A Process-Complete Automatic Acceptance Testing Framework

David Talby, Ori Nakar, Noam Shmueli, Eli Margolin, Arie Keren
Mamdas – Israeli Air Force
Email: davidt@cs.huji.ac.il

Abstract

We present a new automated software acceptance tests framework. The framework is novel in supporting the entire lifecycle and all QA activities, including test maintenance over multiple versions, interaction with programmers and business analysts, traceability to specifications, multi-user test cases and more. This enables a significant increate in QA productivity and product quality. We compare our framework to other available tools, products and frameworks, and present several patterns and anti-patterns for implementing a successful automated acceptance testing solution.

1. Introduction

An acceptance test is a formal test conducted to determine whether or not a system satisfies its *acceptance criteria* – from a user's point of view – and to enable the customer to determine whether or not to accept the system [17]. In an Extreme Programming project, the acceptance tests are written and owned by the customer. In conventional developments methodologies, these tests are usually written by the project's QA team.

Acceptance tests contribute three things to a software project [11]. First, they capture the system's requirements in a directly verifiable way. As one study [8] shows, typical requirement specifications are 15% complete and 7% correct. There is strong indication that exhaustive requirements specifications are impossible. And even if they were possible, the only way to verify them would be to translate them into test cases. Acceptance tests address both of these issues: they can grow as the system grows, and they capture requirements in a directly verifiable way – if the test passes, the requirement it documents works.

Second, acceptance tests expose problems that other type of technically-oriented tests miss. As described in [3], acceptance tests capture many bugs that unit tests don't, even if the unit tests provide full code coverage. Stress tests and system integration tests also do not target the type of end-to-end functionality that acceptance tests do.

Third, acceptance tests provide a ready-made definition of how "done" the system is. A system is deliverable exactly when it passes all its acceptance tests, so the percentage of such tests that pass is the only practical definition of real progress. No other measure – percentage of code written, invested time, used resources and so on – makes sense as a bottom-line indicator of completeness.

For the above reasons, most software development projects have some procedure for acceptance testing, sometimes the responsibility of specialized QA teams. Whoever fulfills the QA role in the project receives the software when it is feature-complete, run the acceptance tests, and report errors back to the developers. The ongoing responsibilities of the QA role also include writing acceptance tests (which requires close interaction with business analysts), helping developers track down bugs, and maintaining acceptance tests as the software evolves.

Automation of QA activities is highly desirable for two reasons. First, many of the tasks are mundane – such as filling forms and validating their results – and highly repetitive – since all edge cases should be tested. Second, running all acceptance tests is a bottleneck before product delivery, and can take weeks for a medium-size project. This affects both the time required to find all bugs, and creates an overwhelming overhead for each delivery. This is why agile software methodologies often consider automated acceptance tests a must-have [1, 2]. Automated tests can also be run more often and earlier in the development process, thereby improving product quality [11,12].

A large variety of tools, frameworks, products and techniques for automating acceptance tests is available. However, there is mounting evidence that developing your own solution is often the more effective option, and that this is not as daunting a task as it seems at first [9,12]. Most of them are focused on writing the

tests in a formal yet user-friendly language, running the tests in a configurable manner, and providing useful reports of the results [4, 5, 6, 10]. This gives a solution for the pre-delivery bottleneck, but not for the many other QA activities in a software project, which may take well over half of the time of QA personnel:

- Interacting with developers to reproduce bugs
- Interacting with business analysts to understand detailed requirements, to ensure full coverage
- Deciding whether a failed test is caused by a bug in the test or in the product
- Deciding if a failed test is a new or known bug
- Tracking coverage and traceability to specifications
- Finding which tests need to be changed, when the requirements of a new version/milestone of the product is written (Impact Analysis)
- Managing multiple concurrent versions of test suites, for multiple versions of the tested product
- Defining and maintaining common portions of test scenarios (edge cases, setup, etc.)

This paper presents a new framework that builds on known best practices, but takes a leap forward, and deals with the entire set of QA activities and responsibilities. The end result positively affects all of the above tasks. Moreover, the framework is being used in a real-world, large-scale enterprise information system project. Both the framework and the development process that it implements have been "forged by fire" in real use, over a considerable amount and scope of written acceptance tests.

This paper is structured as follows. The next section describes the setting of the tested project. The next two sections describe the framework's tools for writing tests, and for running them. Section 5 describes "tricks of the trade" – solutions we implemented for common acceptance-automation problems. The final section summarizes our insights and advice – for building your own automated acceptance testing solution.

