JaTack – Java Acceptance testing Tool ACKronym*

Anders Nilsson
Dept. of Computer Science, Lund University
Box 118
S-221 00 SWEDEN
Tel: +46 46 222 01 85
Fax: +46 46 13 10 21
email: `andersn@cs.lth.se`

Abstract

In XP, automatic testing of the produced software takes a central role in the development process. For unit tests, there are a multitude of freely available testing frameworks for many programming languages while there seems to be a complete lack of available testing frameworks for automating acceptance tests. At the department of Computer Science, Lund University, we felt the need for a light-weight acceptance testing framework that could be used in both students programming projects courses and in our own research projects.

This paper discusses the thoughts behind, and the implementation of, JaTack, a very lightweight framework for automating acceptance tests for batch-oriented programs.

1 Introduction

Extreme Programming (XP) has received a lot of interest from the software engineering community since it's main creator, Kent Beck, presented a paper in 1998 [Bec99]. But, what is maybe more important, XP has received massive interest from the grass-root level of software engineering, software developers en masse fall in love with the principles described by XP.

One of the absolutely most important principles of XP is automated testing of code. Having full suites of unit tests and acceptance tests just one mouse-click or keyboard-stroke away gives confidence to the developers to perform those needed, but seldom implemented, refactorings of the source code that would otherwise degenerate into the well-know hell of patched spaghetti code. Full automated suites of tests also increases the chance of being able to deliver high-quality (almost) bug-free software to the customer. The practice of shipping mediocre software and the wait for the customers to report bugs has no place in XP[1].

*Misspelling on purpose.

[1]Even though commonly practiced by a very large US software company.

In XP there are two kinds of tests that the software should pass, unit tests and acceptance tests.

1.1 Unit tests

A unit test tests one specific unit of code, most often a method or function. The developer should always write the unit test before implementing the tested method. It should always be possible to run all, or a subset of, the unit tests with a single mouse click or keyboard stroke. To do that, testing frameworks are needed that can run all tests and present the result, pass/fail, in a visual way. One of the first, and most well known, unit testing frameworks is JUnit [jun] for automatic unit testing in Java, but since then, many similar unit testing frameworks for many different programming languages have appeared. Many of these are, like JUnit, freely available from `http://www.junit.org`.

1.2 Acceptance tests

Acceptance tests are in XP the customer's way of making sure he gets what he wants. The customer, maybe in cooperation with a developer, should write acceptance tests that verifies user stories; "If we input this and this, we will get that and that as output".

There are usually much fewer acceptance tests than unit tests, but, as the number of implemented user stories grow, so does the need for automating also the acceptance tests. However, automating acceptance tests tend to be more difficult than automating unit tests. An automated acceptance test framework is not only affected by the programming language used, but by the particular user interface, maybe the windowing system, and also the operating system. The fact that most commercial application are very GUI-centric also limitates the possibilities of creating an easy-to-use testing framework.

2 Automating acceptance tests

For the discussion about automating acceptance tests, we may want to divide the to-be-tested applications into two classes; *batch* applications and *interactive* applications[2]. These two classes of applications are fundamentally different with regard to interaction with the user or a testing framework.

Batch

User interaction with applications of the *batch* class basically stops when the user hits the `Enter` key, or in other words, the application's behavior is completely determined by its input arguments. Therefore, it is generally easy to develop automatic acceptance testing frameworks for the *batch* class of applications.

There are numerous ways of implementing a testing framework for batch applications, as found in, for example, Jeffries et.al. [JAH01]. The most intuitive implementation would be to write scripts in some favorite scripting language

[2]One could argue that multi-threaded and/or distributed applications belong to a third class, though it is not orthogonal to these two mentioned. However, this lies beyond the scope of this paper.

that, in turn, runs the application with different input parameters and compares the generated output to the expected one. Some obvious disadvantages are that it takes some familiarity with the particular scripting language used to be able to add or change the test cases. It is also difficult to run just one particular test case, or a selected suite of test cases. A little bit more elaborate techniques are to specify acceptance tests in a spreadsheet, or develop a small dedicated scripting language. That way, the sharp edges in using general purpose scripting languages can be rounded.

Interactive

Interactive applications prove fundamentally more difficult to test than batch applications, just because of their interactive behavior. Where a test case for a batch application can be performed by typing

> `foo some arguments | diff expected` *<Enter>*

could performing a similar test case in the interactive version of the application end up in something like:

> `foo` *<Enter>*
> Choose `File->Load` from the menu.
> Type the name of the input file.
> Press `OK`. Press the button *Do some processing and output.* Compare the screen output with the expected output.

