EasyAccept: A Tool to Easily Create, Run and Drive Development with Automated Acceptance Tests

Jacques Philippe SAUVÉ
Univ. Fed. de Campina Grande (UFCG)
Av. Aprígio Veloso, 882, Bodocongó
Campina Grande, Paraíba, Brazil
+55-83-3310-1122
jacques@dsc.ufcg.edu.br

Osório Lopes ABATH Neto
Univ. Fed. de Campina Grande (UFCG)
Av. João Maurício, 1461, Manaíra
João Pessoa, Paraíba, Brazil
+55-83-32463902
osorinho@gmail.com

Walfredo CIRNE
Univ. Fed. de Campina Grande (UFCG)
Av. Aprígio Veloso, 882, Bodocongó
Campina Grande, Paraíba, Brazil
+55-83-33101365
walfredo@dsc.ufcg.edu.br

ABSTRACT

This paper introduces EasyAccept, a tool to create and run client-readable acceptance tests easily, and describes how it can be used to allow a simple but powerful acceptance-test driven software development (ATDD) approach. EasyAccept takes acceptance tests enclosing business rules and a Façade to access the software under development, and checks if the outputs of the software's execution match expected results from the tests. Driven by EasyAccept runs, software can be constructed with focus, control and correctness, since the acceptance tests also serve as automated regression tests. Results of experiments with undergraduate students demonstrate the benefits of the ATDD approach using EasyAccept and show that this tool can also help to teach and train good testing and development practices.

Categories and Subject Descriptors
D.2.1 [Requirements/Specifications] – elicitation methods, languages, methodologies, tools
D.2.5 [Testing and Debugging] – testing tools

General Terms
Measurement, Design, Experimentation, Languages, Verification

Keywords
acceptance testing, test-driven development, automated testing

1. INTRODUCTION

Kent Beck first mentioned the term acceptance-test driven development (ATDD) in his book Test-Driven Development by Example [1]. Sometimes also called Story Test-Driven Development (STDD) [2], it consists in extending the application of test-driven development (TDD) techniques to the acceptance testing level, using these user-level black box tests instead of (or in addition to) unit tests. For the approach to work, acceptance tests must not be written in an arcane (to the client) programming language that is only understood by developers (this is typically the case in the industry today), because they must be understood by clients. This way, developers can be provided with automated client-verifiable artifacts created in cooperation with clients that can be used to test and drive software development. The chief benefits of acceptance tests created in this fashion are:

- they serve as a precise, effective medium of communication between clients and developers both syntactically and semantically, reducing the communication gap between them, which is still a major source of errors in software development.
- analysis can be made more cost-effective by diverting its effort toward creating executable artifacts that can be automatically used to test software – requirements and business rules are automatically enforced through the tests, rather than suggested by text descriptions and diagrams that are imprecise and prone to misinterpretation by developers.
- with executable analysis artifacts, analysis remains synchronized with the software produced in an automated way, which can not usually be done with “regular” artifacts. This is even more important in projects with unknown, unclear and changing requirements.
- they are both client-readable tests and analysis artifacts at the same time, thus enabling clients to help in test creation and improvement effort – requirements changes become test updates
- developers continually find errors in the tests and suggest new tests, but the client must always review and approve them – this helps improving test quality
- they serve as a quality agreement between clients and developers, setting an overall acceptance goal to the project that can be automatically verified.
- feature creep is prevented because developers keep focused on making the client-approved suite of acceptance tests pass.
- some non-functional requirements can also be enclosed in acceptance tests, allowing these kinds of requirements to drive development too.
at any point of the development, both clients and developers can know, based on a test run, which features are working completely and which are buggy. A full set of acceptance tests also represent an automated suite of regression tests. This gives the project more control and clients and developers gain more confidence in the work being done.

Experience with an ATDD approach has provided evidence for these benefits, as seen in [3], [4] and [5]. However, in spite of its many advantages, only recently has acceptance test-driven development been made feasible, with the introduction of acceptance testing tools that integrate clients and developers in a common framework. This follows a similar pattern that happened with unit tests: they only became well-established in the industry with the emergence of the xUnit family of unit testing tools [6]. These tools allowed the benefits of test-driven development to be effectively put into practice with a standardized, clever and simple method that can be swiftly integrated with software development.