2. The Setting

2.1. The Testing Environment

The tested product is a large enterprise information system. The system has several hundred forms and tables, each with its own set of fields, actions and transactions. The system also has several dozen user types, each with its own permission rules, and numerous on-line interfaces to other systems.

There are two preconditions to building any automated acceptance testing solution. The first is the existence of a small, common set of metaphors that describes the system's UI. This enables us to express all kinds of user actions on the system using a relatively small number of verbs [7,11]. In our case, the system is built on top of an in-house object-oriented framework, which handles all the technical aspects of the web and application server layers, as well as providing a standard user interface. The UI framework is based on standard metaphors such as "Window", "Selection", "Action" and so forth.

The second precondition to an automated acceptance test solution is a tool that manages the system's specifications, and allows querying them. Otherwise – for example if specifications are described in free-text documents – it is obviously impossible to track traceability, coverage, find specific changes between versions, or determine whether a failed test is caused by a bug or by a miss-match between the test and its specification. To enable automation, it is also important that the specifications tool provide an API for other programs (not just humans) to query it. We solved this problem by using the same tool as both the specifications and the tests repository.

The entire system's detailed functional specifications – data entities, forms, tables, actions, permissions and so forth – are written by the business analysts in a tool called the metadata repository [16]. This is a highly configurable tool, which in essence enables the definition of document schemas, and then the editing of documents of these schemas. The documents themselves are stored as XML files on the file system, which is vital because it enables managing the detailed functional specifications using the same configuration control tool used for storing source code.

The metadata repository is a bridge between business analysts and programmers. On one hand, it has a visual editor for business analysts to edit their work, using their own vocabulary; for example, they define an "Entity", and not database tables, application server services or classes. On the other hand, it has a template language and generator, used to automatically generate source code files, database and server initialization scripts, form layouts, reports and so on. The detailed specifications which are formal enough are generated into code directly, while the more complicated business logic has to be manually coded.

For a complete description of our metadata repository, see [16]. For the purpose of this paper, it suffices to know these three of its prominent features:

- Can create arbitrary document schemas, and edit documents of these schemas
- Stores each document in a file, rather than a database
- Has a built-in concept of hyperlink between documents, and supports a query of all hyperlinks from or to a given document

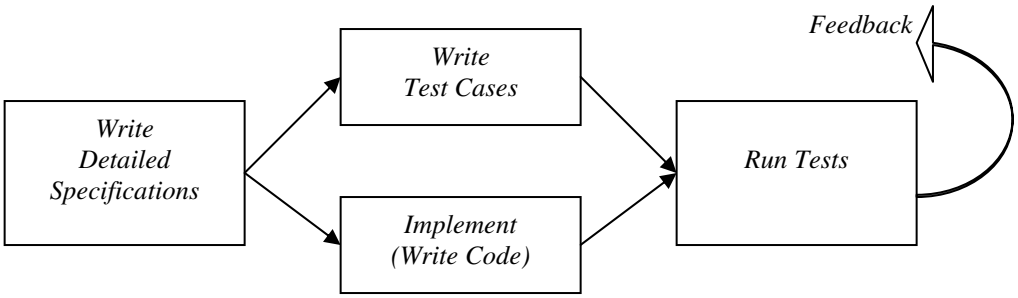

Figure 1: Testing Process

2.2. The Testing Process

The testing process for each milestone of the project is sketched in figure 1.

The test framework presented in this paper was not planned in advance – in fact, when the project began two highly acclaimed commercial products for software testing were purchased (for example see products such as [13, 14]). One was used as the tests repository – it enables a tester to write and organize free-text tests, and to record the results of running manual tests. The other tool was used to write and execute automatic tests; we did not use the tool's recording capabilities, because of the maintenance costs of such tests when the UI changes, but instead used its scripting language to translate tests from the other tool (which were written there in readable non-technical form) to scripts that the tool could automatically run.

The process was as follows. When specifications for a new version were done, the QA team started to write tests for them. When the new version was ready, each new test was run manually – to validate the test itself. For the next version, it was automated by having a QA team member manually create a script to run it in the automated test execution tool. The effort of translating tests to scripts was very high; many tests were not automated. This raised the resources required to run manual tests, which kept growing each version. Also, tests on volatile entities or data were not translated to scripts, to reduce to expected maintenance cost of these scripts – but this created the paradoxical situation in which the most error-prone portions of the software were the least tested. The costs involved were so high that they justified developing a new solution, tossing the bought tools aside, and converting the existing tests to the new solution. The next section describes how we chose that solution.

