Experience Report: using EasyAccept to drive development of software for an energy company

Osório Abath Neto, Eloi Rocha Neto, Jacques Sauvé

Systems and Computing Department - Federal University of Campina Grande (UFCG)
Av. Aprígio Veloso, 882 – Bodocongó. Caixa Postal: 10.106
58.109-970 - Campina Grande – PB
{osorinho,eloi.rocha}@gmail.com, jacques@dsc.ufcg.edu.br

Abstract. This paper evaluates the usage of EasyAccept, a scripted acceptance testing tool, to create from scratch an industry application via acceptance test-driven development (ATDD), a technique that automates software development to some extent. The application was a real-time decision support tool for failures in power system networks, contracted by CHESF, one of Brazil’s largest generation and transmission utility. Our experience led us to some interesting insights involving the roles of analysts and programmers in software development.

1. Introduction

Acceptance test-driven development (ATDD) [Sauvé 2006, Abath 2005] is an approach that uses executable client-readable acceptance tests written in the form of scripts as the key analysis artifacts. The focus is on creating acceptance tests to represent requirements, rather than traditional analysis artifacts like texts and diagrams. As executable artifacts, acceptance tests help to automate software development – they can be used to verify software correctness in a regressive manner as software is developed and thus guide development.

The approach presents a number of benefits, which include an effective bridging of communication gaps between clients and developers, synchronization between changes in requirements and the code written, a boost of confidence on the software that is being developed and automatically enforced focus on the client’s interests, preventing feature creep.

It differs from test-driven development in that acceptance tests are a concern of the client and thus should only involve matters that make sense to him and over which he can discuss: requirements and business rules. For that very reason, acceptance tests must be written in a format that clients – generally non-technical people that don’t have a programming background – can readily understand.

Over the years, a number of acceptance testing tools that focus on understandability by clients have emerged. The most widely known of these tools is FiT (Framework for Integrated Testing), which uses tables in HTML files to represent acceptance tests [Mugridge 2004, Mugridge 2005]. Most other tools, including EasyAccept [Sauvé 2006, Sauvé 2005] and Exactor [Exactor 2006] use a scripted text-driven approach, in which the tests are described using a format that resembles natural language – mostly, a sequence of sentences with verbs and nouns that serve as script commands.
2. Overview of EasyAccept

EasyAccept [Sauvé 2006] is an acceptance testing tool that can be used to test existing software or to create software from its inception, if you use an ATDD approach. Currently, it is available for Java programs. The tool works as follows (see Fig. 1 below):

![EasyAccept diagram]

Figure 1 – How EasyAccept works

The user provides EasyAccept with a number of text files and a Façade – a single entry point to the business logic of the software under test. Façade is a design pattern highly regarded as an industry best practice – it may even already be part of the system’s design for other purposes. The text files contain the tests in the form of sequences of commands that resemble natural language (See Box 1 below for an example). These commands match the signature of methods defined in the Façade. Via Java reflection, EasyAccept accesses the business logic of the program being tested and compares actual results of the program’s execution with expected results defined in the scripts. Divergences found in the script are reported back to the user.

The advantages EasyAccept presents over other acceptance testing tools suitable for client’s understandability [Mugridge 2004, Exactor 2006] are the short learning curve associated with its usage, the low overhead it imposes on the developers’ side (they only have to code the Façade to allow software testing) and the simple design of the testing framework that can be easily extended to incorporate new features (the tool is open source).

3. Overview of the application

Smart Action [Sauvé 2007] is a real-time decision support tool to assist control room operators during restoration procedures in a power system. The tool aids control room operators by suggesting restoration plans or executing them automatically. A restoration plan is represented by a set of steps that must be followed in order to restore a disturbance in the power system network.

Smart Action receives high-level diagnostic data from an alarm processor and can automatically initiate the execution of a restoration plan. The tool offers restoration steps
for the operator to choose from, executes and audits all steps performed. The tool is currently undergoing experimental use in a control room at CHESF, one of Brazil’s largest generation and transmission utility.

There are two types of steps in a restoration plan:

- **Viewable steps** are used to present an action to be executed by the operator. Examples: an auditable step, which checks whether the step has been performed or not and informs this fact to the operator, an information step (not auditable) and a question step (ask the operator a question and show its possible answers).

- **Control-flow commands** (non-viewable steps) are commands used to control the execution flow. Examples: for command, wait/notify command (used to wait and notify a thread of a plan - all plans are concurrent), choice command and wait conditions.

The tests of restoration plans with EasyAccept consist in executing a restoration plan, simulating the commands performed by the operators and verifying if they were executed as expected. Box 1 shows a script that illustrates a very simple restoration plan test. The plan consists in two viewable steps: “OPEN DEVICE 14T2-PEN” and “CLOSE DEVICE 14T1-PEN”.

```
1  # creates an instance of the plan pl
2  instance=instanciatePlan idPlan="p1"
3  # verifies if the plan is not being executed
4  expect "notIniciated" getStatus id=${instance}
5  executePlan id=${instance}
6  # verifies if the plan is executing
7  expect "executing" getStatus id=${instance}
8  # verifies the first step of the plan (OPEN DEVICE 14T2-PEN)
9  expect "OPEN DEVICE 14T2-PEN" getCurrentStepInformation id=${instance}
10 # changes the status of the device in the power network
11 setDeviceStatus device="14T2-PEN" status="opened"
12 # simulating the operator click confirming the conclusion of the step
13 confirmCurrentStep id=${instance}
14 # verifies if the step was finished successfully (it occurs when the device is opened after an opening step)
15 expect "positive" getStatusLastStep id=${instance}
16 expect "CLOSE DEVICE 14T1-PEN" getCurrentStepInformation id=${instance}
17 setDeviceStatus device="14T1-PEN" status="closed"
18 confirmCurrentStep id=${instance}
19 # verifies if the plan has finished
20 expect "negative" getStatusLastStep id=${instance}
21 expect "finished" getStatus id=${instance}
```

