

# Retrofitting an Acceptance Test Framework for Clarity

Rick Mugridge and Ewan Tempero  
Department of Computer Science,  
University of Auckland,  
New Zealand  
{r.mugridge, e.tempero}@auckland.ac.nz

## Abstract

*An XP customer needs to write and check acceptance tests. However, the format for defining the tests needs to be clear. Many acceptance test approaches use arcane formats which do not promote clarity for the customer, due to a conflict of interest between the complexities of automation and the needs of the customer. We discuss the evolution of acceptance tests to improve their clarity for the customer.*

*Sat is an acceptance test system for testing socket-based servers with multiple clients. The first version used an XML file to define the tests in a test suite. Any errors detected were written to a text log. There were two problems with this first version. The XML format made it difficult to read and edit the tests. When an error was given, it was not easy to identify the place in the test where the problem occurred.*

*Sat was altered to make use of Fit, a testing framework that uses HTML tables for defining tests and reporting any errors. We found the new version considerably easier to use. The tabular form makes it much simpler to read and alter the tests. Any errors are reported in a copy of the tables, in the place where they occur. We have also found it convenient to include information about the tests in the HTML, providing a form of "literate testing".*

## 1. Introduction

The practices of XP bring testing to focus early in software development, rather than catching problems well after they have occurred [4]. Tests play a significant role in development, with acceptance (or customer) tests helping to drive story development and with programmer tests driving design. Much of what has been learned about practical testing in traditional software development approaches [2] also apply in XP, with testers providing a cogent perspective [12].

In XP, acceptance tests are ideally written by the customer with help from testers. Such test may need to evolve as the needs of the system are better understood

[8]. Hence it is essential to use a format that is both suitable for automatic execution and which a customer can easily understand. A syntactical format that is well suited to automatic execution may take a lot of effort to learn, more than many customers can afford. Descriptions of the tests also need to be easy to read and understand, so that they can be verified as being correct and complete [2]. In this paper, we discuss the development of clarity in acceptance tests, and in particular explore the use of the Fit system for specifying them [5].

*Sat* (Socket Acceptance Tester) is an acceptance (customer) test system for testing socket-based servers with multiple clients. We initially developed *Sat* to define customer tests for a Chat server that was developed in a student XP project in 2002.

The first version of *Sat* used an XML file to define the tests in a test suite, where each test consisted of a sequence of messages being sent and expected to be received by one or more clients. Any errors detected, such as when an unexpected message was received by a client, were written to a text log.

We found several problems with the first version. The XML format made it difficult to read (and edit) the message sequence in a test and to keep track of the expected state of the server after each message was sent to it. When an error was given, it wasn't easy to identify the place in the test where the problem occurred.

At this time, we became aware of the Fit framework [5]. It was immediately obvious that using the table format of Fit for both defining the tests and for reporting would be superior to the XML and the logs that we had been using. The Fit table format would also allow a customer to write the tests with a simple HTML editor.

We altered *Sat* to make use of Fit. The customer found the new version much easier to use. The tabular form of the tests, with a table column for each client, make it much easier to read, create and edit customer tests. Any errors are signalled in a copy of the HTML, in the tables where they occur. This makes it much easier to understand what test has failed and with what symptoms. It is convenient to add information about the tests outside the tables, providing a form of "literate testing" akin to literate programming [9].

We follow with an introduction to the Chat server. Section 3 introduces the first version of Sat, shows example tests for the chat server, and discusses its limitations. Section 4 introduces the Fit framework. Section 5 discusses the second version of Sat, along with revised tests. Section 6 discusses related work and Section 7 concludes.

## 2. Chat Server

Sat was developed in order to write acceptance tests for an XP project. The project involved 70 second year Software Engineering students who worked in pairs to write a Chat system.

This system would allow users to "chat" with each other across the Internet. Users could join "rooms" (including multiple rooms simultaneously), and would hear everything "spoken" in whatever rooms they were in, and could speak to other occupants of the rooms. They could also make simple queries to determine such information as who the other occupants of a room are, and what rooms are available.