3. Writing Acceptance Tests

3.1. Linking Test Cases to Specifications

Tools such as [13,14] provide an orderly way for a team of testers to write and organize a large database of test cases. At the single test level, each test is composed of a list of steps – the atomic execution unit that can either pass or fail. The difference between these tools and automated acceptance tests frameworks such as FIT [5], FAT [4] and JAccept [10], is the formalization of tests. Instead of a free-text description saying "Write 999 in the amount field and hit 'Save'", we'll define:

Step Type	*Parameter 1*	*Parameter 2*
Edit Field	Amount	999
Do Action	Save	

The key is that such a format is still readable, but formal enough to be automatically executed. Each step starts with selecting a step type (Edit Field, Do Action, etc.) – a predefined list selected from a combo box – and adds parameters to that step type. In our case, four parameters were sufficient for all step types.

There are two patterns proposed in [7] for designing the editor and test language, which we embrace and pass on:

- Provide a simple, high-level domain language. Avoid inventing a new programming language – expressions, conditions and loops are not required and will make the tests unreadable. use high-level verbs such as "Open Form - Customer" instead of low-level ones like "Call Session Bean - http://...".
- Provide a visual test editor. Testers and business analysts will resist working inside an IDE or an XML editor. They need a tool that lets them write fast, navigate and view tests in a readable fashion, and ignore low-level issues.

However, all the existing tools that we found lack one important feature – linking the tests to the specifications. The problem is that field names, action names, form names, error messages and so forth must be entered into the tool manually by the tester – and even one spelling mistake is enough to cause the test to fail. This requires the testers to be slower and more careful, and also lengthens the test execution phase since most failed tests will be caused by a bug in the test rather than in the system.

We have solved this problem by using the metadata repository as the tests repository. We defined a new document schema named "Subject Testing", which contains a list of tests for a given category; each test is a list of steps as usual. We then used the scripting capabilities of the repository, to implement combo-boxes and auto-completion for all relevant step types and parameters. This is done by using the repository's API to query the detailed specifications, also stored within it. Consider the following example:

Step Type	*Paramater 1*	*Parameter 2*
Open Form	Account	
Edit Field	Customer	John Doe
Check Field	Amount	0.0
Edit Field	Amount	-1000.0
Check Action	Withdraw	Enabled
Do Action	Request Loan	
Confirm Message	"Cannot loan on overdraft"	
Do Action	Save	
Check Action	Save	Disabled

Figure 2: Test Case Example

In this test case, all values except for the numeric amounts and customer name can be selected from combo-boxes or auto-completion boxes. This dramatically increases tester productivity. Another worthy addition was usability improvements that allowed writing multiple steps using the keyboard alone, making test writing more fluent. Values can also be verified as-you-type, decreasing the number of errors in test cases.

3.2. Automated Queries

Another important outcome of linking the test cases to the specifications is the ability to trace the link. Since we know precisely where action names (for example) are used in test cases, and where they are defined in the specifications, we can write automated queries that answer questions such as these:

- Which actions are never used in any test case? (Coverage report)
- Which tests use actions that don't exist? (Finding reason for failed steps; Finding tests that need to change when specifications change)
- Which tests will have to change if the specification of action X changes? (Impact Analysis)
- Which actions have defined business logic for when they are enabled, but there is no test step that checks whether that action is enabled or disabled? (Detailed coverage analysis)

As you can see, we can go into great detail in our queries, and automate many of the most difficult tasks in managing a large suite of test cases. Writing these queries uses the query language of the tool you use – we use the language that our metadata repository provides. In your solution, that language may be SQL or VBA or some scripting language.

Using the same tool for both specifications and test cases is one of the strongest points of our solution, and is highly recommended. However, you can use any tool – even Excel for example – for writing tests, as long as you invest the time to implement combo-boxes and auto-completion for test steps, and relevant queries and reports for the type of questions presented above.

4. Running Acceptance Tests

4.1. Architecture

The main benefit from an automated test solution comes when the product is ready to be tested. Then, as fast as possible, all tests must be run, and every failed step must be categorized as a new bug, known bug, or mistake in the test case.

Architecting an automated tests framework requires making three decisions:

1. Are tests cases generated to code first, or executed using an interpreter?
2. What is the protocol to execute steps on the tested application?
3. Where are execution results stored, and how are they reported?