Clearly, the latter test case seems very hard to reproduce in an automated test framework. But, here we can make a distinction between applications that are developed from scratch, XP style, with automated tests in mind, and projects where the XP methodology and automated tests are introduced during the application's life-cycle. In the latter case the situation seems rather hopeless unless using one of these GUI record/playback tools, see for example [Win], which can record the user interaction, as described in the second example above, at the GUI level, and then make a playback in an automated test case. How attractive these tools may seem, they do have a big drawback in that they work at the visual GUI level. If the looks of the GUI, but not necessarily the functionality, is slightly changed, one may have to re-record all test cases once again. However, when alternative methods are scarce, automated acceptance tests are surely better than manual tests, and the cost of once in a while re-recording the tests may be justified. On the other hand if we, when developing a new interactive application from scratch, have our thoughts on how to make automated tests possible, we may come up with better solutions than using playback tools.

To be able at all to even unit test interactive applications, we have to separate the GUI from the logic, i.e. all GUI event handlers must contain no application logic but a call to the non-GUI part of the application. This way we may write unit tests to test the complete application logic, and can rely on fairly simple manual tests to ensure the looks of the GUI. Since we know that no application logic is inside any of the GUI event handlers, it is fairly easy to write another front-end to the application, a batch front-end. This batch front-end can define the application behavior by its input arguments, or it can read and parse a file with a dedicated scripting language that corresponds to user GUI actions. It does not really matter since both alternatives enables us to test also interactive applications within an automated acceptance test framework.

2.1 Acceptance test specifications

Specifying test cases in an automated acceptance testing framework can be done in numerous ways, with different degrees of abstraction. The abstraction levels range from writing test cases in the programming language used for the application in the same way unit tests are usually implemented, via using different scripting languages — general purpose, or specific — to specifying test cases in spread sheets or web applications. When choosing test case specification strategy for a testing framework implementation, some questions need to be answered:

1. Who will write the test case specifications and which technical capabilities do they possess?
2. Using this abstraction level, how hard/easy will it be to specify typical test cases for the kind of applications that will be tested?
3. How hard will it be to implement the testing framework using the chosen abstraction level?

Who will write test specifications?

One of the the many differences between XP and most other software development processes, is who will write tests. Unit tests are an integral part of coding and are written by the programmers themselves during coding. Unit tests test the internal behavior of the application and are written by people with good knowledge both in the application architecture and in the programming language(s) used. The most feasible alternative obviously is to write unit tests in the same programming language(s) as is used for the application itself.

Acceptance tests, especially as used in XP, are different creatures. Acceptance tests is the way the customer ensures she get what she wants in the way she wants it. This means that acceptance tests are written by people with very varying programming knowledge, and who have almost no knowledge of the application internals. Using the xUnit frameworks for both unit tests and acceptance tests may be a good idea, but only if one know for sure that the customer is capable of using this programmatic mean of expressing test cases. In all other cases the level of abstraction need to be increased to a level that can be handled by the customer. Specifying test cases on a programmatic level may also cause some frustration among the people writing the tests because of all those nitty-gritty programming details that do not add any value to the test case, but nevertheless are mandatory as a result of the low abstraction level.

Test specification complexity

Naturally, the complexity of acceptance tests can vary dramatically, from a simple input resulting in a yes/no output to cases where a large set of input results in a large and complex output. While this situation may be handled having the full power of a programming language, increasing the abstraction level one notch too much may make it impossible to handle complex test cases.

Implementation complexity?

Using an already existing unit testing framework requires the least amount of implementation while implementing a framework based on spread sheet specifications may require a lot more work, depending on the expressive power needed in the specifications. Basing the testing framework on a scripting language parser supposedly falls in between the extremes.

2.2 Acceptance test framework

For a testing framework to be useful in an XP environment, it should be lightweight and it should be easy to add or change test cases. Given the previous discussion on testing interactive applications we conclude that the least common divisor for acceptance testing applications, is a batch application.

The discussion in section 2.1 concludes that to make it possible for others than the programmers to write test cases, we can not use existing unit testing frameworks, but must raise the level of abstraction so that also people with little or no knowledge in programming (customers) are capable of writing acceptance tests.