The most widely known tool for acceptance testing is FiT (Framework for Integrated Testing) [7], created by Ward Cunningham. It is based on input-output table tests that can be created by clients in HTML files. Other acceptance testing tools include Exactor [8], which uses textual scripts, JAccept [9], based on a graphical editor and XML test files, and TextTest [10], which tests programs with command-line textual input and output.

This paper presents an acceptance testing tool called EasyAccept, an open-source tool that can be found at [11]. Its main qualities are the understandability and expressiveness of the acceptance tests clients and developers are able to write and the tool's suitability to be incorporated in an acceptance test-driven approach for software development. Furthermore, a simple method is described showing how this can be done.

The paper is organized as follows. In section 2, EasyAccept is described, illustrating its operation and discussing its main features. A brief comparison with other tools is made. In section 3, an ATDD approach using EasyAccept is outlined and illustrated. Section 4 presents and discusses results of some experiments that were performed in undergraduate software design courses using EasyAccept. In the final section we present the conclusions.

2. HOW EASYACCEPT WORKS

In short, EasyAccept is a script interpreter and runner. It takes tests enclosed in one or more text files and a Façade to the program that will be tested. Accessing the program through Façade methods that match user-created script commands, EasyAccept runs the entire suite of tests and evaluates actual and expected outputs or behaviors of the program under test. In a test report, the tool shows divergences between actual and expected results, or a single message indicating all tests were run correctly. Fig. 1 shows an outline of the tool's operation.

User-created commands define a script language which should be co-defined by clients and developers. A simple method can be employed to create a script language directly from user story descriptions, as will be discussed in the next section. Scripts can be written using any text editor, combining built-in and user created commands to express the tests. A sample script for a Monopoly game user story can be found in Figure 2. Additional scripts can be found in [11].

The overhead of getting started with EasyAccept is practically zero, and it requires minimal additional work on the part of the developers. They only need to provide a Façade to the program to be tested containing methods whose signatures match the user-created commands. A single Façade that exposes the program's
business logic helps separate business and user interface concerns, and may even already exist in programs not created with an ATDD approach, since this separation is an advocated architectural best practice. A sample Façade created in Java for the script of Figure 2 can be seen in Figure 3.

Compared to other acceptance testing tools, one of the main advantages of EasyAccept is its expressiveness to create tests both sequentially and in a tabular format. Table-driven tests are best suited to express business rule examples in input-output pairs that can be linked to the business logic algorithmically. On the other hand, sequential command-driven tests are suited to express the business logic workflow. Being primarily a text-based tool, EasyAccept also has table testing capabilities through the built-in command `expectTable`. Other text-driven tools [8, 9, 10] don’t have table-driven capabilities. Conversely, FiT [7], a strictly table-driven tool, must rely on sequences of tables, each with its own design, when it comes to express sequential tests made up of several different commands.

Another advantage of EasyAccept over other tools is the lesser burden imposed on the developer side to adapt to its test format: a single Façade is needed, which may even already exist for software already under construction. Moreover, the single Façade can be naturally integrated with a test-driven development approach, as is detailed in the next section. FiT requires developers to design individual fixture classes with hookup code for every type of table used in the tests and cope with data being referenced across tables. Other textual testing tools use various approaches, none of which involves the use of a single Façade.

3. ATDD WITH EASYACCEPT

In this section, we present the basics of an ATDD approach that uses EasyAccept as the main tool. It consists of the core activities of a development process based on Extreme Programming [12] and TDD [1] practices. A high-level outline of the approach is depicted in Figure 4. Due to space limitations, we have summarized the exposition of the approach. For a full-fledged discussion, see reference [13].

Fig. 2 – A sample EasyAccept script

Fig. 3 – A sample Façade used with EasyAccept

Fig. 4 – An outline of the ATDD core activities

Three groups of activities are defined: planning activities, activities performed in an iteration to implement a new user story and maintenance activities that are done throughout iterations until the complete software is delivered.

3.1 Planning Activities

Before development begins, clients and developers join for a few meetings in a planning activity. Clients write user stories, developers estimate the time they need to implement them, clients prioritize user stories based on developers’ time estimates. As a result of this activity, a release plan is created. Among other high-level planning decisions, it defines the order in which user stories will be implemented. At the end of the planning activity, clients must also provide a list of non-functional requirements
(performance, response time, and other kinds of restrictions), some of which can also serve to create acceptance tests.

