# The Perceived Value of Authoring and Automating Acceptance Tests Using a Model Driven Development Toolset

David Talby
*Hebrew University of Jerusalem*
*davidt@cs.huji.ac.il*

## Abstract

*One approach to applying keyword driven testing in a model-driven development environment is by defining a domain specific language for test cases. The toolset then provides test editors, versioning, validation, reporting and hyperlinks across models – in addition to enabling automated test execution. This case study evaluates the effectiveness of such a solution as perceived by two teams of professional testers, who used it to test several products over a two year period. The results suggest that in addition to the expected benefits of automation, the solution reduces the time and effort required to write tests, maintain tests and plan the test authoring and execution efforts – at the expense of requiring longer training and a higher bar for recruiting testers.*

## 1. Introduction

Keyword driven testing, also known as action word testing, dictates a planning stage before starting to write test cases, in which a set of abstract keywords about the application under test is identified [5]. Figure 1 illustrates a test case for a form-based application – "Edit Field" is an example of a keyword, a.k.a. action word. Once the keywords are defined, test cases can be written by non-programmers, and a driver to automatically execute tests can be developed. This approach rewards the initial planning effort by making tests more readable, reusable and easier to maintain, compared to script-based automated testing [1].

Defining the action words, their parameters and semantics for a given application domain is essentially defining a domain specific language (DSL) for test cases. DSL's are a popular approach to model-driven software development (MDSD), in which a planning stage defines an explicit meta-model, or language, for each aspect for an application domain – data entities, transactions, integrations, rules, forms and so on [7]. Tools enable developers to work with models as first-class artifacts, and to automatically transform them into executable code or configuration.

As DSL tools such as [2,6] mature, there are potential advantages to using them to implement keyword driven testing instead of specialized test tools like [3,4]. Both types of tools enable the definition of action words for a given domain, and then generate a UI to edit test cases, support versioning, custom validation, reporting and integration with the build process. However, since a DSL tool can be used to edit multiple models in a given application – with test cases being just one kind of model – it can provide unique features which are not otherwise possible.

This industry case study presents testers' evaluation of such a solution and the benefits it delivers.

## 2. Background

The toolset this paper evaluates is described in detail in [9]. It had four unique features taking advantages of its generic and integrated meta-modeling environment.

The first is auto-completion while editing tests. Since all the data entities and their fields, actions and forms are modeled using the same toolset and metadata repository [8], then the first parameter is each of the rows in Figure 1 can be filled from a combo box, with auto-completion, based on the context. This speeds up test authoring and prevents the common case of test failures due to spelling mistakes. In addition, each reference to another model is a hyperlink, making it easier for a tester to navigate and learn models.

| Step Type | 1st Parameter | 2nd Parameter |
|---|---|---|
| **Open Form** | Account | |
| **Edit Field** | Customer | **John Doe** |
| **Check Field** | Amount | **0.0** |
| **Edit Field** | Amount | **1000.00** |
| **Check Action** | Withdraw | **Enabled** |

**Figure 1: Keyword Driven Test Case Example**

The second feature is validation. Before they are checked in, test cases can be validated against the current baseline of the other (non-test) models, to make sure they don't reference models that do not exist.

The third feature is impact analysis. In the case study organization, a new baseline of detailed specifications would be delivered every other month, which caused testers – before this solution was implemented – to spend up to a quarter of their time figuring out which test cases must be created or updated. The integrated metadata repository enabled quick, automated answers to questions such as "show all test cases which refer to field X of entity Y" or "show all changes in entity Z between the current and the previous baseline".

The fourth feature is detailed coverage reports. Once test cases have formal, validated hyperlinks to every other model they use, it becomes easy to ask detailed coverage questions: Show all actions which are never executed in any test, or show all UI fields which have an 'Is Enabled' logic specified but no test which checks whether this field is enabled. Beyond metrics for management, such reports also provide direct guidance to testers on what needs to be improved.

All of the above are potential benefits – this case study presents how the testers who actually used this solution in a real-world setting evaluated it after extended use. Since the organization under research was an early adopter in applying this approach for a suite of large-scale enterprise applications, to the best of our knowledge this dataset is unique at this time. The next section describes the methodology for gathering the research data, and the following sections present and analyze the results.

As a clarifying note – this case study is *not* about model based testing [10], which focuses on automatically generating test cases from models. It's about applying "traditional" MDSD to empower human testers.

