Developing acceptance tests from existing documentation using annotations: An experiment

David Connolly, Frank Keenan and Fergal Mc Caffery
Dundalk Institute of Technology
Software Technology Research Centre (SToRC)
Dublin Road, Dundalk, Ireland.
{david.connolly,frank.keenan,fergal.mccaffery}@dkit.ie

Abstract

The importance of good software testing is often reported. Traditionally, acceptance testing is the last stage of the testing process before release to the customer. Unfortunately, it is not always appropriate to wait so long for customer feedback. Emerging agile methods recognise this and promote close interaction between the customer and developers for early acceptance testing, often before implementation commences. Indeed, Acceptance Test Driven Development (ATDD) is a process that uses customer interaction to define tests and tool support to automate and execute these. However, with existing tools, tests are usually written from new descriptions or rewritten from existing documentation. Here, the challenge is to allow developers and customers to annotate existing documentation and automatically generate acceptance tests without rewrites or new descriptions. This paper introduces the related ideas and describes a particular experiment that assesses the value of using annotated text to create acceptance tests.

1. Introduction

A large part of software development expenditure is attributed to *testing*. Traditionally, with plan-driven development, *acceptance testing*, the process of testing functional requirements with "data supplied by the customer" [1] occurs as the final stage of the development process long after the initial investigation has completed [2]. Many reports, however, highlight that costs can be reduced by detecting errors earlier in development [3]. Also supporting this, in many domains, such as the medical device industry, software is developed subject to a regulatory environment with a tendency for extensive documentation. Despite many constraints already being specified, these are often ignored with acceptance tests written from scratch after implementation is complete. In contrast, agile approaches require constant customer collaboration throughout development, with customer provision of acceptance tests being an important part of this role. Often, it is recommended that tests be identified before implementation commences.

In eXtreme Programming (XP) [4], for example, acceptance tests are defined as a part of the *User Stories* practice and, as such, are written before coding of the story begins. In this context, functional tests are synonymous with acceptance tests [5]. Further, for accurate user stories, Cohn recommends customers themselves specify acceptance tests with developers and testers providing support as required [6]. The XP practice of *Continuous Integration,* that is, building and testing a system frequently, maximizes the use of the executable and automated products of *Test Driven Development (TDD)* [7]. TDD visibly links executable unit tests to the overall development process. TDD is widely practised and has many reported benefits [8] but successful use does rely on tools such as JUnit [9].

ATDD adds to this established test-first philosophy with acceptance testing of an automated and executable nature. In keeping with agile principles, ideally customers write acceptance tests guided by developers. Its practice "allows software development to be driven by the requirements" [10]. A key advantage of ATDD in its wider context is that it leverages existing agile infrastructure supporting continuous integration.

As with TDD, support from tools makes ATDD feasible. However, Andrea [11] claims that existing tools exhibit several deficiencies and produce tests that are "hard to write and maintain". To overcome this Andrea also suggests that the next generation of functional testing tools need to support writing (and reading) functional tests in multiple formats.

Given the widespread adoption of information and communication technology, in many organisations business rules are documented in numerous formats, for example, in web based Content Management Systems. However, ATDD is currently not well supported with tools that enable reusing such existing documents, without rewrites, to create executable tests. A challenge, therefore, is to support a customer in easily creating tests from existing material. However, successful identification of accurate acceptance tests in this manner is not necessarily straightforward.

Key challenges include *Under-Specification* and *Over-Specification*. While both refer to the accuracy of test specification, each has distinct effects on testing outcomes. In an *Under-Specified* test, elements that should cause the test to fail have not been identified and included in a test. This has the consequence of a test passing as a *false positive*. In an environment, using continuous integration such a false positive is costly because it appears that work is progressing correctly therefore delaying the process of bug discovery. In contrast, an *Over-Specified* test is one where superfluous elements that should not cause the test to fail have been identified and included in the test. A common cause of over-specification at this level of granularity is inclusion of implementation details in a test. At some point in the development cycle, these elements are likely to change and cause the test to fail as a *false negative*. This devalues testing effort and forces either rewrites of a test at the correct specification level or exclusion of a test, even if, in principle, a testable part of the system goes untested.