3 JaTack

JaTack is a lightweight automated acceptance testing tool being developed at the Department of Computer Science in Lund. There is a need for such a lightweight testing framework both for student project courses and for automatic testing of software written as parts of research projects.

3.1 User stories

To get a grip on what requirements this tool should fulfill, a couple of user stories were written down to describe what was thought to be the needed functionality:

1. **Text based input/output.** The tool should support automated testing of batch programs with text based input and output. A typical example is a compiler taking a source code file as input, and produces error messages as output. Or, a sort program taking an unsorted list of text lines as input producing a sorted list as output.

2. **Test suite.** The tool should support several input/output file pairs in a directory, and that all test cases can be run in a suite. It should be possible to easily add new test cases (input/output file pairs).

3. **Generate output for a new test case.** For each new test case, it should be possible to write a new input file, run the program, inspect the output and, if it is OK, add the input/output file pair as a new test case in a test suite.

4. **Update output.** Due to changed behavior of the program the output file will need to be updated. It should then be easy to run the program on the given input file, inspect the diff's between the old expected output and the actual output, and to update the expected output file to reflect the changed functionality of the program.

5. **Several test suites.** There should be a possibility to have several test suites organized in several subdirectories.

6. **Several programs.** It should be easy to test several programs, typically on different test suites. For example, one would like to run a compiler with different options on different test case suites.

7. **Selective testing.** It should be easy to choose which test cases that should be run. For example, all test suites, one particular test suite, or one particular test case in one test suite. Preferably in some kind of interactive tool.

3.2 Implementation

When the JaTack program executes, it looks in the current directory and all subdirectories for test specification files that have the suffix `.jatack`. Alternatively, one can supply a specification file as an argument in which case JaTack will not search for more files. When all specification files have been found, they are in turn parsed returning a list of test cases. Each test case is executed and a diff between actual and expected output is shown.

The tool is built as a platform independent Java jar archive, and can test applications written in almost any programming language. Note that the current implementation makes use of the standard Unix utility `diff` which is not by default found on other platforms. It is though easy to change the tool to use any available file comparing utility.

3.2.1 Parser

A parser for the grammar shown in figure 3 was built using the JavaCC [Met] parser generator together with JastAdd [HE01], a tool which is being developed at the department. JastAdd is a tool that allows one to create object-oriented JavaCC ASTs and then use aspect weaving or reference attribute grammars (RAG) to supply functionality to this AST in a clean way. To create an object-oriented AST with these tools at least two input files are needed, plus the wanted aspects. A context-free grammar, see Figure 1, and the aspects are fed to JastAdd which will create a class hierarchy of AST nodes with methods as specified in the aspects. A concrete grammar, see Figure 2, is then fed to JavaCC[3] which builds a parser that will construct an AST with previously mentioned node types.

In JaTack, JastAdd was used to weave in a small aspect for assembling a list of test cases which is returned to the front-end of the application.

3.2.2 Specification files

The format of the specification files is described by the BNF in figure 3. The reasons behind this specific design are that the scripting language should be minimal with as few grammatical rules as possible (easier to remember how to write test cases), while being as easy to read as possible. Comments are denoted with `//` for one-liners or `/* ... */` for multi-line comments.

[3]The jjtree preprocessor of JavaCC is also used for supplying the AST building functionality.

```
abstract Any;
Start : Any ::= TestSpecificationList;
TestSpecificationList : Any ::= TestSpecification*;
TestSpecification : Any ::= OptDescription Command InputOutputList;
Command : Any ::= IdentifierList;
OptIdentifierList : Any ::= [IdentifierList];
IdentifierList : Any ::= Identifier*;
Identifier : Any ::= <ID>;
InputOutputList : Any ::= InputOutput*;
InputOutput : Any ::= OptDescription Input OptOutFile Expected;
OptDescription : Any ::= [Description];
Description : Any ::= IdentifierList;
Input : Any ::= OptIdentifierList;
OptOutFile : Any ::= [OutFile];
OutFile : Any ::= Identifier;
Expected : Any ::= Identifier;
```

Figure 1: The context-free grammar of the JaTack parser.

An example of a specification file is shown in figure 4. This example shows a few test cases with and without input arguments, and with and without file output.

3.2.3 Front-end

The front-end of JaTack today is strictly batch-oriented. Depending on the given arguments, find JaTack specification files, instantiate the parser to get a list of test cases, execute test cases and print the diff between actual and expected output on stdout.

However, the separation of front-end from the specification file parser makes it easy to write a new interactive front-end, which would be nice to have for running specific test cases, or adding and updating the test cases.