## 3. Methodology

Data was collected in two ways. The first is a set of questionnaires filled by nine professional testers who used the toolset extensively on a daily basis. The nine testers had an average experience of fifteen months of using the toolset; six months of hands-on experience was the minimum for a tester to be included in this case study. Six testers had previous experience with other testing tools. All nine testers worked in the same organization, using the same methodology over the same two year period, so different responses should not be attributed to organizational factors.

The questionnaires were anonymous, and contained a mix of open and closed questions. Aggregated results for some of the closed questions are presented in the following sections, and in all cases include all nine responses. Areas in which similar questions were answered inconsistently, as well as questions whose answers included extreme outliers, are excluded from the discussion below. Therefore the results presented focus on the significant and consistent findings – although this sample size is too small for rigorous statistical analysis of significance.

The second form of data collection for this case study was in-depth interviews – in particular with the two test team leads, who were responsible for planning, tracking, training and overall effectiveness of the test effort, in addition to writing and executing tests. Together with the open questions on the questionnaire, these interviews enable interpretation and validation of the quantitative results.

## 4. Authoring Tests

Table 1 summarizes results for two key questions on the time required to write and maintain tests using the model-driven toolset. This is individual work so time represents effort as well. The numbers are the average responses – i.e. "10% longer", "10% shorter" and "15% longer" would average to "5% longer".

The first result is surprising – writing formal tests is usually labor intensive, due to the need to conform to machine-readable syntax and specify details that are often – for better or worse – implied by free-form text. Consider for example Figure 1 versus this text:

*"Open a new account for 'John Doe', change the amount from 0 to 1000 and verify that 'Withdraw' is enabled".*

The interviews and open questions explain the fact that test writing in this toolset takes less time than writing free text by the effectiveness of auto-completion in the editor. This is verified by another closed question whose result stated that test writing time with auto-completion is **40% shorter** (averaged) than test writing time without it. Step names, field names, actions and

| **Compared to writing free-form test cases for manual testing, how long does it take to …** |
|---|
| Write a test case? |
| **12% shorter** |
| Update test cases when specs change? |
| **40% shorter** |

**Table 1: Writing and maintaining test cases**

| Compared to executing manual tests, how long does it take to … |
|---|
| Execute a single test? |
| **48% shorter** |
| Execute the entire test suite? |
| **67% shorter** |
| Reproduce an application bug? |
| **49% shorter** |
| Decide if a failed test is a new or existing issue? |
| **30% shorter** |

**Table 2: Executing tests**

| Compared to an environment in which tests are written and executed manually, evaluate the … |
|---|
| Ease of estimating the time required to write tests |
| **8 of 9: Easier** |
| Ease of estimating the time required to execute tests |
| **7 of 9: Easier** |
| Required professional level of a tester |
| **7 of 9: Higher** |
| Time required to train a new tester |
| **12% longer** |

**Table 3: Managing the testing effort**

choices (such as enabled/disabled) are selected from combo boxes or via auto-completion – so after several weeks on the job a tester could write a test step in just a few keystrokes. In addition, testers do not have to memorize or look up exact names.

The time and effort to maintain tests as specifications change is another big win, perceived on average to be 40%. This is attributed to the ability to compare models to their previous versions – immediately see what changed – and then find all the test cases that refer to changed, renamed or deleted specs. This replaces the tedious work of manually going through all test cases to find these occurrences, which also heavily relied on the experience of the specific tester doing that work.

## 5. Executing Tests

Table 2 summarizes the main results on test execution. This does not entail just clicking the 'run' button and watching an automated test run, but the entire process of testing a new version of the application:

- Setting up an execution environment
- Running the automated tests
- Analyzing each failed test
- Fixing the tests where necessary and re-running
- Interacting with developers and other testers to find if failures are new or existing defects
- Describing defects well in the bug tracking tool

Having an automated solution is perceived to eliminate roughly half the time it take to execute a single test, and two thirds of the time required to execute the full suite. The interviews suggest that this is largely attributed to test automation, meaning that these improvements are not specific to an MDSD based solution, and are comparable to what other capable test automation toolsets would provide.

This reasoning also applies to the reduction in the time it takes to reproduce a bug – although on that subject the interview feedback stressed the importance of interactive, step-by-step test execution from the model editor, rather than the ability to run tests in batches.

## 6. Managing the Testing Effort

Table 3 summarizes results on planning, estimation, recruiting and training. For the first three questions only a generic comparison was requested (more/less), because only two of the nine subjects were experienced team leads who performed these tasks regularly. Training was delegated across the team and most people had some experience training others, in addition to how they were personally trained.