In general, this research examines the automatic generation of acceptance tests from existing electronic documents by a responsible and knowledgeable customer working in close collaboration with the development team. Such a customer will work by identifying and helping to annotate existing documents. A staged adoption of any prototype is planed with developers first helping customers to annotate documents. Once confidence exists, it is planned to move to customers annotating and developer reviewing. This paper examines the potential benefit of using annotated documents to identify test cases. The second section summarises related work and progress realised so far. This is followed in section three by an overview of an exploratory experiment concerning annotations. Finally, an outline of future work is included with conclusions in section four.

2. Related Work

Many approaches to conducting acceptance testing exist. Some concentrate on acting as a "recording device" allowing user actions to be replayed against a system, checking for deviations. However, this approach is mainly limited to Graphical User Interface (GUI) testing of a specific version of a system, using a tool such as the Selenium IDE [13].

Tools for writing acceptance tests in a customer friendly format and appropriate for continuous integration exist. RSpec, for example, is a "Behaviour Driven Development framework for Ruby" [14]. It promotes a workflow that involves writing stories in a somewhat prescriptive natural language style and then manually translating these steps into Ruby. While the authors consider this approach interesting for new stories, it has limitations in dealing with existing documents that were written without reference to RSpec's style of writing tests.

Other open source tools aimed at supporting ATDD exist including EasyAccept that has a script syntax that supports tests written in both tabular and sequential style [15].

Generally, the Framework for Integrated Tests (FIT) is the most widely accepted tool for managing acceptance tests in agile development and therefore practising ATDD [16]. In FIT's simplest workflow a user, places inputs and some expected output into a tabular format, a *ColumnFixture* [17]. The developer then writes code (*fixtures)* that executes this data against the system's production code.

A simple example of arbitrary precision integer multiplication is represented as a FIT table in Figure 1 and the associated fixture code in Figure 2. This example, while trivial, still differs from a Unit Test, as it does not test implementation details such as null checking. Implementation details can be appropriate in a unit test but at this level of granularity, it would represent over specification. In a *ColumnFixture* table, the first row is a reference to the fixture code; the second row contains structural elements in the form of labels for data, *t1* and *t2*, or actions, *product().* Subsequent rows contain inputs and expected responses. The third row, for example, contains a testable event, here; from this, the fixture code, *t1* is initialized with "0", *t2* is initialized with "0" and the product() method returns "0". The fourth and subsequent rows are further events. Events may depend on previous events e.g. tests can have state or be stateless; this is an implementation detail of the system and fixture code.

multiplication.AIntegerFixture		
t1	**t2**	**product()**
0	0	0
1	1	1
1	2	2
2147483647	2	4294967294
9223372036854775807	2	18446744073709551614

Figure 1 Example FIT Table

Other built-in fixture included in FIT include *ActionFixtures* for testing a “sequence of commands” and *RowFixtures* for “comparing test data to objects in the system” [17].

```
public class AIntegerFixture  extends
ColumnFixture {
  public String t1;
  public String t2;

  public String product() {
    return AInteger.multiply(t1, t2);
  }
}
```

Figure 2 Example FIT Fixture Code

FitNesse is a Wiki framework developed to support FIT [18]. It facilitates the editing of FIT tables in a browser allowing non-programming experts to add content. While FIT tables can be written in any tool that can export HTML, such as Microsoft Excel, these generic tools do not have any authoring features directly supporting the task domain. Existing tools that support either FIT or FitNesse include AutAT and FitClipse. AutAT seeks to assist “business-side people” taking a visual approach to building Acceptance Tests [19]. As FitClipse [20] builds on FitNesse tests are entered using its wiki syntax. Mugridge introduces a process based around a library of *fixtures* named FitLibrary, which improves FIT’s “business-level expressiveness” to emphasise a “domain-driven design approach” [21]. It supports a type of fixture, *DoFixtures,* which approach natural language in readability.