The first few iterations of the project only involved the Chat server, so tests were only required for the messages that were to be sent and received by the server itself. The message formats were defined by the project customer; they are encoded in XML. For example, to create a new chat room with a given name, the following XML message is sent to the server by a client:

```
<NewRoom>MyRoom</NewRoom>
```

Consider that once this client has entered the new chat room, a second client has connected. On sending the message <DisplayUsers/>, the following message is received by the first client (with no white space):

```
<UserList>
  <User>
    <PhoneNumber>1111</PhoneNumber>
    <ChatRoomList>
      <ChatRoom/>
      <ChatRoom>MyRoom</ChatRoom>
    </ChatRoomList>
  </User>
  <User>
    <PhoneNumber>33333</PhoneNumber>
    <ChatRoomList>
      <ChatRoom/>
    </ChatRoomList>
  </User>
</UserList>
```

## 3. First SAT

Sat was designed for testing socket-based servers. It defines an interface that is implemented for testing a particular server, by defining methods to start up and shut down the server, and to provide details for Sat to connect clients to the server once it has started for each test.

In the first version of Sat, a test suite is defined in XML as a set of tests. Initially the tests were organised according to the clients that would communicate with the server.

For example, the following test involves a single client, which sends three messages to the server at the specified times and expects to receive a single message at time 2:

```
<SocketAcceptanceTest
name="SingleUserMessage">
  <Client>
    <ToServer>
      <Msg time="0">
        <Connect>0</Connect>
      </Msg>
      <Msg time="1">
        <Message>Hi</Message>
      </Msg>
      <Msg time="10">
        <Shutdown/>
      </Msg>
    </ToServer>
    <FromServer>
      <Msg time="2">
        <Message sender="0" room="">
          Hi
        </Message>
      </Msg>
    </FromServer>
  </Client>
</SocketAcceptanceTest>
```

There are two levels of XML here. The first element, <SocketAcceptanceTest>, defines a test within a test suite and contains one or more clients, with messages that are to be sent and expected to be received by that client.

Within the message elements are the messages themselves, which happen to also be encoded in XML. For example, the first message is a Connect message to be sent from the first (and only) client, identified by the id "0".

This organisation made it easy for the Sat system to create client connections to the server, but it was awkward from the customer's point of view. It was necessary for the customer to reorganise the messages into a time sequence in order to check the test and it was easy to make mistakes. To avoid this, we changed the structure of the tests to better reflect the needs of the customer, basing them on the time ordering.

In this new organisation, each test consists of a sequence of messages sent or received by one or more clients. For example, Fig. 1 shows a test suite for an early story of the project, where a single client connects, sends a message, and expects to receive a reply.

A `<ToMsg>` is a message that is to be sent to the server, while a `<FromMsg>` is expected to be received. The messages are processed in the sequence that they appear in the test. Time delays are inserted between them to ensure that the tests are deterministic when more than one client is sending messages to the server.

```
<SocketAcceptanceTest
name="SingleUserMessage">
  <ToMsg client="c1">
    <Connect>0</Connect>
  </ToMsg>
  <ToMsg client="c1">
    <Message>Hi</Message>
  </ToMsg>
  <FromMsg client="c1">
    <Message sender="0" room="">Hi</Message>
  </FromMsg>
  <ToMsg client="c1">
    <Shutdown/>
  </ToMsg>
</SocketAcceptanceTest>
```

Figure 1 Time-ordered organisation of tests

This version of Sat reads the XML through a DOM and creates objects that represent the test suite and its tests, each with a sequence of messages. When Sat is run with a particular server, it runs through each of the tests in turn. For each test, it creates a new version of the server. For each client, it opens a new socket connection to the server.

For each message for a particular client within a test, it either sends a message after a suitable delay or expects to receive a message within a timeout period. If it fails to receive the correct message, or it times out, it writes an error message to the log, identifying the test by the name in the XML.

Any white space is removed from a message before sending it and before comparing a received message to what was expected. A simple equality check is made on the XML, allowing for the attributes being unordered and allowing for case changes in the tags. As the XML messages tend to not be very long, the expected and (incorrectly) received message are both written to the log. If the messages were longer, it would be convenient if a *diff* of the two XML documents was provided instead.