Box 1 – Example of a test script for Smart Action
4. Discussion of the usage of ATDD with EasyAccept to develop Smart Action

Before assessing the usage of ATDD with EasyAccept in the project, we need to explain some of the project’s characteristics. Smart Action is a project being developed at Universidade Federal de Campina Grande (UFCG) under a contract by the electrical power generation and distribution company CHESF.

It involves a team of eight people: one project manager, two analysts, four programmers and one client (a specialist of the problem domain). The client is an engineer from CHESF who does not have technical background in software programming. Thus, he is not able to write tests himself. Moreover, he doesn’t have enough time to sit down and write tests even if he could. Thus, analysts write the acceptance tests after several rounds of talking with the client.

The Smart Action project has been using ATDD with EasyAccept for more than eighteen months. During this period, tests were very important as a means to describe user stories themselves (not only testing) and were also used to drive development. Figure 2 presents the evolution of the number of tests during the project (each command in a test script is counted as a test in the chart). Note that between the months 11/06 and 01/07, the number of user stories didn’t increase as a result of code refactoring being done.

![Fig. 2 – Number of tests in the Smart Action project](image)

These were the benefits we have observed from the application of ATDD and EasyAccept:

- We have subjectively perceived a steep reduction in communication problems between analysts and developers compared to similar projects done in the past;
- We had a subjective perception of a substantial quality improvement compared with others systems developed by the same team (even though we didn’t make hard data...
comparisons). Fewer bugs were discovered when the system was placed in production, and most of them were related with the GUI;

- 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; this gave confidence on the code being produced and reduced fear of modifying and refactoring it;
- Acceptance tests functioned as a precise medium of knowing when the implementation of a given user story was completed;
- Development is focused in making tests pass, which avoids feature creep (developers creating unrequested features);
- Although tests don't cover GUI operations, they simulate well the actions performed by operators (commands in test scripts simulate operators pressing buttons);

On the other hand, we also identified some problems in the usage of the approach, which we list below:

- Sometimes a given user story was implemented differently from what was expected, and we believe this resulted from unclear scripts (i.e., there is still room for misinterpretation in ATDD if the scripts are not well written);
- Some test scripts were written with errors or didn't cover all possible inputs;
- In critical time situations, the development team changed some test scripts with problems in order to finish the release (even though this is a condemned practice in ATDD: test changes must be reviewed by the client);
- Tests are not concurrent, so it is not easy to simulate and test concurrent situations;
- Tests don't cover the graphical interface, only the business logic.

In addition to these benefits and shortcuts, our experience with ATDD and EasyAccept led us to an important insight involving the dichotomy between the roles of analysts and programmers, which we detail in the following paragraphs:

Communication is one of the main sources of errors in software development. As the number of communication transfers increases (clients to analysts to programmers), the odds that gaps in these transfers occur also increases. In order to reduce the effect of communication gaps in software development, the direction that has been historically given to software development was that of reducing more and more the separation of roles between analysts and programmers. Analysts and programmers had formerly well-defined non-overlapping roles. Analysts talked to the clients to capture requirements and handed them to programmers, who created working software. People concluded that blurring the two roles into a single common “developer” denomination, with mixed responsibilities, allowed less information to be transferred and thus less miscommunication would arise. However, it is hard to find professionals that excel in both roles. The profile of a typical analyst is that of an expansive, talkative professional specialized in negotiating with clients and getting the most out his communication skills to capture software requirements. On the
other hand, programmers tend to be technically-oriented, more comfortable with machines than with people.

We think (and observed this in practice) that, with ATDD, an efficient separation of roles between analysts and developers can be finally established, as executable analysis allows requirements to be captured unequivocally. Analysts can capture requirements efficiently and translate them into executable requirements (acceptance tests), even if they don’t have expertise in a programming language. On the other hand, programmers need not have as much direct contact with the client as analysts and simply create the code that meets the executable requirements.

Another insight on the usage of ATDD that relates to this role separation is that it may be the ideal approach for developing outsourced code. An enterprise can specialize on having analysts write acceptance tests with the desired level of quality and simply use these tests as requirement artifacts for subcontracted external companies that will create the working code.

5. Conclusion

We found that the usage of ATDD and EasyAccept was an important decision for the development of the Smart Action project, and that the benefits outnumber the limitations. Driving development with acceptance tests enabled us to produce working code even using undergraduate students as programmers. We are committed to keep using the approach in future projects.

Although communication problems between analysts and programmers have reduced drastically, we recommend that programmers themselves (not just analysts) must have a discussion with the client about the user story before its implementation; if the test script is not clear or has errors, it must be rewritten by the analysts.

Ideas for future work involve making hard data comparisons between projects our team has done in the past without using EasyAccept and ATDD with Smart Action project, in order to put in numbers what we have subjectively perceived.

References