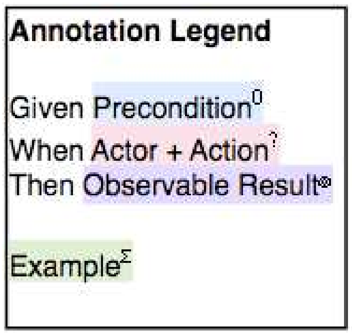
Commercial software also supports such a workflow, with GreenPepper [22] supporting “executable specifications” while providing an expressive library of table types. For clarity, it is important to note that GreenPepper uses code annotations (Java and C#) that are unrelated to the annotations in this paper.

However, none of these tools is focused on reusing existing documentation, so unlike the proposed approach these approaches require re-writes of content. In the requirements authoring process, Melnik and Maurer found that the use of FIT helped students to “learn how express requirements in a precise, unequivocal manner” [23]. In a number of experiments aimed at evaluating the impact of FIT tables on the implementation of change requests Ricca et al. [24], found improvement in the correctness of code produced. The addition of FIT Tables to plain text descriptions had the most impact on more experienced students, and they found no significant increase in time taken to implement the changes.

The use of annotations was proposed because it provides users with a simple conceptual framework allowing them to add detail to text descriptions of tests. Annotations are used here to allow for links to be made between descriptions and corresponding FIT Tables. These annotations are based on elements of an acceptance test description recommended by Jain [16]. There are four basic types, covering most elements of an individual acceptance test:

- *Precondition*: event that must occur before a test is run,
- *Actor + Action:* part of system and functionality.
- *Observerable Result*: a verifiable response generated by the system,
- *Examples*: represent the input data given to a test.

The passing or failure of test resets with variance from specified *Observerable Results*. A visual representation of the annotations is contained in Figure 3. Here, each annotation type is modified to include a unique colour shade and a greyscale symbol to make them more easily identifiable.


Annotation Legend

Given Precondition^{0}
When Actor + Action$^{?}$
Then Observable Result$^{\circledast}$

Example$^{\Sigma}$

Figure 3. Annotation Legend

3. Annotations Experiment

The research question to be investigated is *to what extent can Acceptance Test Driven Development be improved by supporting the elicitation of executable acceptance tests from existing text?*

This experiment was designed to evaluate the impact of annotations on the process of authoring acceptance tests. The scenario used to write the question descriptions given to respondents concerned the management of software packages on a computer system, such as GNU/Linux [25].

There were six participants, each experienced in computing as either a postgraduate or professional. However, none had prior experience of writing FIT tables. All were given a short, two-hour training session on FIT Tables and ATDD. Participants were tasked to create tests using either annotated descriptions or from non-annotated plain text descriptions. The plain text descriptions serve as a reference for comparison against annotations. The only difference between descriptions was the presence or absence of annotations. Three participants were randomly assigned to Group A and Group B and each group received four questions. Group B started with annotated descriptions while Group A were given a non-annotated version. For subsequent exercises, the groups alternated between annotated and non-annotated. Apart from a common assignment of question, to their group, participants worked alone. In providing these descriptions, the first author acted in the role of a customer on an agile project. The experiment considered annotations in paper–based experiment in isolation aside from usability considerations of prototypes.