3.3 Implementation results

User stories 1,2,5,6 are currently fully implemented in the JaTack tool. The remaining user stories will be implemented in a future interactive interface to JaTack. The tool has been used during spring 2002 in a students project course at Lund University, see Figures 5 and 6 for a test specification file and results taken from one of the course projects. The feedback returned from the students was positive, they found it much easier to specify test cases with JaTack than running manual test cases. Unfortunately were there no possibilities for comparing our own testing framework with any other available acceptance testing frameworks, which would otherwise have been very interesting.

4 Related work

Although acceptance tests, or functional tests as they are known as, are very often described in the software engineering literature, it is not easy to find acceptance testing framework implementations. Crispin [Cri] has written a paper where she talks about automating acceptance tests in an XP environment, and also briefly describes an implementation of a testing framework which sadly has not been made available for testing. However, from the brief description, this tool seems to be very GUI-centric which may very well be appropriate

```
ASTStart Start() #Start : {}
{   TestSpecificationList()
    { return (ASTStart) jjtThis; }}

void TestSpecificationList() #TestSpecificationList : {}
{   ( TestSpecification() )+}

void TestSpecification() #TestSpecification : {}
{    <TESTDESCR> OptDescription()
    <COMMAND>  Command() InputOutputList() }

void OptDescription() #OptDescription : {}
{   [Description()]}

void Description() #Description : {}
{   IdentifierList()}

void Command() #Command : {}
{   IdentifierList()}

void OptIdentifierList() #OptIdentifierList : {}
{   [IdentifierList()]  }

void IdentifierList() #IdentifierList : {}
{   (Identifier())+}

void Identifier() #Identifier : {Token t;}
{   t = <ID>
    { jjtThis.setID(t.image);}}

void InputOutputList() #InputOutputList :{}
{   (  InputOutput() )+}

void InputOutput() #InputOutput : {}
{   <IO> OptDescription() <INPUT> Input() OptOutFile()
    <EXPECTED> Expected()}

void Input() #Input : {}
{   OptIdentifierList()}

void OptOutFile() #OptOutFile : {}
{   [<OUTFILE> OutFile()]}

void OutFile() #OutFile : {}
{   Identifier()}

void Expected() #Expected : {}
{   Identifier()}
```

Figure 2: The JavaCC input file for the JaTack grammar.

for many commercial software systems but does not suit our needs. Miller and Collins [MC] also describes a framework for running automatic acceptance tests. This framework was also not available for testing.

5 Conclusions and future work

Automated acceptance testing is, in the general case, not very easy to accomplish. This is especially true having in mind that most commercial, and many other, applications these days are highly interactive and have a GUI. However, since we have to separate the GUI from the application logic to be able to run automatic unit tests, which are crucial in XP, the overhead cost of adding also a batch front-end to the application will not be too dramatic. With a command line front-end, either script based or argument based, it is much easier to

```
TestSpecificationList:
    TestSpecification*
TestSpecification:
    TESTDESCR: OptDescription COMMAND: InputOutputList
OptDescription:
    [Description]
Description:
    [IdentifierList]
IdentifierList:
    Identifier*
Identifier:
    <ID>
InputOutputList:
    InputOutput*
InputOutPut:
    IO: OptDescription Input OptOutFile Expected
Input:
    INPUT: OptIdentifierList
OptIdentifierList:
    [IdentifierList]
OptOutFile:
    [OutFile]
OutFile:
    OUTFILE: Identifier
Expected:
    EXPECTED: Identifier
```

Figure 3: BNF grammar of JaTack test specification files. Reserved words are not case sensitive but are here shown in upper case for clarity.

```
testdescr: Example 1
command:
    java Example

io: No arguments
input:
expected:
    empty.res

io: One argument
input:
    1
expected:
    simple.res

testdescr: Example 2
command:
    java Example
io: Write to file
input:
    slask.in 1
outfile:
    slask.out
expected:
    slask.res
```

Figure 4: Example JaTack test specification file.

automate the acceptance tests in a light-weight testing framework.