The implementation of user stories is done in an iterative fashion. Iterations are short timeframes (typically two weeks) after which working software (implemented user stories) is delivered. For each iteration, an iteration plan is created to describe in detail which user stories will be implemented and how (which tasks will be needed).

As development progresses, the release plan can be updated with the addition or modification of the order of user stories that will be implemented.

### 3.2 Iterations

At each iteration, a number of user stories is implemented using ATDD. Before any code is written, however, scripted acceptance tests that define the rules and behavior for the user stories must be created. For each user story, the sequence of steps is to create a script language, to create the acceptance tests and to implement using test-driven techniques.

An important case occurs during the first iteration, though. It is advisable that the tests for the non-functional requirements that can actually be tested (i.e., that are measurable, not prone to subjective interpretation, etc.) be created during the first iterations, so that all the code written thereafter is forced to make those tests pass, assuring software compliance with these special types of requirements. They are hard to tackle later in development if they are not thought about in advance. Other areas to which some effort should be invested in an upfront manner are architectural design and database design, because they are costly to change later.

#### 3.2.1 Creating a script language

The script language defines a “vocabulary” of commands that will be used to write the tests. It is the first step in the implementation of a user story. For the first user story, the script language consists merely of the commands required to write its tests. For each subsequent user story, the script language is augmented with additional commands that will be needed to create their tests. As development follows and iterations are completed, this step tends to become shorter for additional user stories, because commands that serve multiple user stories are reused and need not be defined.

There are basically three types of script commands: getters, doers and preparers. **Getters** are commands that capture a current characteristic or state of a program’s entity or element. In a test script, they are typically coupled with an `expect` built-in command to check if the value returned corresponds to the value expected. Ex.: `getEmployeeWage`, `getCustomerName` and their arguments.

**Doers** are commands that correspond to actions taken in the program’s business logic. Generally, a doer is coupled to an actual operation issued by a user when operating the software, or to a step in the sequence of operations that, when combined, correspond to such a user action. Ex.: `payEmployee`, `createOrder` and their arguments.

**Preparers** are ancillary commands in the script language. They exist only to serve the purpose of facilitating the writing of test cases, by setting up special situations that would only be obtained through a large number of doers. Ex.: `createTestDatabase`

Most script commands can be derived directly from a well-written user story, almost automatically. Taking a user story definition, one must follow three simple rules: domain nouns are associated with a getter, verbs representing actions users of the software can take become doers, and verbs representing actions of the software not triggered by users are candidates for preparers. The remaining required or useful commands can be defined during test creation or test refactoring. An example of this technique can be seen in Box 1.

#### 3.2.2 Creating acceptance tests

The test creation effort is done cooperatively between clients and developers. Although the ideal situation would have clients writing all tests themselves, this is typically hard to achieve: clients have a declarative need (user story), but testing involves an algorithmic specification that needs a background they typically don’t have. However, creating acceptance tests by analysts in close contact with clients works well if the tests are frequently reviewed by the client. Developers also aid in the testing effort by suggesting new test cases (which should always be approved and reviewed by the client) and raising suspicion of errors in the tests.

Some acceptance test creation patterns include:

- **construct tests as sequences of preparers, doers and `expect` or `expectError` commands coupled with getters**;
- **for every new script command created, think out and include `expectError` commands for invalid arguments or situations**;
- **for every new script command created, define boundaries and limits, if applicable, and check them with `expect` commands in the inner boundaries and `expectError` commands in the outer boundaries**

A list of patterns for creating acceptance tests with EasyAccept is being built and can be found in [14].
3.2.3 Implementation

Developers (implementers) use the following rules, taking one test case from the user story suite of tests at a time:

1. Write the simplest code that makes the acceptance test pass – first, code the methods in the Façade that correspond to the commands in the script, if they don’t already exist; then, from the Façade methods, think out the actual program’s classes, and continue from this point down to the code.
2. If you think there is an error in an acceptance test, submit it to the client for review (finding errors in the acceptance tests forces developers to think about business rules and requirements, promoting discussion and consequently understanding of what needs to be done).
3. Change correct code only if you have proved it is wrong by creating a new acceptance test case that breaks it (and this test must be reviewed by the client), or if you are refactoring code (see the continuous activity refactoring).
4. If, to make the acceptance test pass, you need to create untested structures not directly accessible or testable by any acceptance test (because they involve infrastructure entities such as a database connection, or object caches, for example), first write unit tests for the structure, then code what you need, using the preceding rules substituting “acceptance” by “unit”, and disregarding references to the client.