3.1. Design

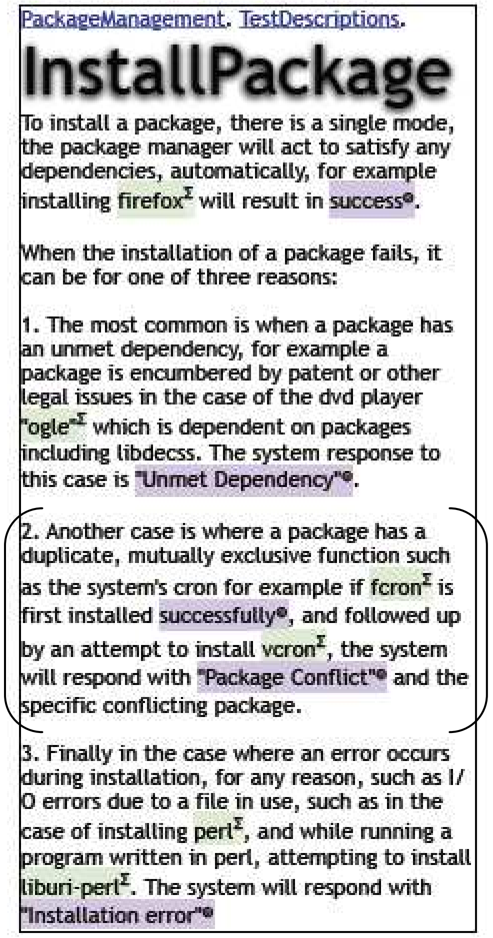
For comparison purposes, the first author wrote reference tests, providing an "ideal" test description against which the participants' tests were compared. Each was in the form of high-level descriptions of how a system should function, including handling of error conditions and intended to be of approximately equal difficulty: Question 1 covered *initial bootstrap* of the package management system; Question 2 covered *installation* of new packages; Question 3 covered *removal* of packages; Question 4 covered *upgrading* of packages. The metrics used to assess the experiment were gathered under the following headings:

- *Errors*: elements that should not appear in the test. From participants' answers, all error occurrences counted towards the average.
- *Correct Elements*: The first occurrence of an element in participants' answer. Participants were free to reuse structural elements (for example the first row in a FIT Table) as this only affects readability. However, repeated data elements are counted as *Errors*. Presence of a data element irrespective of corresponding structural element was enough for it to count as correct, so two penalising respondents twice.
- *Missing Elements*: defined as elements that were omitted by the participants compared to the reference test.
- *Time*: amount of time taken to complete FIT table.

After the experiment, participants were required to respond to a short survey with questions formulated using the Likert scale for responses.

3.2. Question and Responses

A reproduction of Question 2 with annotated text is presented in Figure 4. This version was provided to Group A with Group B receiving a non-annotated plain text version.



PackageManagement. TestDescriptions.

InstallPackage

To install a package, there is a single mode, the package manager will act to satisfy any dependencies, automatically, for example installing firefox$^{\Sigma}$ will result in success●.

When the installation of a package fails, it can be for one of three reasons:

1. The most common is when a package has an unmet dependency, for example a package is encumbered by patent or other legal issues in the case of the dvd player "ogle"$^{\Sigma}$ which is dependent on packages including libdecss. The system response to this case is "Unmet Dependency"●.

2. Another case is where a package has a duplicate, mutually exclusive function such as the system's cron for example if fcron$^{\Sigma}$ is first installed successfully●, and followed up by an attempt to install vcron$^{\Sigma}$, the system will respond with "Package Conflict"● and the specific conflicting package.

3. Finally in the case where an error occurs during installation, for any reason, such as I/O errors due to a file in use, such as in the case of installing perl$^{\Sigma}$, and while running a program written in perl, attempting to install liburi-perl$^{\Sigma}$. The system will respond with "Installation error"●

Figure 4. Sample question

A simple FIT Table (*ColumnFixture*) reproduced in Figure 5 represents the bracketed text of Figure 4.

package	install()	failReason?	failPackages?
fcron	TRUE		
vcron	FALSE	Package conflict	fcron

Figure 5. Sample 'ideal' answer

This table acknowledges the flow of events encoded in the text and unambiguously represents the specific package name of the “conflicting package”. For illustration and comparison with the “ideal” response, two respondent answers are provided in Figure 6 and Figure 7. Figure 6 reproduces the answer attempt from respondent A2, who had been provided an annotated version of Question 1.

package	**success?**	
fcron	TRUE	
vcron	FALSE	“Package Conflict”

Figure 6. Respondent snippet (annotations)

Here, the respondent A2 correctly identifies the sequence of events, but fails to include the name of the package, “fcron”, causing the failure. However, the chosen label heading “success?” does not reflect the action name but this is not considered an error because the respondent correctly labelled the table. Respondent A2 achieved the fewest Errors and both the most Correct Elements and fewest Missing Elements in Question 2.