JaTack was designed with testing of batch-oriented programs in mind and is, at least not in its current implementation, not at all appropriate for testing of interactive GUI-oriented programs. For its intended use, this is not a too large limitation since most of the software produced by students and researchers tend to be batch-oriented. Interactive applications can though, if carefully designed and supplied with a batch mode interface, also be tested with JaTack.

```
testdescr: Acceptance tests for the IPT project
command:    java -classpath .. Sort

io: Acceptanstest --etapplopp2-5--
input:      -k etapplopp2-5/config
outfile:    output.res
expected:   etapplopp2-5/korrekt.txt

io: Acceptanstest --varvlopp-5--
input:          -k varvlopp-5/config
outfile:    output.res
expected:   varvlopp-5/korrekt.txt

io: Acceptanstest --etapplopp_orimligtid--
input:          -k etapplopp_orimligtid/config_e_orimligtid.cfg
outfile:        output.res
expected:   etapplopp_orimligtid/korrekt.txt

io: Acceptanstest --etapplopp_shouldfail--
input:          -k etapplopp_orimligtid/shouldfail.cfg
outfile:    output.res
expected:   etapplopp_orimligtid/korrekt.txt
```

Figure 5: A test specification file from a students project.

```
dvalin% jatack
Starting jatack version 0.04
Checking specFile: acctest.jatack
----- Test: Acceptance tests for the IPT project
----- IO description 1: Acceptanstest --etapplopp2-5--
----- Executing: java -classpath .. Sort  -k etapplopp2-5/config
----- diff output.res etapplopp2-5/korrekt.txt
-----
----- IO description 2: Acceptanstest --varvlopp-5--
----- Executing: java -classpath .. Sort  -k varvlopp-5/config
----- diff output.res varvlopp-5/korrekt.txt
-----
----- IO description 3: Acceptanstest --etapplopp_orimligtid--
----- Executing: java -classpath .. Sort
      -k etapplopp_orimligtid/config_e_orimligtid.cfg
----- diff output.res etapplopp_orimligtid/korrekt.txt
-----
----- IO description 4: Acceptanstest --etapplopp_shouldfail--
----- Executing: java -classpath .. Sort
      -k etapplopp_orimligtid/shouldfail.cfg
----- diff output.res etapplopp_orimligtid/korrekt.txt
8c8
< 5;Görl,Patrik;SMI MK Enköping;KTM;02:53:38;5;00:34:58;00:34:41;\
00:33:49;00:34:33;00:35:37;11:01:00;11:35:58;11:43:00;12:17:41;\
12:26:59;13:00:48;13:16:28;13:51:01;14:11:58;14:47:35
---
> 5;Görl,Patrik;SMI MK Enköping;KTM;02:53:31;5;00:34:58;00:34:41;\
00:33:49;00:34:33;00:35:30;11:01:00;11:35:58;11:43:00;12:17:41;\
12:26:59;13:00:48;13:16:28;13:51:01;14:11:58;14:47:28
-----
```

Figure 6: Result output from students project example run.

The current implementation of JaTack has been used in a XP project course, where the students worked in 8–10 person teams full XP style, with good results. Since the task of running all acceptance tests is so easy to perform with just a few key strokes, acceptance tests will be run through very often, and bugs and anomalies are found quickly.

On top of the further work wish list is an interactive GUI for JaTack that enables one to interactively add or upgrade test cases, and run specific test cases.

A more visual way of presenting the differences between actual and expected output from the tested application, preferably using colors, would also be nice to have.

It would also be interesting to examine the possibilities of a simple common scripting language that could be used for acceptance testing scriptable interactive applications, as discussed in section 2.

References

[Bec99] Kent Beck. *eXtreme Programming Explained*. Addison-Wesley, 1999.

[Cri] Lisa Crispin. The need for speed: Automating acceptance testing in an extreme programming environment. *http://www.xp2001.org/xp2001/conference/papers/Chapter23-Crispin+alii.pdf*.

[HE01] Görel Hedin and Magnusson Eva. Jastadd - a java based system for implementing front-ends. In *Proceedings of LDTA'01, First Workshop on Language Descriptions, Tools and Applications (in conjuction to ETAPS), Genova, Italy*, April 2001.

[JAH01] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. The XP Series. Addison-Wesley, 2001.

[jun] *http://www.junit.org*.

[MC] Roy D. Miller and Christopher T. Collins. Acceptance testing. *http://www.xpuniverse.org*.

[Met] Java-cc parser generator. Metamata Inc. *http://www.metamata.com*.

[Nau60] Peter Naur. Revised report on the algorithmic language algol 60. *Communication os the ACM*, 3(5):299–314, May 1960.

[Win] Winrunner homepage. *hhtp://www-svca.mercuryinteractive.com/products/winrunner/*.