The iteration is considered complete when all acceptance tests created for its user stories pass, all acceptance tests for completed user stories (which become automated regression tests) continue passing, including the tests for non-functional requirements, and all developer unit tests also pass. With regard to business rules and requirements compliance, as well as software correctness, the project can be considered complete at the end of the last iteration. Final delivery and acceptance will depend solely on other subjective non-automated types of acceptance, such as client satisfaction with the user interface.

3.3 Maintenance Activities

Despite the fact that closing an iteration completes development of working software features defined by its user stories, two activities force developers to come back to the code of such already “completed” user stories. Refactoring concerns code quality control and software design improvement to help cope with change. Test maintenance, in addition to strengthening software correctness, is a facet of change itself.

3.3.1 Refactoring

Merely writing the simplest code that makes acceptance tests pass in the long run can make the code messy and brittle, if developers do not “see beyond” the tests. As features are implemented, test runs for preceding user stories may start to fail and new acceptance tests become ever harder to make pass.

The solution to alleviate this is to refactor code and redesign, making extensive use of coding best practices and design patterns, which boost reuse and prepare software for change and additions. Developers must always exercise this activity, as they feel the need or following specific practices such as design review, code review, pair programming, or any other they come up with. No matter which techniques developers use, however, they must follow the basic rules of refactoring:

1. Before refactoring, all tests must pass;
2. After refactoring, all tests must pass.

3.3.2 Test Maintenance

This activity ensures the quality of acceptance tests, making them more understandable and accurate. This is accomplished by:

- Finding additional relevant test cases;
- Removing irrelevant/repeated test cases;
- Improving test documentation (in the form of intra-script comments);
- Organizing tests.

Test maintenance can be viewed as a refactoring of the acceptance tests, with the exception that it also involves the client, who not only helps providing new test cases or exemplifying business rules, but also reviews the tests cases created by developers or clarifies their doubts, generally intermediated by a tester, analyst or quality assurance person.

A major risk for the project occurs when broken acceptance tests (those that don’t reflect correctly business rules) start to appear in the test database. Thus, it is extremely important that developers always do three things to cope with this risk:

1. Question tests they think are wrong;
2. Suggest what they think might be the correct test to the client;
3. Only incorporate suggested tests to the test database after the client’s review.

Paraphrasing the famous statement of extreme testing supporters: “untested code is nonexistent code”, in the realm of acceptance testing, one may say: “acceptance test not reviewed by the client is a nonexistent acceptance test.”

4. EVALUATING EASYACCEPT

Subjective impression from the use of EasyAccept in software development projects undertaken at Universidade Federal de Campina Grande (UFCG), Brazil, where the tool was developed and studied, suggests that it helps create much better quality software than prior to its creation and use.

In order to assess the validity of these impressions, we have followed three groups of software projects developed in a regular software design course of an undergraduate Computer Science curriculum. In this course, students are taught object-oriented design, coding and testing best practices, but analysis – which we believe requires more maturity – is deferred to a later stage in the curriculum. Students are also assigned a typical undergraduate software project to be completed in one semester. Since analysis is taught in a later course, a full analysis model is provided to students. Students form teams of at most three students, and all teams are assigned the same project. Projects are reasonably large (for students) and typically involve several thousand lines of code.

The projects studied were taken from classes (for the same design course) given over several semesters. Although they were not assigned the same project from semester to semester, due to the
risk of student cheating, the assignments were similar in complexity. In the first group analyzed (in which EasyAccept was not used), the project’s requirements were given to the students in the form of an analysis model consisting of textual user stories, glossary and conceptual model. Additional student questions concerning the analysis were answered through email. In the second group, the students were provided with the same artifacts but were also provided with executable requirements in the form of EasyAccept scripted acceptance tests. The first group comprised a single class, whereas in the second group we included two classes, each with a different project assigned.

For the projects of the first class, we did a retrospective evaluation of software correctness using EasyAccept acceptance tests that were created a posteriori. To comply with the script language, we adapted each team’s project with a Façade that merely accessed the business logic, so that the tests could run. No original code or design was modified. In this way, we tested the software projects and obtained the percentage of requirements not complied with. For the second group of projects, the percentage of requirements not complied with was calculated by running test scripts, and no further adaptations were necessary.