The corresponding snippet from respondent B1, who had used a non-annotated version, is reproduced in Figure 7.

Install	**Result**
Duplicate (vcron)	Package Conflict

Figure 7. Respondent answer (non-annotated)

Here, the respondent B1 failed to identify from the text that the ‘install()’ action should fail due to the prior installation of a conflicting package. Indeed respondent B1 didn’t correctly identify “install()” as an action at all, instead specifying the package name “vcron” combined with the error detail as data to be verified.

In comparing these answers with the reference answer in Figure 6 one element was missed by respondent A2 while four elements were missed by respondent B1 in Figure 7.

Finally, it should also be noted that respondent B1 performed better when using annotated texts and respondent A2 performed worse when using non-annotated texts. The next section summarises the overall results for the experiment.

3.3 Results

The results gathered from the respondents answers, are summarised in Table 1. For clarity, the *row number* is included in column 1. Columns 2, 3 and 4 introduce the *question number*, which *group* is responding (A or B) and the *type* of description provided in the group’s question. Columns 5, 6 and 7 contain the arithmetic mean of the counts for each group’s *Errors*, *Correct Elements* and *Missing Elements*, respectively. The presence of *Errors* indicates *Over-Specification* while that of *Missing Elements* indicates *Under-Specification*. In all cases, *Correct Elements* plus *Missing Elements* equals Total Elements of the “ideal” answer.

We analysed both the data element and the structural element of the responses. An *Error* occurs whenever a response is matched against the “ideal” answer and a mistake is identified. A mistake may be identified in either the data element or the structural element. All mistakes that occur in the data element are counted as errors, whereas only the first occurrence is counted as an error in the structural element. For example, if we matched an individual’s response against the “ideal” response and discovered that a data element “*fcron”* had been included by a respondent three times; the first two match the “ideal” response counting as *Correct* but the third element would be incorrect and count as one error.

Table 1. Results from annotations experiment

Row	Q	Group	Type	Errors	Correct	Missing
1	Q1	B	Annotated	7.33	12	14
2	Q1	A	Plain	13	14	12
3	Q1	-	Difference	55.74%	**(15.38%)**	**(15.38%)**
4	Q2	A	Annotated	4.67	14	7
5	Q2	B	Plain	9.67	10.67	10.33
6	Q2	-	Difference	69.77%	27.03%	38.46%
7	Q3	B	Annotated	9.67	9.67	3.33
8	Q3	A	Plain	11.67	9	4
9	Q3	-	Difference	18.75%	7.14%	18.18%
10	Q4	A	Annotated	11.5	14	6
11	Q4	B	Plain	14	8.67	11.33
12	Q4	-	Difference	19.61%	47.06%	61.54%
13	-	-	Average Difference	40.97%	16.46%	25.70%

Each row in Table 1 presents the results of one group for a particular question. For example, Row *1* represents the arithmetic mean of responses from Group B for Question 1 (annotated). In the case of Row *1*, the three members of Group B obtained an average of 7.33 errors, an average of 12 correct elements and 14 missing elements, with the total elements of the “ideal” answer being 26 (12 +14). While Row *2* represents the arithmetic mean of responses from Group A for Question 1 (plain, non-annotated). Further, the percentage difference (55.74%) between Group A and Group B is represented in row 3. This is obtained from as follows:

Row 3 = ((|Row 1 – Row 2|)/ (Row 1 + Row 2)/2)*100. For example, in the case of the obtaining the percentage difference of *Errors*:

55.74% = (|7.33 – 13|) / ((7.33 + 13) /2) * 100

In the case of a worse performance when given annotations, the result will be distinguished in Table 1 by making it appear in bold. This pattern continues for each question given to respondents.

The final row, Row *13*, contains the overall percentage difference; these results included the cases of decreased performance in Row *3* as negative numbers.

In each case, the occurrence of *Errors* is significantly reduced for the annotated versions. This holds across both groups even with a pattern of Group A taking less time on average compared to Group B. For example, the figure of 55.74% in row 3 indicates that there were 55.74% less errors identified in the annotated version. This means responses with a lower incidence of *Over-Specification* occurred when respondents were provided with annotations.