About 80 acceptance tests have been written for the Chat server, making up over 4000 lines of formatted XML.

Some of these tests are rather complex. For example, one story allows a privileged user to close a chat room, which means that no new users may enter and it will be deleted once all users in the room have left. Some of the acceptance tests for this story result in over 30 messages being sent and received by two clients.

We needed to be confident that the acceptance tests were correct before we supplied them to the student project pairs. Because of the complexity of the tests, the only realistic way to check the tests was to build our own version of the Chat server. In doing this, we found many minor errors in the tests. For example, the customer often got confused about the state of the server and so the test had incorrect expectations for messages received by the clients.

In trying to work out whether the tests or the server was at fault, we found that the XML format was not convenient to read. Errors in the log are not easy to relate back to the XML input, in order to determine the fault in the server (or in the test itself). The tests were not straightforward to alter. We could have improved the reporting by merging the errors into the XML and displaying the results mapped into HTML.

We became aware of Fit at this point. It was immediately obvious that using the HTML table format of Fit for both defining the tests and for reporting would be superior.

## 4. Fit

Ward Cunningham developed Fit, a publically-available Java framework for creating customer tests [5]. Versions for other languages are becoming available. A test is defined within an HTML table; there may be several such tables in a page. The first row of a table specifies the name of the class (called the fixture) that interprets the tests contained in the rest of the table.

Fit comes with several general fixture classes for running tests. In some cases, a class can be used directly; in others the fixture class is subclassed to provide the desired test behaviour. For example, *eg.Calculator* is a subclass of *fit.ColumnFixture*; this tests the various functions of an HP-35 calculator.

Fit has several advantages over other general approaches, due to the use of tables and the ease with which new sorts of tests can be added to the framework through defining new fixtures. HTML tables are used to define the tests, so a simple HTML editor can be used to create a web page containing tests and a browser can be used to view them and the reports. However, Fit may not be so well suited to tests which have a complex structure that don't fit well into the table format (or a sequence of tables).

When the tests are executed, extra information is added to a copy of the HTML. The table cell of each test that passes is coloured green. Each test that fails is coloured red, and information about the error is provided within the cell. Examples are provided in the next section.

It is straightforward to extend the fixtures that are supplied with Fit, as well as to make up new types of fixtures, as we did for the second version of Sat. Only 260 lines of Java code were needed to define a new fixture class for Sat.

## 5. Second Sat

The second version of Sat used Fit. This meant discarding the code from the first version that gathered test suite information from a DOM and discarding the logging process. Sat was substantially refactored, as the XML input and logger output had a strong influence on its architecture. Sat was then reengineered to utilise Fit and the tests were programmatically translated into the required HTML format.

This format displays the tests in a two-dimensional structure, with time downwards and clients across. This layout makes it much easier to read and check the tests.

We wrote a new general type of Fit fixture for Sat, an *ActionRowFixture*. This interprets the first cell of each row as the name of a method, which is called reflectively. The class *sat.fit.SatXMLFixture*, a subclass of this fixture, defines appropriate methods that are called from the *ActionRowFixture* when the test executes. In the case of Sat, there are several methods provided, including ones to send messages, to receive messages and to delay. The send and receive messages act on any messages found in the columns corresponding to each of the clients.

For example, Fig 2 shows part of a web page with a table for testing that the *DisplayUsers* message gives the correct response when two users have connected and one has entered a new chat room. The tests in this web page file are run by passing the filename of the HTML page to the Fit file runner, along with the name of the HTML file that will be produced as a report.

The first row of the table in Fig. 2 includes the name of the fixture class *sat.fit.SatXmlFixture*. The Fit file runner reads each of the tables in the page in turn. For each table, it creates an object of the fixture class named in the first row and passes control to that object. For Sat, the fixture class accesses each row in turn (ie, time is downwards). It reflectively calls the method of the fixture object named in the first column, passing to it a reference to the cells in the rest of the row.

