were designed separately and were meant to test a particular function of the system, starting from the front-end (e.g. finding a pushing a button on the UI) to the back-end (e.g. connecting to the Betfair API to authorize a login request).

For performance reasons these were ran separately from unit tests since the UI operations tend to be slow.

For example the test in 2 checks the login functionality: the user should fill in their Betfair account and password and click on the login button. The login attempt is checked against the Betfair API and a label is updated to indicate success or failure. Obviously, an attempt to login with the test account should result in a success message being displayed.

```
1     @Test
2     public void correctLoginSuccess() throws Exception {
3         SWTBotText username = bot.text("username");
4         username.setText(Main.getTestUsername());
5         SWTBotText password = bot.text("password");
6         password.setText(Main.getTestPassword());
7         loginButton.click();
8         SWTBotLabel success = bot.label(LoginShell.SUCCESS);
9         assertNotNull(success);
10    }
```

Listing 2: Initial failing test for the login window. The username and password for the test account are read and decrypted from a local config file by the Main class. Using the UI bot we then fill the data in on the login window and click the login button.

We have adopted a similar approach for all features.

However, due to limitations in the SWTBot API some features have proven impossible to test. For example we have not found a way to test the functions

of a context menu (pop up) or a progress bar.

** new acceptance tests will not pass until the feature is implemented

acceptance tests for completed features catch regressions and should always pass (might take longer to run)

once an acceptance test has passed, if it fails again that means a regression (the existing code has been broken)

SWTBot **

2.3 Regression Testing

2.4 Integration Testing

Since the project depends a number of external service providers (in particular Betfair) integration tests are vital. This has been achieved by...

2.5 Continuous Integration

Since the beginning of iteration 5 a continuous integration has been installed on a virtual machine provided by the Computing Support Group, which emulats a dual core, 1GB, 64bit machine. We have decided to use Hudson, which comes with a plugin for running UI tests. These are different than normal tests since they require a display and cannot run on a headless server, unless some in memory display mechanism is provided. There are two alternatives: xvfb and hudson's plugin. Since we initially assumed xvfb would be harder to setup Hudson provides easy integration with git. The CI server is set to poll the repository and, when changes are detected, checks out a fresh copy and runs our normal build script.

2.6 Measuring Test Coverage

Since we did not adopt a TDD approach from the very beginning, it was important to obtain an overview of the parts of the code that need to be tested. Starting with iteration 5 we have used Cobertura, an open source tool which provides neat test coverage reports for Java programs.

To facilitate report generation, a single ant task has been set up to compile the code with the debug info (vital for Cobertura to indicate line numbers and measure coverage), run all the tests (unit, acceptance, integration) and generate

human readable reports which provide an indication of the current test coverage as well as branch coverage and complexity measures.

This data enabled us to identify lines which were not touched by tests. This is usually fixed by writting another test to cover the specific path, but it can also be the case that the functionality is actually never required, in which case it is completely removed. Again this illustrates how tests have been used to ensure a neat design/code etc.

Branch coverage indicates when tests do not cover particular cases and has proven useful with regards to particularly tricky conditionals.

Cobertura also generates cyclomatic complexity values for each class/package, measuring the number of independent paths through the control flow graph of the code. Since it has been shown that high values are usually an indication of error prone code[2] this measurement is used to check code sanity.

2.7 Code Sanity Measures

2.8 Logging

2.9 Test Guided Design

Refactoring

Used extract superclass, pull up (for tests), extract class

2.10 At which development stages did you use tests

2.11 Which portions of your code base were tested in this manner

2.12 what bugs did testing reveal

3 General Validation

General Validation: to decribe any other methods you may have used to validate your executable deliverable; the following list is only suggestive but should give you an idea of what things you could do: Did you validate your user interface design? Did you use lint or similar tools? Did you conduct a manual code inspection (not by the person who wrote the code)?

Yes, as explained above we have test the UI through our acceptance tests. Also visual inspection by team members was used to ensure no layouts are messed up. We used PMD and Cobertura report to check code "sanity". We also used a review mechanism in which, normally, code has been inspected by at least two team members.

Did you exert your software to stress testing?

No - should do.

Did you test the GUIs of your software?

yeees

4 Managerial Documentation

to give a formal account of group management and group activities: Collaboration tools used (eg svn)

git, github Management policies (eg code change/validation policies) Management of knowledge transfer within the group

dropbox, google docs A table of the group meetings - including dates, format and which members attended A table of the hours spent per week on which tasks or activities by each member on the project The Log-Book

Evaluation Criteria for Report Three: the same as for Reports One and Two.

Rferences

The cyclomatic complexity of a section of source code is the count of the number of linearly independent pathsthrough the source code. For instance, if the source code contained no decision points such as IF statements or FOR loops, the complexity would be 1, since there is only a single path through the code. If the code had a single IF statement containing a single condition there would be two paths through the code, one path where the IF statement is evaluated as TRUE and one path where the IF statement is evaluated as FALSE. Mathematically, the cyclomatic complexity of a structured program[note 1] is defined with reference to the control flow graph of the program, a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second. The complexity M is then defined as:[2] M = E N + 2P where E = the number of edges of the graph N = the number of nodes of the graph P = the number of connected components

References

[1] Steeve Freeman, Nat Pryce, *Growing Object-Oriented Software, Guided by Tests*, Addison-Wesley 2010

[2] Robert Chatley's course on Software Engineering Methods `http://en.wikipedia.org/wiki/Cyclomatic_complexity`, Imperial College London

[3] Jonathan Rasmusson, *The Agile Samurai*, Pragmatic Bookshelf, October 2010.

[4] Jeff Langr and Tim Ottinger, *Agile in a Flash*, Pragmatic Bookshelf, January 2011.

[5] Robert Chatley's course on Software Engineering Methods `http://www.doc.ic.ac.uk/~rbc/302/`, Imperial College London