Regarding the first question, some of the commercial tools we reviewed require translating test cases into code-style scripts before execution. The better approach, taken by the majority of newer frameworks, is to build an interpreter which translates the test steps into operations on the tested application. This saves the time of the code generation, and enables making minor changes to the test cases on-the-fly. The performance penalty of interpreting the tests is negligible in most kinds of tested projects.

Regarding the second question, there are significant changes between different tools. Some frameworks rely on reflection: Each step type is associated with a class and method name, and the framework uses reflection to call the appropriate method with the right parameters. This requires the tested application to be written in the same language as the framework and to be started using the framework. Other tools, particularly ones that support recording, mimic the operating system. This is the least-intrusive option, but does not work for every UI, and is very sensitive to UI change in the tested app.

Our solution is to use a remote socket. The tested application implements a server socket listening to some port; the test execution framework opens a TCP connection to that port to begin execution. The message protocol is standard XML – each message contains one step to execute (type and parameters), and is responded by the step's outcome (success or failure) and its result (if any). The framework is stateless – each step's execution is independent.

This solution gives us several unique advantages. First, the tested application need not use the same programming language or operating system as the test framework. Any platform that supports sockets will do. Second, the tested application and the test framework can run on different machines. This allows a tester to run tests on several different installations or configurations without physically going there, as long as a LAN or Internet connection is available. And third, this enables writing a test that runs on multiple computers. Here is a typical example:

Step Type	*Paramater 1*	*Parameter 2*
Select Station	First Station	
Open Form	Customer	
Edit Field	Full Name	John Doe
Edit Field	Amount	100.0
Do Action	Save	
Select Station	Second Station	
Open Query	Customers	
Select Table Row	John Doe	
Check Field	Amount	100.0

Figure 3: Multi-Station Test Case

This is a high-end feature of some commercial tools, which is very simple to implement using sockets. Since each step is executed independently, a "Select Station" step simply tells the framework where to send the next executed step. A separate configuration file maps logical station names to IP addresses and ports, to separate this technical issue from the test case.

The third architectural question to be answered is where execution results are stored, and how they are accessed. Existing tools usually store the results in a database, and provide a set of web reports to view them. The implied process is that a tester prepares an execution configuration, runs all automatic tests at once, and after they are all done reviews the reports.

We took a different approach. In our solution, the results of running each step are written back to the metadata repository, to a column in the same test that includes the executed step. To do this, we exploit the fact that the API used to query the metadata repository for tests is read-write. The test results can be written inside the test document itself – this is the mode used when running tests in interactive mode – or in another document that inherits the original one, and contains only the addition of the results.

In both cases, we enjoy several rewards. First, the user of the framework need only know one tool – the visual editor of the metadata repository – for both writing and analyzing test results. Second, we provide an equivalent interface for manual and automatic runs – in manual runs, the tester runs each step and marks whether it succeeded or failed on the same form. Third, we can use the metadata repository's powerful query language to extract reports on test run results. This is in contrast to the limited configurability of the reports of some of the existing tools. And fourth, the test results are stored in files under the same configuration control rules that are used for all documents, which makes it easy to record results of past testing cycles.

Step Type	*Param 1*	*Param 2*	*Result*
Open Form	Customer		Passed
Edit Field	Full Name	John Doe	Passed
Edit Field	Amount	100.0	Passed
Check Field	Amount	100.0	Passed
Do Action	Save		Failed
Check Action	Save	Enabled	Not Run

Figure 4: Test Execution Results

Figure 4 is an example of how results look from the framework user's perspective. The above table can be viewed using the same tool that is used for writing tests, or for manually executing tests. A step may pass or fail, and by default the test is stopped after the first failure, marking the subsequent steps as "Not Run". Another column that does not appear in the figure is the "Result Comments" column, which contains a message describing why the test failed (for example "A customer called 'John Doe' already exists"). The results column is useful for manual testing as well.