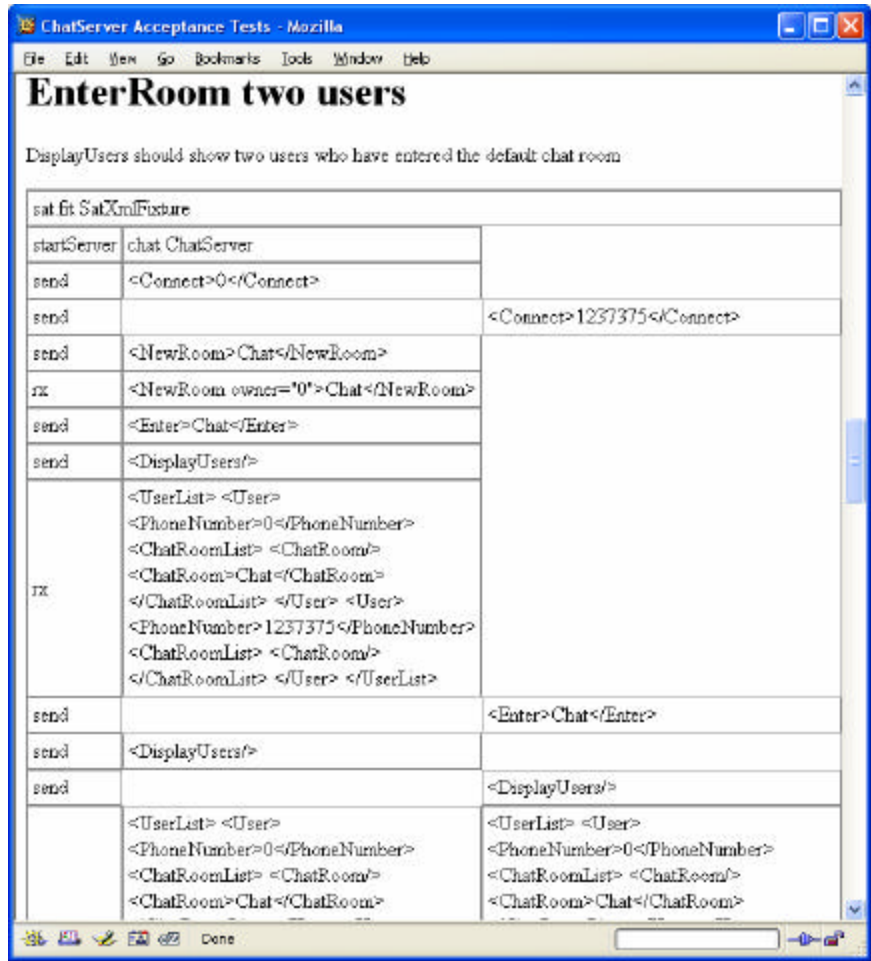


Figure 2. Page showing part of a test

The second row of the table results in the *startServer* method being called. This passes control through a standard interface to server-specific code that is responsible for starting a new copy of the server (as well as returning details for accessing the server). The method *stopServer* (not shown) is responsible for any teardown (as well as checking that there are no extra unexpected messages received by any of the clients).

When the *send* method is called, it checks the contents of each of the subsequent cells; if a cell is non-empty, it takes the contained XML, removes any HTML tags, removes text that is only white space from the XML and sends it through the appropriate socket to the server. On the first communication involving a client, a socket connection is made for that client through to the server.

When the *rx* method is called, it also checks the contents of each of the subsequent cells. If a cell is non-empty, it takes the contained XML, removes any HTML

tags, removes white space from the XML and compares it to the next message received from that client. A timeout on receiving a message results in an error, as does receiving an incorrect message. For example, Fig. 3 shows the report that results from the test shown in Fig. 2 when unexpected messages are received.

Several other methods are provided. The *delay* method specifies a delay time, to ensure that message from two clients are received and processed in the correct order. The method *rxTimesOut* tests that no message is received. The *comment* method does nothing, but allows arbitrary comments to be added in the middle of a test, such as to record the current state (eg, the rooms that a client is currently in).

As shown by Fig. 2, Fit provides for a form of "literate testing", akin to literate programming [9]. Arbitrary (non-table) HTML may be placed between the tables to document details of tests and stories.