A majority of people stated that it was easier to estimate the time required to both write and run tests using the MDSD based solution. In interviews, both team leads strongly agreed that this was true, resulting in a much more predictable test team. Two explanations were given for this benefit.

The first is that in a model-driven environment, the detailed specifications – against which tests are written – are formal and validated, enabling a better estimate of their complexity in advance. For example, if a new version of the specifications added or changed 11 transactions, 38 fields in 7 entities and 15 UI actions – the test team leads could rely on these numbers to estimate how long it would take to test them. Over time they developed metrics as specific as averaging how many minutes it takes to test a single field in the UI. This was widely perceived as being easier and more accurate than trying to estimate the complexity hidden in a 20-page free-text functional spec.

The second explanation for easier planning is that the formal test cases are clearer than free-form tests,

enabling more flexibility in assigning test execution to different people without losing predictability. Consider for example the test case in Figure 1 – it can be executed as quickly by someone who was hired just four weeks ago and by the domain expert on opening accounts who has two years of experience. This is often not the case with free-text test cases, which often assume both domain and application knowledge (Which dialog box is used to change the account type? Which types of users are authorized to do so?).

One of the benefits of keyword driven testing is that the number of steps per hour that a tester can execute can be measured and remains predictable over time. There's also more flexibility in planning – anyone can execute (or maintain) any part of the test suite, at roughly the same speed and quality.

The last two questions in Table 3 suggest that a drawback of the solution studied here is that it is more complex than a manual testing environment. While the perception that it requires more talented testers may be the result of the testers testifying on their own skills, this would not affect the perceived training time. The interviews confirmed that training takes longer in this environment, although new testers were still expected to become fully productive within a few weeks.

Experienced testers and the test team leads did not consider this to be a major problem, since extra training is required to use any automated testing tool. The MDSD based solution was perceived as adding a relatively small overhead in this respect, similar to other keyword driven testing tools, and better than script-based tools which required programming and familiarity with tool-specific APIs.

## 7. Overall Value and Summary

Table 4 summarizes the perceived bottom-line value of the solution. Cutting time to market by two thirds compared to a manual testing environment is the primary perceived benefit. The open questions explain it as the cumulative result of the highly efficient tools and processes to maintain the test suites, execute tests, and work with developers to reproduce defects. The team leads added predictability and flexibility in task assignments as another factor.

The perceived reduction in the number of defects that go undetected was explained simply by the more effective use of the testers' time: Since testing is always both time- and resource-constrained, substantial gains in tester productivity result in a higher quality product.

These results indicate that the professionals using this solution on a daily basis value it as highly effective and as a step forward from traditional test automation

| **Compared to an environment in which tests are written and executed manually, what is the …** |
|---|
| Time from a code complete to a delivered product? |
| **63% shorter** |
| Number of defects that go undetected? |
| **34% smaller** |

**Table 4: Overall value**

tools – which as detailed in [9] this solution replaced. In particular, modeling test cases inside an integrated DSL environment and metadata repository enables several benefits that were not available before.

This case study covers two teams of experienced testers working for two years on several real-world, enterprise-critical applications. Since they were early adopters of this approach, this is an early and limited data set – future work is required to expand it to more organizations, projects and people, as well as to validate perceived benefits against measured results.

## 8. References


[1] Buwalda H., Janssen D., Pinkster I. and Watters P., *Integrated Test Design and Automation: Using the Testframe Method*, Addison-Wesley 2001.

[2] Cook S., Jones G., Kent S. and Wills A. C., *Domain-Specific Development with Visual Studio DSL Tools*, Addison-Wesley 2007.

[3] FIT Acceptance Testing Framework, http://fit.c2.com

[4] IBM Rational Functional Tester Plus, http://www-01.ibm.com/software/awdtools/tester/functional/plus/index.html

[5] LogiGear, Action Based Testing, http://www.logigear.com/test_automation/action-based-testing.asp

[6] MetaCase, http://www.metacase.com/

[7] Stahl T. and Voelter M., *Model-Driven Software Development: Technology, Engineering, Management*, Wiley 2006.

[8] Talby D., Adler D., Kedem Y., Nakar O., Danon N. and Keren A., "The Design and Implementation of a Metadata Repository". In Proc. of Intl. Council on Systems Engineering Israeli chapter conf., March 2002.

[9] Talby D., Nakar O., Shmueli N., Margolin E. and Keren A., "A Process-Complete Automatic Acceptance Testing Framework". In Proc. of 2005 IEEE Intl. Conference on Software - Science, Technology and Engineering (SwSTE '05), February 2005.

[10] Utting M. and Legeard B., *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufmann 2006.