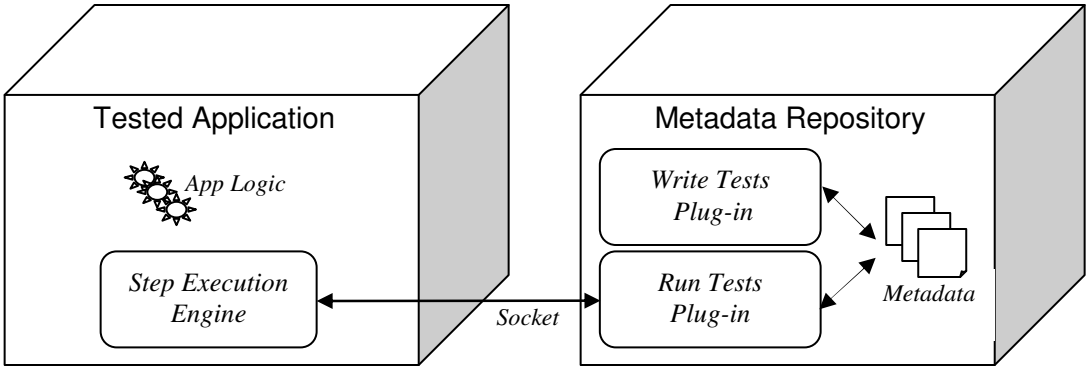

Figure 5: Framework Architecture

Figure 5 summarizes the framework's architecture, which is composed of three modules:

- A plug-in of the metadata repository, which is used to fill combo-boxes and auto-complete fields, in the test writing phase.
- Another plug-in of the metadata repository, which is the interpreter part of the test execution engine. It opens a socket to the tested application, sends it test steps, and updates the results back in the repository. This module has the important role of translating the human-oriented test definition language to the code-oriented API of the tested application, thus keeping these two languages decoupled. This enables keeping the test language readable and the app's API minimal without conflict.
- A module within the tested application, that receives requests to run steps through a socket, and knows how to execute all step kinds and report the result.

Some of the code is by nature specific to the tested application, while other parts are "true framework".

4.2. Interactive Execution

The synergy of the three architectural decisions we made enables us to provide the testers with another unique advantage, namely interactive execution of test cases from within the visual test editor.

Using the customization capabilities of the metadata repository, we added to it a menu and a toolbar that provides commands such as "Run Step", "Run Test" and a few others that we'll soon explore. To run a test in interactive mode, the tester need simply start the application, open the test case in the editor, and hit "Run Test". All steps will run until the first failed step, which will become selected. The tester can then inspect the step's result comments, the state of the application, and the specifications (which are also edited and viewed in the same tool), and decide what to do.

If the tester's decision is to change the test case, then s/he can edit it on the spot, and re-run the failed step. Since the steps are interpreted for execution, no pre-processing is required, not even restarting the run. Since the execution engine is stateless and steps are sent to the application one at a time, then it doesn't matter if steps are repeated or run out of order. And since results are reported right back to the repository, and so the test writing and running tool is the same, the tester does the whole process within the same screen.

The interactive mode of test execution – from the test case editor and not another code/script/report programmers' IDE – is unique to our framework. It plays central role in several QA use cases. One is fixing a test case on-the-fly while it's executed. Another is a mixed manual-automated execution: A tester can stop an automatic execution at some step, work manually on the application (for whatever reason), and return to running the rest of the test automatically. Another case is reproducing a bug, on request from a programmer. This is done by loading the failed test and running it until at fails. At failure, the tester and programmer can switch to the application window and examine it. Sometimes there is a need to stop at steps that pass – for example, a step passes when it's not supposed to – and for this we provide both a "Run to Selected Step" command, and the ability to mark the result of a step as "Break", which stops execution when it gets to that step, and so mimics a breakpoint.

4.3. Continuous Testing

In addition to support manual and interactive modes of test execution, our framework also provides a command-line interface. This is required for the important best practice of continuous integration: Builds of the systems should be conducted often (a nightly build is the norm), and all automatic acceptance

tests should be run on the result of that build as well. The rational is simple: automatic tests serve as regression tests, and we'd like to know that new code delivered today does not break existing behavior. If it does, we'd like to know right away, since the time required to fix the problem is probably the shortest now – when the responsible programmer has a fresh memory of the changes that were delivered. Continuous integration reduces the overall time required to fix bugs, and reduces the amount of work to do during the high-pressure pre-delivery period – because some of the tests have already run and passed.

In our case, implementing continuous testing was simple, since our metadata repository supports command-line activation of plug-ins, and since one of its predefined plug-ins can generate reports defined by its query language. To define run configurations, we created another document schema in the repository, which allows specifying the list and order of tests to run, and the computers (IP & port numbers) on which to run them. The test execution plug-in reads a given configuration and runs all its tests; results are written back in the repository. The next step is to generate reports – using the tests and the results data, both of which are inside the repository. The reports are in HTML format, and are copied to the network folder where our web server expects to find them.