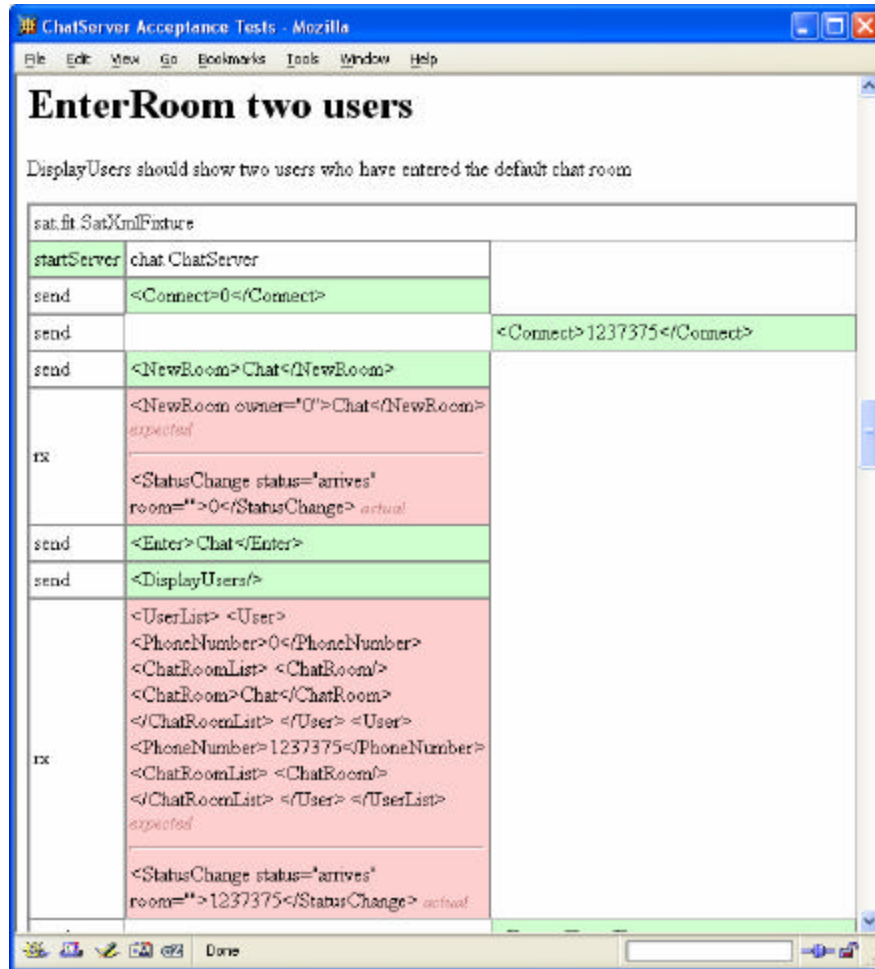


Figure 3. Web page reporting test failure

When the fixture for Sat was being developed, several other fixture methods were included. These were used to (partially) test that the fixture code and XML processing worked correctly against a known server (which simply echoed received messages back). For example, *rxDiffers* only fails if the received message is the same as the (un)expected one; this is used to check that various minor variants in a XML message are detected correctly as errors.

## 6. Related Work

Buwalda et al [2] describe a table-based approach to testing which organises both automated and manual tests in a similar way to Fit, and which uses action words to define test steps. Spreadsheets are often used by testers to organize tests [2, 4].

There are several general-purpose acceptance testing frameworks and tools that allow for tests to be defined in

XML. In each case, the XML tends to be complex and thus is likely to be awkward for a customer to use.

*Accept* can be tailored to different sorts of testing [10]. In general terms, it is similar to Fit, but differs in the details of how the tests are managed and the results are reported. The tests are defined in terms of strategies, actions and properties, which are tailored to a particular application by writing Java code (analogous to the fixture code that can be added to Fit). Tests are created using an interactive application, generated as XML and run against an application. Failing tests are shown in HTML, having been mapped from the XML with errors attached. Fit provides the same generality, but is likely to be more convenient for a customer, as tests can be edited easily in HTML and viewed in the same form.

