Getting

A practical
approach
to test
development
and automation

by Hans Buwalda
and Maartje Kasdorp

Automated Testing Under Control

Testing of systems is probably the most difficult task there is in IT—especially if you want to execute the tests automatically. We make our living running test projects for customers, and have experienced many of the potential problems in testing and test automation ourselves:

- Testing costs time and money (usually during the end phase of a development project, when such time and money is least available)
- Manual execution of test cases is often a tedious and error-prone task (especially when tests have to be repeated on successive versions of a system)
- Automated tests tend to be very sensitive to changes in the target systems interface
- It can be difficult for outsiders to get a clear overview of what is being tested (especially when using automation scripts that non-IT people find hard to understand)

During the years we have evolved an approach to counter these issues. Although you shouldn't regard it as a "silver bullet," we think it can be very practical; we would like to use this article to share it with you. The approach is marketed under the name TestFrame™, but you can easily apply the techniques and principles outlined in this article yourself.

Test Clusters

Let us start with a look at what we have called the "test analysis process." The first step is to identify so-called *test clusters*—collections of tests that have more or less the same scope and level of detail. Even for big projects the number of test clusters should never be very large; typically there should be no more than two dozen of them for each project.

QUICK LOOK

- Identifying test clusters
- Using action words in test design

For a banking application, you might expect something like the following list of clusters:

- **Tests of the User Interface** Are all screens, fields, and controls accessible? Do they work as they are supposed to? Are the proper help screens displayed? Does the tab bring the focus to the proper next fields?
- **Tests of the Entry Process** Are the mandatory fields really mandatory? Are the proper default values displayed? Do the right values appear in list boxes?
- **Tests of Relationship Management Functionality** Do the customer names you've entered really appear in the database? Can they be updated, removed, etc.?
- **Tests of Payments and Transfers** Are all possible forms of payments and other money transfers working properly? Is all data, like account balances, updated appropriately?
- **Tests of Interest Calculations** Are mathematical calculations processing data as you had expected?

The table below is a list that might help you decide about the division of the testing task into clusters (don't see this as the law; your project may require different priorities). The rows are in descending order of importance.

You might begin creating test clusters by separating module tests, system tests, functional tests, and performance tests. Then you might further divide along lines—such as the part of the system that is being tested, or the department that has to be involved in the test development.

We always try to make at least a provisional list of test clusters one of the first actions in a project. If possible we do this together with the users and other involved parties in a joint session with a whiteboard. Listing intended test clusters is a high priority because it's a very good basis for planning and organizing the rest of the testing project. For each cluster, for example, we can decide on its priority and who will do the work. We can also track progress of development and execution by cluster.

Dividing Testing Tasks into Clusters

PRIORITY	CRITERION	EXPLANATION
1	Logic	The division should be perceived as logical by the people
2	Independence	Execution of each test cluster should generally be independent of the execution of other test clusters (i.e., output of one test cluster should not be used as input for other clusters).When there are dependencies between test clusters, these should be the result of a well-considered decision.
3	Type of Test	The division should take into consideration the type of test to be done, e.g., module tests, system tests, functional tests, or performance tests (but less formal tests such as user-friendliness tests can also be identified as a "test cluster").
4	Scope	The division should take into account the scope that has been decided upon in the test strategy.In many cases this is more or less the part of the system to which the tests in the cluster will apply.
5	Intended Method of Execution	Separate clusters can be identified by the way the test is probably going to be executed (e.g., manually, automated with a record-and-playback test tool, automated with a C program, or organized in a usability lab).
6	Project Issues	What functionality does the customer want to have tested first? When is necessary design information going to be available? In which order will parts of the system be completed? These are questions to take into account as you divide a test into clusters.
7	Cluster Size	To some extent, the size of the test clusters should be taken into consideration as well.If a test cluster becomes very large, consider splitting the clusters further.On the other hand, if clusters are very small, you might consider combining them.

Test Design for a Cluster

In the TestFrame approach all further test development is done at the level of the individual test cluster. Test clusters are usually kept in spreadsheet files, which are further divided into individual sheets. In some of our projects we have connected the spreadsheet environment to a repository-type database containing information such as system requirements, test plans, and test results. Why use spreadsheets? They are very rich in functionality, are well suited to manipulate lines and columns, and also have the possibility to perform calculations. For example, it is easy to make copies of tests and vary them using spreadsheet formulas.