Continuous integration is simple to implement, but difficult to assimilate into a project's development process. It is often tried and deserted, due to inability to distinguish between old and new bugs. This happens as follows: at first, the nightly build succeeds and all tests pass. Then, as code and tests are added, some of the tests fail. Not all tests can be fixed on the next day – this is an important distinction between unit and acceptance tests [15]. Eventually, no one looks at the night build results, because many tests fail in it regularly and nothing should be immediately done about it (these are known bugs). Sorting out new failures from the old is too time-consuming to be a daily activity, and so the project reverts to testing only before product delivery, when all failures are equally important.

We have designed a simple mechanism to eliminate this problem. We have not seen a solution to this issue in the existing testing frameworks, and strongly recommend that a mechanism of this type be a part of any automated testing solution.

The process is as follows. When a new bug is found – either during interactive execution or a continuous testing run – a tester opens the failed test case to review the problem. Sometimes the problem is not a bug (network failure, etc.) and requires no action; sometimes it is an error in the test case, in which it can be corrected on the spot; and sometimes it is a bug in the application. If (as it the common case) it can't be fixed immediately, then a new bug should be reported. However, apart from opening the bug in a separate bug tracking tool, it must also be written in the test case that this bug is known, and future runs should be able to distinguish it. To denote this, the tester simply changes the result column of this step to "Skip". Optionally, the result comment column could contain the new bug's ID. Steps marked with "Skip" will be skipped during automated execution, and not affect the results. In most cases skipping a buggy step; in some cases, the inability to execute a step means that the rest of the test case can't be run as well, and then the entire test is skipped.

Therefore, the bottom-line report of a command-line execution of multiple test cases can look like this:

Measure	*Result*
Executed Test Cases	170
Executed Test Steps	25390
Passed Test Steps	25302
Skipped Test Steps	85
Failed Test Steps	3

Figure 6: Test Run Executive Summary

This means that there are 85 known test steps that don't work, and three new potential bugs. Note that the number of skipped tests is not the number of known bugs (the bug tracking tool has that information), since several skipped tests may be caused by one bug. But we do know that failed tests indicate new failures – the new failed test should be examined the morning after the night build, and must be either corrected or marked as skip.

5. Tricks of the Trade

During the actual implementation and assimilation of the framework, we had to find solutions for several general problems that appear in acceptance testing. This section describes them.

5.1. Variables

In some cases, the test needs to use values that the tested system created during execution – technical keys, values that depend on date and time, randomly generated value and so on. For example, each entity has a technical key, automatically generated the first time it is saved. Fields that link a data entity to another entity hold the target entity's technical key. Therefore,

when testing whether a hyperlink is correct, we need to compare it to the technical key of the expected entity – which will only be available at runtime.

To handle this, our framework supports defining a set of variables for each test case. These can be initialized to a default value, and also modified using a special "Set Variable" step type. Variables can be used in test steps anywhere a value can, and they are identified by surrounding << >> characters. For example, the following test steps verify that once an account is linked to a customer, the customer's main account field also changes to refer to that account:

Step Type	*Paramater 1*	*Parameter 2*
Open Form	Account	
Edit Field	Customer	John Doe
Edit Field	Amount	100.0
Do Action	Save	
Set Variable	AccountID	ID
Open Form	Customer	John Doe
Check Field	Main Account	<<AccountID>>

5.1. Functions: Shared Test Cases

It is very common that a large portion of a test case be shared with other test cases. This happens when setting up a test case, or when testing different edge cases of the same basic use case.

Going back to our customers and accounts example, assume that we would like to test the transfer customer to new back use case. This could have dozens of minor edge cases, depending on open business of the customer in the source bank, whether it is active or has a history in the target bank, whether the account types in the two banks match, and so on. However, all these tests must start by creating the client and several accounts in the source bank. This could take from several steps to several hundred steps, judging from examples in our own project.

Therefore, it is necessary to support define "function test cases", and allow "calling" these functions. This is required so that these shared functions could be written once and changed once when necessary. To support this, we defined a new document schema called "Test Functions"; a document of this schema represents a "package" of test functions on the same subject. We also added a "Call Function" step, which allows parameter passing. The function itself defines each parameter as a variable (see 5.1), so no special syntax for defining or using parameters was required.

Step Type	*Paramater 1*	*Parameter 2*
Call Function	Delete Account	ID=<<AccountID>>

5.2. Sequences: Dealing with Side Effects

In sharp contrast to unit tests, acceptance tests have serious side effects, and they cannot be designed to avoid this [15,12]. Since acceptance tests verify end-to-end functionality, such tests can – and typically will – create, modify and delete persistent data.