In Question 2 to Question 4, the average number of *Correct Elements* for the annotated version is greater than that for the non-annotated version. A similar reduction in the number of *Missing Elements* occurred. For example, 27.03%, *Correct* in Row *6* means that there were 27.03% more elements identified by the group given annotations. Similarly, 38.46%, M*issing* in Row *7* means that there were 38.46% less missing elements identified by the group given annotations.

As with *Error Rates*, the number of *Correct Elements* achieved by respondents appears unrelated to the amount of time spent. However, the effect of annotations on *Correct Elements* and *Missing Elements* was smaller than on the *Error Rates*, therefore annotations had less of an impact on *Under-Specification*.

3.4.1. Participant Observations and Survey A questionnaire participant noted "Annotations were helpful to identify potential tests" but qualified it by saying that it led to "ignoring the rest of text"; the quality of annotations is a major issue. Another participant noted that annotations were particularly useful when first getting to grips with the problem domain. Another respondent felt it would have been desirable to view a fully worked up example in the problem domain before responding.

In general, participants stated that they were somewhat unfamiliar with the problem domain and were new to writing FIT Tables. However, respondents were quite positive about the benefit of annotations. The responses to the question *did annotations help to clarify the descriptions provided.* Are reproduced in Figure 8 with the X-axis representing the Likert scale from 1 (Disagree Strongly) to 5 (Agree Strongly). The central tendency is represented by the median, corresponding to "Agree".

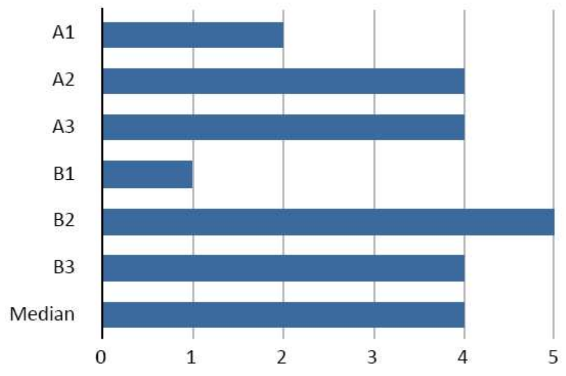


Figure 8. Response to question on benefit of annotation

3.4.2. Limitations The small number of participants makes it difficult to draw generalisable conclusions based on quantitative analysis. Also, as the question descriptions and reviews of responses were both conducted by the authors there is a potential for experimental bias. This will be addressed in future work by adopting a marking scheme approach with multiple reviewers who were not involved with the authoring of descriptions.

4. Conclusions and Future Work

This paper has argued that software testing can be enhanced by improving support for ATDD in the situation where constraints are already defined in elec1tronic documents. The experimental results presented here considered the value of using annotated documents to author acceptance represented by FIT Tables. The results indicate that using annotated documents helped to identify more *elements* that are *Correct* with fewer *Missing* elements and *Errors* when creating acceptance tests. Additionally this helps in reducing the possibility of specifying tests incorrectly and of clarifying existing business rules.

The results from this limited study have been very encouraging and further investigation is planned. In particular, tool support is being examined. This should allow customers to identify acceptance tests from existing documents in a collaborative environment. This will be facilitated though development of text editor add-on for FitNesse that supports the definition of FIT Tables from pre-existing documents by customers using the annotations described here. In addition a new type of annotation describing grouping will be defined which should help to clarify independence of tests that look similar.

The experiment consider the operation of annotations in the abstract, there effects as part of a full software engineering process will be studied later with future work focusing on evaluating the use of annotations by relevant stakeholders, especially the collaborative use of annotations by customers and developers. Facilitation of self-assessment though collaboration by users of the prototype is a key prototype requirement. Tool support will allow future experimental evaluations with, for example, a larger sample group from industry. These experiments will form part of the next step towards answering the research question.