In *Avignon*, acceptance tests are also written in XML [1]. The XML tag of a test is mapped to a Java class which carries out the actions of the test. This was developed for web testing but could be applied more

generally. *Puffin* [14] is similar to Avignon, but also allows the sequencing of actions into transactions.

XML is also used to define tests in various specialised testing tools, and clarity is an issue there too. For example, *Abbot* is a system for acceptance testing of Java Swing GUI systems which provides a much more convenient way of specifying tests than programming them through a robot [13]. However, the XML is still complex because of the selection of GUI components in the user interface in order to check and manipulate them.

*HttpUnit* is a popular system for automated web site testing [6]. It provides a Java API for accessing web sites without a browser. As tests are created programmatically it is not suitable for customers. It is also a fairly low-level API, so that it is difficult to create and check tests.

There have been several efforts to provide a higher-level mechanism than *HttpUnit* for web-site testing. For example, the *jWebUnit* API describes the tests in terms of navigating a web application (following links or submitting forms) and assertions about what should result from these actions [7]. *Canoo WebTest* provides a similar level of abstraction, but uses XML to specify the tests [3].

*Isis* is a system for testing web-based systems that takes a different approach to defining tests [11]. The tests are defined as sample pages, which include information about links to be followed and forms to be submitted (with the corresponding data). This is convenient for a customer to read, because it is in almost the same form as web pages that a server will provide. However, it has two problems: some of the detail of the sample pages may not be essential to the tests and it is inconvenient to make changes due to the likely redundancy between sample pages.

## 7. Conclusions

The format of defined acceptance tests needs to be straightforward for a customer to create, check and alter the tests [2]. While XML is sufficient from the perspective of test automation, it is less than ideal for customers to use. The XML can be translated into a more convenient form for viewing and checking, but this does not help with changing the tests. We found a tabular structure to be superior for our tests, although this won't always be appropriate.

Acceptance tests were needed for testing a Chat server which allows multiple clients to connect by sockets for communication. Having created Sat, a general system for acceptance testing of a socket-based server, we found that the tests and error logs were not easy to use. This was because of the XML format and because of having to relate errors written in a textual log back to the tests in the XML.

Retrofitting Sat to use the Fit framework was straightforward and has resulted in a much-improved system for all concerned. Being able to lay out the messages sent and received in a table, with a column for each client, has made it much easier to write and understand a sequence of communications of the clients with the server. It also makes it much easier to understand an error shown in a resulting report, as the error message is placed directly in the context of the failing test.

## Acknowledgements

Many thanks to Brian Marick for his helpful suggestions.

## References

- [1] Avignon, <http://www.nolacom.com/avignon/>
- [2] H. Budwalda, D. Janssen and I. Pinkster, *Integrated Test Design and Automation*, Addison Wesley, 2002.
- [3] Canoo Webtest, <http://webtest.canoo.com/>
- [4] L. Crispin and T. House. *Testing Extreme Programming*, Addison Wesley 2003.
- [5] W. Cunningham, "Fit: Framework for Integrated Test", <http://fit.c2.com/>
- [6] R. Gold, "HttpUnit", <http://httpunit.sourceforge.net/>
- [7] jWebUnit, <http://jwebunit.sourceforge.net/>
- [8] K. Beck. *eXtreme Programming Explained*, Addison Wesley, 2000.
- [9] Knuth D.E. "Literate Programming", *The Computer Journal* **27**, 2, 1984, pg. 97-111.
- [10] R. W. Miller and C. T. Collins. "Acceptance Testing", *Procs. XPUniverse*, July 2001.
- [11] R. Mugridge, B. MacDonald and P. Roop. "A Customer Test Generator for Web-Based Systems", *procs. XP2003*, Genova, Italy, Springer-Verlag 2003.
- [12] B. Petticord, "Testers and Developers Think Differently", *Software Testing and Quality Engineering*, Jan/Feb 2000, [www.stqemagazine.com](http://www.stqemagazine.com)
- [13] T. Wall, "Abbot: Java Swing Test Framework", <http://abbot.sourceforge.net/>
- [14] K. Weissinger, Puffin, <http://www.puffinhome.org>