The problem is magnified by the fact that we must keep each test case independent. Consider, for example, a set of test cases that test different edge cases of customer deletion. All these tests start by calling the "Create Customer" function, which creates (in the database) a customer with several accounts. The tests modify some of these accounts to create different edge cases, and then delete the account and verify that the system behaves as required. If "Create Customer" is naively defined, for example by creating a customer called "John Doe", then consecutive calls to this function will fail, because a customer by that name already exists. A test cannot rely on the fact that the previous test will delete "John Doe" – that test may fail, or this test may be run individually or out of order. We must support side effects on one hand, and maintain test case independence on the other hand.

The solution we find to this problem is a mechanism of sequences. In a global document, testers can define a global list of sequences. Each sequence has a name, a prefix, and an initial value:

Sequence Name	*Prefix*	*Initial Value*
Customer Name	John	1

Sequences are accessed from within test cases using a variant of the "Set Variable" step type, which in fact has three parameters: The variable's name, the source of its new value, and the new value. The second "source" column may be "Value" (to read a value or copy another variable), "Field" (to read a field value from the current form), or "Sequence" – to read the next unique value from a given sequence. For example:

Step Type	*Param 1*	*Param 2*	*Param 3*
Open Form	Customer		
Set Variable	Name	Sequence	Customer Name
Edit Field	Full Name	<<Name>>	

The "Set Variable" step will return values composed of the sequence's prefix plus a unique number, i.e. "John 1", "John 2" and so on. By using these values, each test is guaranteed to receive a "fresh" customer, regardless of the execution or results of any other test.

5.3. Computations

In some cases, test steps are defined in terms of expressions that need to be computed. For example:

- Verify that the customer's amount field is the sum of the amounts in all of her attached accounts
- Set field "Expiration Time" to be the current time plus one hour.
- Set field "Requested Loan Amount" to be the value of the "Amount" field, plus 1.0.

Extending the test definition language to include expressions (such as additions) and functions (such as the current time) would greatly complicate the language, which should not be done for the sake for relative rare cases. In fact, over-complicating the test language is a well known anti-pattern [7].

On the other hand, some solution had to be provided, since the alternative was to require such tests to be run manually. The compromise that we finally reached is as follows: The "Set Variable" is extended to be able to execute a script, and read its result into the variable. The script language is VBScript, and testers are not expected to code in it – they request the help of a programmer when required. This keeps the language simple, keeps the framework clean and simple, and still provides a full solution for arbitrary computations:

Step Type	*Param 1*	*Param 2*	*Param 3*
Set Variable	Value	Script	result = var("other") + 1

Since VBScript scripts can access COM objects, including operating system services, this is also a solution for some extreme cases in which tests involve external files, servers or applications outside the normal scope of the tested application.

5.4. Server Testing

The majority of acceptance tests are implemented using the application's user interface. This is natural since they test end-to-end behavior, and not specific units or modules inside the system. However, there are two cases in which even an acceptance tests needs to bypass the user interface and directly activate business logic at lower levels of the application.

The first is data that is received from external systems, though a computerized interface. In some cases, there may not be a user interface inside the application to change or even create this data. For example, if the list of disqualified customers is received from another system, then testers have no way of creating a new disqualified customer. Doing so requires coding new steps, which create new database records for such customers. The general requirement is to bypass the user interface in three cases: to modify fields, to create new data, and to activate transactions.

The second case is testing logic that double-checks client-side logic, for example for security reasons. Such code should never be reached in normal execution, and so the only way to test it would be to bypass the conditions at the client layer, and send the unchecked data directly to the server. A mechanism to receive the results must be devised as well.

Note that this problem exists for manual testing as well. There are two possible solutions: one is to write code at the client layer, which forwards requests to the application's server without running the client business logic code first; another is to implemented the framework executing engine in the server as well, and have the framework open a direct socket to the server and operate on it using the "Select Station" step type. We have not implemented a solution yet, but our preliminary design suggests that the first solution will be simpler to implement and use.

6. Insights

Designing and implementing the acceptance tests framework was a learning opportunity. In the spirit of [7], we'd like to summarize patterns and anti-patterns of a successful acceptance testing solution as we see it.

First, here are the anti-patterns, or "don't do" list:

- Don't create a tool for programmers. This happens because programmers usually build the automated acceptance test solution; however, testers and customers don't edit XML, don't use an IDE, and don't find writing their own queries amusing.
- Don't use record-and-playback tools. These tools create non-readable tests, which must be re-done upon any change to the system's user interface. They should be used only on strictly legacy systems.
- Don't create a mega-language. The framework is intended for non-programmers, and most of the test cases are simple and do not require computations, conditions, loops, recursion and so on.
- Don't aim to automate 100% of the tests. 80% is fine – edge cases often won't return the investment.