In Figure 1, an example of a test cluster for an imaginary banking application is shown. It includes a list of "test conditions" and one or more sheets consisting of "test lines." Let's first look at the test conditions. A *test condition* is a concise and readable statement on how a certain aspect of the system should behave. In many cases it is directly related to an underlying business rule in the system. At the level of these test conditions, the different steps that need to be taken when testing the business rules are not yet mentioned. It is more the "what" that will be formulated than the "how." The test conditions are a good level for a business expert to assess whether the tests will be correct and complete.

Connected to the test conditions we have the tests themselves. They consist of shorter or longer sequences of *test lines* (during the execution of the test these lines will be

test sheet	Example of a TestFrame Test Cluster				
version	1.0				
author	Marly Testwell				
test condition	TC1	New customers can be entered into the system			
	last name	*first name*	*account nr*	*balance*	
enter customer	Green	John	458473948	1500	
enter customer	Wood	Anne	422087596	2100	
	account nr	*last name*	*first name*		
check name	458473948	Green	John		
check name	422087596	Forest	Anne	(incorrect line)	
test condition	TC2	Money can be transferred between two accounts			
	from	*to*	*sum*		
transfer	458473948	422087596	500		
	account nr	*balance*			
check balance	458473948	1000			
check balance	422087596	2600			
test condition	TC3	Every client has to have a unique account number			
	last name	*first name*	*account nr*	*balance*	
enter customer	Savy	Danique	456182101	89005	

FIGURE 1 **An example of a Test Cluster**

interpreted one by one). Every line starts with a field called the "action word," which specifies what has to be done.

Action words are used for entering one or more values, generating an event, or checking an outcome. The action word is followed by a number of arguments specifying data needed by the action, such as input that has to be entered or "expected values" that are to be compared to the real outcomes.

The two most important advantages of working with action words are probably *readability* and *maintainability*. Tests are easy to read because all details needed for their execution (like which buttons have to be pushed or at what location on the screen an outcome can be found) are hidden behind the action words. The testers don't have to bother with them. When those execution details change, even if these changes are substantial, it will most likely not influence the test cluster.

Our example starts with a couple of commentary lines: the name of the cluster, the version, and the author. Next, we record which test condition we'll be testing. Then, we can start entering customers, using the action word **enter customer.** Note that such action words are specific for an application (when doing test jobs for military ships, for example, we have seen much more often action words such as "fire torpedo"). You can clearly see that the same action word is used two times, with different arguments.

In the lines that state **check name**, the names are checked against the list of names already existing in the database. The first argument contains the account number of the customer to be checked; the next two arguments contain the expected values.

A later line starting with **transfer** describes a money transfer. To execute it we might have to activate a "transfer screen" by selecting a menu item, entering the data there, and pressing a "process" button. But it can also mean that we have to enter a record into a batch file, wait until we have done all other test lines specifying input, execute a batch job, download a result file, and perform all the checks specified in the same cluster.

Finally, the balances of the two customers are checked. Their values should be the initial balances plus or minus the sum that was transferred.

Results

Of course if you are executing a test you want to know what the results are. We always try to produce reports automatically and give them easily accessible layouts, matching their level of detail to the level used by the tester in the cluster (a sample report format is shown in Figure 2). Reports begin with general information about the test, followed by the test lines. When there are differences between the expected and the actual results, they are shown as "failures." (In the example shown here, it's clear that the tester confused the last name "Wood" with "Forest," resulting in a failure.) At the end of the report, a summary is produced, showing general statistics such as the number of passed and failed checks—as well as the lines in which the fails occurred.

This report format is just an example. You can easily convert it to meet the needs of your particular situation, or to display the report in another technical form—such as an HTML document, or entries into a bug-tracking system. What is important is that the report displays the results at the same level of detail as in the test cluster, avoiding the distraction of unwanted details (e.g., which button was pushed or what the title of a displayed window was). The report must be clear and concise, giving the tester and developer the level of information needed to assess the results and to track any problems. To that purpose, we do sometimes print some additional information when there is a fail, such as dumps of the windows as they appeared at the time of the problem.