First Project: Snake Games

In this assignment, students were asked to create a framework for snake games (typically found in cell phones) in an evolutionary fashion. Three milestones were used to create several snake games of increasing complexity, so that the students’ design skills, especially in producing a working framework, could be assessed. Students were given a simple textual description containing the requirements for each game.

Second Project: Direct Mail

In this project, students were asked to create a direct mail system. A record for an enterprise’s potential client must be adaptable so that new attributes can easily be created or removed. Mailed documents are generated from templates stored in specific folders. During this semester, the analysis artifacts included scripted acceptance tests encompassing the user stories for the system. Thus, the students themselves had to interface their programs to EasyAccept to enable the automatic execution of scripted tests, and no further adaptation was necessary. Thus, the percentage of acceptance tests that passed could be directly verified.

Third Project: Monopoly

This project was assigned to a third class, using the same method as the second project, i.e., automated acceptance tests were provided before development began. Students had to create a full-fledged Monopoly game. Its complexity resembles to a great extent the one expected in typical (albeit small) IT programs, involving detailed and complex business rules prone to be overlooked, resulting in bugs. The full set of tests for this project, including sample source code, can be found in [11].

4.1 Evaluation

For the first project, an average compliance rate – measured as the percentage of user stories delivered with all acceptance tests passing – of 71.4% was obtained, while for the second, average compliance was 97.9%. The third project, which used the same method as the second, achieved 98.6%. Table 1 contains more detail regarding the projects’ numbers.

<table>
<thead>
<tr>
<th>Project Name</th>
<th># of Student Projects</th>
<th># of User Stories</th>
<th>Average Tests Per User Story</th>
<th>Compliance Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snake Games</td>
<td>6</td>
<td>14</td>
<td>5.07</td>
<td>71.43</td>
</tr>
<tr>
<td>Direct Mail</td>
<td>12</td>
<td>5</td>
<td>12.80</td>
<td>97.92</td>
</tr>
<tr>
<td>Monopoly</td>
<td>10</td>
<td>12</td>
<td>19.17</td>
<td>98.62</td>
</tr>
</tbody>
</table>

The results speak for themselves, but some considerations are necessary in order to validate the results.

Although the projects studied were taken from different classes, students in all three classes were equivalent in terms of skills, capabilities and experience, as they came from the same undergraduate curriculum and had comparable backgrounds, since the software design course they were enrolled into was given at the same point in the Computer Science curriculum (4th semester). Moreover, the projects were similar in complexity, and the classes were taught by the same professor.

Fewer projects were evaluated in the first group because, for some projects, building the Façade to interface to the EasyAccept tool would require large modifications to the inner mechanisms of the program, because of excessive code coupling (e.g., logic embedded in the GUI); since this would introduce a lot of observer noise that could influence the results, those projects were excluded from the evaluation. However, judging from observation of executing those programs and from the fact that they had far fewer tests, we feel that they were among the worst in quality and would achieve even worse results if fully tested.

5. CONCLUSION

We found EasyAccept to be an interesting tool to produce better quality software. As more features are added to it in the future, such as additional built-in commands and a full-fledged testing framework integrated with development tools like Eclipse, it can become even more beneficial to software development.

Getting untrained developers (our software design students), who regularly produced buggy software delivered late, to produce correct software delivered on time, is a notable accomplishment by itself, even more so if the approach can also be used to guide development and promote more automated project control. We believe one immediate scenario in which this approach might be very beneficial is when the developers’ quality is doubtful, e.g. in offshore outsourcing. We found that EasyAccept is particularly suitable to teach software development. Teachers and project course stakeholders assuming the role of clients can provide students with analysis artifacts, improving their development skills while facilitating their own process of project evaluation. Conversely, analysis and test courses can use EasyAccept tests as a medium to imbue the students with good testing practices.

Ideas for future work involving EasyAccept include a comparison to other ATDD approaches, particularly those that use different
formats of acceptance tests, like FiT tables; such a comparison would allow us to abstract away which ATDD patterns and techniques are tool-dependent and which are general, improving the state-of-the-art of acceptance testing.

6. REFERENCES