Here are known patterns that worked well for us:

- Create a high-level domain language. Use simple verbs, named after the system's main metaphors.
- Design for built-in testability in the application. Build a framework separates the UI layer from the data and actions layer, so that actions can be easily accessed by both the UI and an external engine.

- Add determinism to your application.
- Integrate with build – acceptance tests should be run as part of the night build. However, such tests may fail, and this should not break the entire build.
- It's all right to fake it – Not every layer should be tested as thoroughly. For example, bypassing the UI framework layer, that has no specific business logic, is worth the time it saves.

Some of the patterns, such as "Tests are the Requirements" and "Tests as Conversation Pieces", do not fit in our environment. This is due to the size of our project, in which the customer, business analyst and tester roles are all played by different people. Such patterns are much more natural in smaller XP teams.

We have also formulated several new patterns, which we highly recommend for any similar framework:

- Hyperlink to Specifications. Develop an automated link to your specifications repository, and provide as much as possible auto-completion for testing in writing test steps. This dramatically raises test writers' productivity, reduces errors in test cases, and enables automated coverage and impact analyses on the tests.
- Store tests in files, not a database. This is required to enable the use of the same configuration control tool used for the code, for the tests as well. It leads to a simple definition of a baseline of tests, which matches a baseline of the product.
- Support Interactive Automated Test Execution. The interactive mode speeds us the initial automation of test cases, enables on-the-fly corrections, and helps programmers reproduce bugs faster.
- Unify tools. In our case, the same metadata repository was used for editing both specifications and tests. It was also the same tool for writing and for running tests. This makes life easier and more productive both for the framework's users and for its developers, since many of the underlying aspects of the framework – hyperlinks, queries, scripting, configuration control and so forth – behave in the same way and can be shared.
- Use sequences to deal with side effects. After considering a multitude of possible solutions, sequences were the easiest for the testers to use, are simple to implement, and work well in practice.

7. Conclusion

We have presented our new acceptance testing framework, which builds on existing best practices, yet offers several new insights of its own. The major innovation is supporting the entire range of QA tasks equally well, using a combination of architecture and methodology which together lead to a significant productivity boost to any QA-related role. Since both tool and process were implemented in a large, real-world project, we are confident to strongly recommend the same path for your next acceptance testing solution.

We would like to thank the following people for their help and sound advice in building the framework and writing this paper: David Berenthal, Dror Zalika, Uri Landau, Gil Kulish, Ora Brener and Albert Haroni.

8. References

[1] Beck, K., *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[2] Beck, K., *Test-Driven Development: by Example*. Addison-Wesley, 2002.

[3] Canna, J., "Testing Fun? Really?", IBM DeveloperWorks Java Zone, 2001.

[4] FAT Acceptance Testing Framework. *http://sourceforge.net/projects/fat/.*

[5] FIT Acceptance Testing Framework. *http://fit.c2.com*

[6] FitNesse Acceptance Tests Framework. *http://fitnesse.org*

[7] Hanly, S., "Build Your Own Acceptance Test Framework", Presentation from XP Day 2003. *http:/.xpday.net/scripts/view.pl/Xpday2003/Program.*

[8] Highsmith, J., "Adaptive Software Development", Presentation at OOPSLA 2000.

[9] Hunt, A. and Thomas, D., *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley, 2000.

[10] JAccept. *http://roywmiller.com/papers/AcceptanceTesting.htm*

[11] Kaner, C., James, B. and Pettichord, B., *Lessons Learned in Software Testing*, John Wiley & Sons, 2001.

[12] Kanglin Li and Mengqi Wu, *Effective Software Test Automation: Developing an Automated Test Tool*. Sybex, 2004.

[13] Mercury TestDirector, WinRunner and QuickTest Pro. *http://www.mercury.com/us/products/quality-center*

[14] Rational TestManager, Robot and other products. *http://www-360.ibm.com/software/rationa/offerings/tesing.html*

[15] Rogers, R., "Acceptance Testing vs. Unit Testing: A Developer's Perspective", XP/Agile Universe 2004, LNCS 3134, pp. 22-31, 2004.

[16] Talby, D. et al, "The Design and Implementation of a Metadata Repository", INCOSE/IL, 2002.

[17] Various Authors. *http://c2.com/cgi/wiki?AcceptanceTest*