5. Acknowledgements

This research is partially supported by Institutes of Technology, Technological Sector Research Programme, Strand 1 Fund and Science Foundation Ireland through the Stokes Lectureship Programme, grant number 07/SK/I1299.

6. References

[1] Sommerville, I. Software Engineering, 8th edition, pages 80-81, Addison-Wesley, 2007.

[2] Pressman, R. S. Software Engineering: A Practitioner's Approach, European Adaption, 5th edition. McGraw-Hill, 2000.

[3] G. Tassey, "The economic impacts of inadequate infrastructure for software testing" National Institute of Standards and Technology (NIST), May 2002.

[4] Beck, K. and C. Andres. Extreme Programming Explained: Embrace Change, 2nd ed, Addison Wesley, Boston, 2005.

[5] Sauvé, J. P. and Neto, O. L. A. Teaching software development with ATDD and EasyAccept. In SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer Science Education, pages 542–546, ACM, 2008.

[6] Cohn, M. User Stories Applied. Addison-Wesley, Boston, 2005.

[7] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002.

[8] Jeffries, R. and Melnik, G. Guest editors' introduction: TDD–the art of fearless programming. IEEE Software, volume 24(3), 2007, pages 24–30.

[9] Kent Beck, Erich Gamma, and David Saff. JUnit 4. Website, URL last accessed 16th January 2009: http://junit.sourceforge.net/

[10] Park, S.S. and Maurer, F. The benefits and challenges of executable acceptance testing, in APOS '08: Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral, ACM, New York, NY, USA, 2008, pages. 19–22.

[11] Andrea, J. Envisioning the next generation of functional testing tools. IEEE Software, volume 24(03), 2007, pages 58–66.

[12] Cunningham, W. FIT: Framework for Integrated Testing. Website, URL last retrieved 21st April, 2008: http://fit.c2.org.

[13] Kasatani, S. Selenium IDE. Website, URL last accessed 1st December 2008: http://seleniumhq.org/

[14] RSpec Development Team. Website, URL last accessed 1st December 2008: http://rspec.info

[15] Sauvé, J.P., Cirne, W., Osorinho and Coelho, R. EasyAccept Sourceforge Project. Website URL last accessed 3rd Dec 2008: http://easyaccept.sourceforge.net

[16] Jain, N. Acceptance Test Driven Development. Presentation URL last accessed 30th Nov 2008. http://www.slideshare.net/nashjain/acceptance-test-driven-development-350264/

[17] W. Cunningham, Framework for Integrated Test, September 2002. Website, URL last accessed 16th January 2009: http://fit.c2.com

[18] FitNesse.org. Website, URL last accessed 7th February 2008: http://fitnesse.org

[19] Schwarz, C., Skytteren, S. K., and Øvstetun, T. M. AutAT: an eclipse plugin for automatic acceptance testing of web applications. In OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on OOPSLA, 2005, pages 182–183.

[20] Deng, C., Wilson, P., and Maurer, F. FitClipse: A FIT-based Eclipse plug-in for Executable Acceptance Test Driven Development. In proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2007). 2007.

[21] R. Mugridge, Managing agile project requirements with storytest-driven development, Software, IEEE, 25 (Jan.-Feb. 2008), pp. 68–75, 2008.

[22] Pyxis Technologies inc., GreenPepper Sofware, Website, URL last accessed 19th January 2009: http://www.greenpeppersoftware.com/confluence

[23] Melnik, G. and Maurer, F. The practice of specifying requirements using executable acceptance tests in computer science courses. In OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on OOPSLA, pages 365–370, ACM, 2005.

[24] F. Ricca, M. D. Penta, M. Torchiano, P. Tonella, M. Ceccato, and C. A. Visaggio, Are fit tables really talking?: a series of experiments to understand whether fit tables are useful during evolution tasks, in ICSE '08: Proceedings of the 30th international conference on Software engineering, New York, NY, USA, 2008, ACM, pp. 361–370.

[25] Free Software Foundation, About the GNU Project. Website, URL last access 16th January 2009: http://www.gnu.org/gnu/the-gnu-project.html