To make the tests more readable we organize them into groups within a sheet. The test lines can be grouped in several ways, depending on the kind of test. For functional tests, we most often use small test cases that have only a few actions and checks, or we use longer test scenarios that simulate complex business processes. Some test scenarios we call "soap operas." They describe sequences of events taken from everyday business life—but exaggerated in the

```
cluster name          :   Example of a TestFrame Test Cluster
cluster version       :   1.1
cluster author        :   Marly Testwell

application version   :   2.5a

run date and time     :   January 1,2000 13:52:19
-------------------------------------------------------------------
Test Condition TC1 New customers can be entered in the system

2 (7):   enter customer Green          John     458473948   1500

3 (8):   enter customer Wood           Anne     422087596   2100

4 (11):  check name     458473948      Green    John
pass                                   Green    John

5 (12):  check name     422087596      Forest   Anne
FAIL                                   Wood     Anne

Test Condition TC2 Money can be transferred between two accounts

7 (16):  transfer       458473948      422087596   500

8 (19):  check balance  458473948      1000
pass                                   1000

9 (20):  check balance  422087596      2600
pass                                   2600

Test Condition TC3 Every client has to have a unique account number

11 (24): enter customer Savy           Danique  456182101   89005

-------------------------------------------------------------------
end of cluster        :   Example of a TestFrame Test Cluster
finished at           :   January 1,2000 13:52:31
time used             :   12 seconds

number of cluster lines :  431
number of checks      :   75
number passed         :   72
number failed         :   3
percentage passed     :   96%

failed at report line(s):
5,36,402
```

FIGURE 2 An example of a TestFrame report

way most television soap operas or novellas are. That makes them a good test for the system. Soap opera tests are usually made by (or in conjunction with) end-users or business specialists.

Test Design Templates

Although this article's focus is not test specification techniques (e.g., decision tables, limit analysis, etc.), most of these techniques do play a role in the TestFrame approach. Decision tables, for example, can be used to formulate test conditions, and limit analysis can be used to make test lines. In our projects we use many techniques to produce our tests, depending on the situation. One extension to our approach that we've found particularly interesting is that proposed by Edward Kit called *test design templates*. These templates are a particularly good technique to help get you from test conditions to test lines.

Navigation Engineering

It is common practice to automate the execution of the test made with TestFrame test clusters by translating the lines into step-by-step instructions for a test tool. When for some reason it is desirable to execute the tests by hand, the spreadsheet can be used to generate instructions for the user in much the same way.

We use the term "navigation" to describe the automation of the test execution because it indicates the job of finding a way through the winding paths of an application's interface. The navigation is a separate activity focused on the execution of the tests. We usually refer to the people responsible for the navigation as the "navigation engineers." In this section we will explain something about that navigation process. (This part of the article is a bit more technical then the rest, but non-technical people can skip it without losing too much of the basic ideas in our approach.) For automated execution, a "navigation scheme" is constructed. This scheme consists of several components, the most important of which are:

A **Actions created by the navigation engineer, including**

- low-level actions
- intermediate-level actions
- high-level actions

B **The "engine"**

The low-level actions implement single actions on individual elements of the systems user interface (for example, pushing a button on a window). The intermediate-level actions, although still aimed at the interface of the target system, are more complex actions (e.g., enter all data in a window and push the OK button or enter a transaction in an ERP system). The high-level actions are complex and aimed at the test, not necessarily the target systems interface. One high-level action can use more than one window and/or use only part of the fields available per window.

The actions mentioned above, like **enter customer**, are examples of high-level actions. In a typical navigation scheme, these high-level actions are the starting point.

They will call the intermediate-level actions, which in turn use the low-level actions.

The easiest way to implement action words is to use the script language of a testing tool. The three-level navigation scheme can be implemented by making a function for every action. The functions for the high-level actions call those for the intermediate level, which in turn call the functions for the low-level actions.

Our example could look something like Example 1, using an imaginary test tool script language.

Functions like **PushButton**, **EnterField**, and **SelectCheckBox** are low-level functions. In most cases, their implementation is straightforward using functions in the testing tool. We also use our approach for testing software without a user interface, such as embedded software. In such cases, the lowest-level functions do things like calling api functions or sending network messages, usually directly in a programming language like *C*. For the structure of the navigation scheme, this makes very little difference.

The function **EnterPersonalData** is a medium-level function, designed to operate on one window, following more or less the layout of that window. It is meant to be used by high-level functions like **EnterCustomer** in this example. The high-level function, directly connected to an action word in a spreadsheet, takes the arguments from the cluster line ("arg(2)" is the B column), adds extra default values (e.g., "female" for the gender), and inputs them in one or more of the windows in the target system.

The last line, **RegisterAction**, puts our high-level action **(EnterCustomer)** into a table, which we call the "action list." This table is used by a standard module that we have called the "engine." It reads the lines from the cluster one by one and executes the proper function for every line—based on the registered action word. It also puts the arguments from the cluster line into another table called the "argument array," making them available to the script executing the action word.

When we want to execute a test, we normally export the cluster first from its spreadsheet format to a tab-separated text file (a standard export option in most spreadsheet programs). The engine reads the lines from that text file and interprets them. It also does other general tasks, such as producing reports. Over the years our engine has become quite an extensive standard product—doing complicated tasks such as running tests simultaneously—but it is not too difficult to create one yourself with enough functionality to process most of your tests.

We think it is important to mention two extensions to the navigation scheme here, namely "table-driven navigation" and "template-based navigation."

By *table-driven navigation* we mean the use of tables containing information about details of the target systems interface (e.g., all the screen objects for a given window). Using such tables it is possible to implement intermediate-level actions very efficiently. For example, one function can be named **EnterScreenData**, designed to input all fields of a window. The window itself is a parameter for that function. The function **EnterCustomer** referred to in Example 1 could now read something like Example 2 (next page).

(In this example, arguments 4, 2, and 3 from the cluster are entered, followed by the value "female" as a fourth entry.)

In *template-based navigation*, we specify the high-level action not in the scripting language, but in just another cluster (spreadsheet)—as we would the actions in a test. To do that, we have introduced a standard action word **DefineTemplate**, which defines a new action word with parameters. Once defined, the new action word can be used like any other action word. The lines following the **DefineTemplate** line contain the actions that have to be executed if the new action is used in a cluster. Our high-level action word implemented with template-based navigation could look like Example 3.

This defines the action word **EnterCustomer** with the parameters **firstname** and **lastname.** The "**&**" indicates the parameters. The new action word can be used later on in a cluster like Example 4.

First the lines defined with **EnterCustomer** are executed with "Olivia" as **firstname** and "James" as **lastname**; they are then executed again with the values "Eduardo" and "Lopez."

Using TestFrame in Practice

We have described an approach for creating a maintainable and structured test set and automating its execution in a reusable way. In this approach the *design* of the tests is strictly separated from the *automation* of the test. This article has given a first introduction to the approach; for more information you are welcome on our web site about TestFrame (www.testframe.com).

```
// function for a high level action
Function EnterCustomer
        PushButton 'Relations'
        EnterPersonalData arg(4), arg(2), arg(3), 'female'
        EnterFinancialData arg(4), arg(5)
        ...

// function for an intermediate level action
Function EnterPersonalData ( number, firstname, lastname, gender)
        EnterField 'Account Number', number
        EnterField 'First Name', firstname
        EnterField 'Last Name', lastname
        SelectCheckBox 'Male/Female', gender
        ...

// register the high level action and connect it to a function
RegisterAction 'enter customer', EnterCustomer
```

EXAMPLE 1 Using a test tool script language to implement the action word "enter customer"

```
// high level function
Function EnterCustomer
        PushButton 'relations'
          EnterScreenData 'personal', arg(4), arg(2), arg(3), 'female' ...
        ...
```

EXAMPLE 2 Table-driven navigation for "EnterCustomer"

DefineTemplate	EnterCustomer	&firstname	&lastname	...
PushButton	relations			
EnterData	personal	&firstname	&lastname	female
PushButton	financial			
EndTemplate				

EXAMPLE 3 Template-based navigation for "EnterCustomer"

EnterCustomer	Olivia	James
EnterCustomer	Eduardo	Lopez
...		

EXAMPLE 4 The defined action word can be used in other clusters

In most cases the approach, with test clusters, test lines, and navigation, is fairly straightforward to start with. Having a spreadsheet and a test execution tool is usually enough. However, both testing and test automation are difficult areas in IT, with or without our approach. There is a wide choice of pitfalls to stumble into and lessons to be learned. Stick to the principle of keeping tests separated from their execution, and pay attention to the way tests are divided into clusters. Also keep in mind, as we mentioned in the introduction, that our approach is not a magic wand. Testing is a complex and highly critical activity that should never be underestimated, with or without the approach outlined here. Take care that everybody involved in a testing project understands this. Having tests automated does not mean that a "push of button" will solve all testing. Careful planning and attentive management of the testing activities stay as important as with any method. STQE

References:

Hans Buwalda. "Testing with Action Words." Presentation for STAR'98West Conference, San Diego, California, October 1998.

Hans Buwalda. "Testing with Action Words: Abandoning Record and Playback." Presentation for Eurostar 1996 Conference, Amsterdam, December 1996.

Edward Kit. "Integrated, Effective Test Design and Automation," *Software Development Magazine* February 1999.

Maartje Kasdorp is a consultant for TESTars, a leading International Testing services group based in the US, and India. Mr. Buwalda was the original founder of TestFrame. Ms. Kasdorp has been responsible for much of the test development lifecycle in the TestFrame approach. You can reach Maartje Kasdorp at:

maartje@testars.net .