

Automation Test Framework - ATF

**A framework for language implementation
of procedure-based test cases**

Authors

Daniel Barboza Franco
Daniel Drigo Pastore
Marcel Augusto Gorri

July / 2010

| Author | Version | Description |
|---|---------|---|
| Daniel Franco, Daniel Pastore, Marcel Gorri | 1.0 | Initial version |
| Daniel Franco, Daniel Pastore, Marcel Gorri | 1.1 | Improvements in the layout of the document and examples section |

Summary

| | |
|--|----------|
| Automation Test Framework - ATF | 1 |
| A framework for language implementation | 1 |
| of procedure-based test cases | 1 |
| 1 - Introduction..... | 4 |
| 1.1.1 - Test Definition Mechanisms..... | 4 |
| 1.1.2 - Test Procedures..... | 4 |
| 1.1.3 - Execution | 4 |
| 1.1.4 - Report | 5 |
| 1.2 - Supported test approaches | 5 |
| 2 - Architecture | 6 |
| 2.1 - Introduction | 6 |
| 2.2 - Test Definition Mechanisms..... | 6 |
| 2.2.1 - Editors and Views | 6 |
| 2.3.1 - Description | 7 |
| 2.3.2 - Command Definitions and Commands | 7 |
| 2.3.3 - Image Based Test Definition | 8 |
| 2.3.4 - Reusing Procedures | 8 |
| 2.3.5 - Using property files..... | 9 |
| 2.3.6 - Assertions | 9 |
| 2.4 - Execution | 10 |
| 2.5 - Report | 11 |
| 3 - Technical Approaches | 12 |
| 3.1 - The GEL Language..... | 12 |
| 3.1.1 - Description | 12 |
| 3.1.2 - Being generic..... | 12 |
| 3.1.3 - Examples of use | 12 |
| 4 - Appendix | 14 |
| 4.1 - Showcase..... | 14 |
| 4.1.1 - Web Applications | 14 |
| 4.1.2 - Camera | 14 |
| 4.1.3 - GPS | 15 |
| 4.1.4 - SQL | 16 |
| 4.1.5 - FTP | 16 |
| 4.1.6 - Circuit Boards | 17 |
| 5 - References..... | 18 |

1 - Introduction

Problem: Currently software is getting more and more complex. To ensure the quality of such programs is a task rather laborious and time consuming, always prone to human error.

Proposed solution: An automation test framework, that is language and platform agnostic, providing means to define and implement any sequence of steps for a test, with a seamless integration of tools and support necessary for tests in a single Integrated Development Environment based on Eclipse technology. Through the use of this framework, the user will be able to test a mobile application interacting with a device emulator, a GPS simulator, a camera connected to the PC's USB port, a user registration web service, among many other possibilities, in an automated way.

1.1 - Description

The objective of Automation Test Framework is to allow the creation of automated test cases in an easy way. The user will be able to specify test requirements, purposes, steps and assertions written in an procedural language.

We propose the following flow to create and run tests:

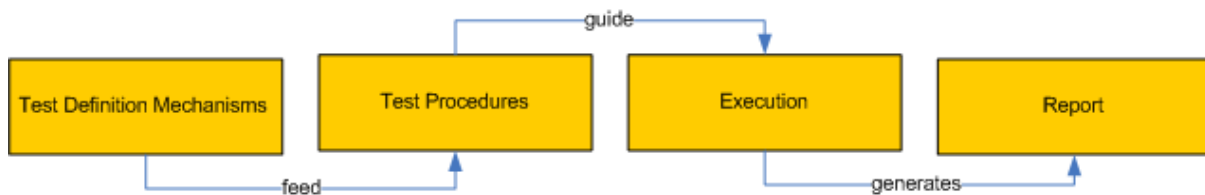


Figure 1: Test definition and execution flow in Automation Test Framework

1.1.1 - Test Definition Mechanisms

Typically, the user will define test cases in Automation Test Framework using Eclipse Editors and Views. The user can also define the test cases using any other tool or editor (Notepad, Vi) and then import them into an Eclipse project defined in Eclipse.

1.1.2 - Test Procedures

The test cases are written as a sequence of steps. These steps are inputted by the user through the Test Definition Mechanisms and will compose procedures. We call them procedures because they are groups of steps of an algorithm, and each step is defined by a command that can be implemented by the user if it is not already implemented. Such procedures are written in a procedural language called GEL (which stands for Generic Extensible Procedural Language) that will be described later in this document. GEL shall be generic and extensible enough so that the commands defined by it can apply to a wide range of platforms.

1.1.3 - Execution

Once defined, a test procedure can be executed in the Eclipse IDE, integrated with all the features available in this environment. The user will be able to define different

launch configurations for every case needed. Another important feature in Automation Test Framework is the ability to keep track of the execution of the test. In other words, the user will be able to perform operations that affect the execution flow of the test, such as play, pause, resume, record. This feature adds a high level of control and flexibility to the execution of tests in general.

1.1.4 - Report

After the test execution finishes, it will be possible to gather all relevant test information, including percentage of passed tests, failed tests, test execution history and test execution trace (Java stack trace, for instance). Such reports will be available in different ways, such as summarized report, detailed report, graphical representation, among others. Likewise, it will be possible to save any test evidences generated such as images, videos and log files, so any problem identified during the test execution can be further analyzed with more details.

1.2 - Supported test approaches

Being a flexible and extensible framework, Automation Test Framework shall support different test approaches [2], such as:

- **Data-driven testing** is defined as a methodology used in test automation where test scripts are executed and verified based on the data values stored in one or more central data sources or databases [5]. Once Automation Test Framework can manage test data stored in property files or any other source, data-driven testing is supported.

- **Modularity-driven testing** requires the creation of small, independent scripts that represent modules, sections, and functions of the application-under-test [6]. Automation Test Framework allows the use of scripts by means of commands.

- **Keyword-driven testing** is a software testing methodology for automated testing that separates the test creation process into two distinct stages: a Planning Stage, and an Implementation Stage [7]. Automation Test Framework contemplates not only the implementation of tests, but also the Planning Stage through the edition of test related information.

- **Model-based testing** is the application of Model based design for designing and executing the necessary artifacts to perform software testing. This is achieved by having a model that describes all aspects of the testing data, mainly the test cases and the test execution environment [8]. Automation Test Framework also allows the definition of models through the use of commands.

2 - Architecture

2.1 - Introduction

Automation Test Framework's architecture was devised strongly based on Eclipse concepts, particularly those of extensions and extension points. Figure 2 shows the main components that comprise the architecture for this solution, and they are later explained in the further sections of this document.

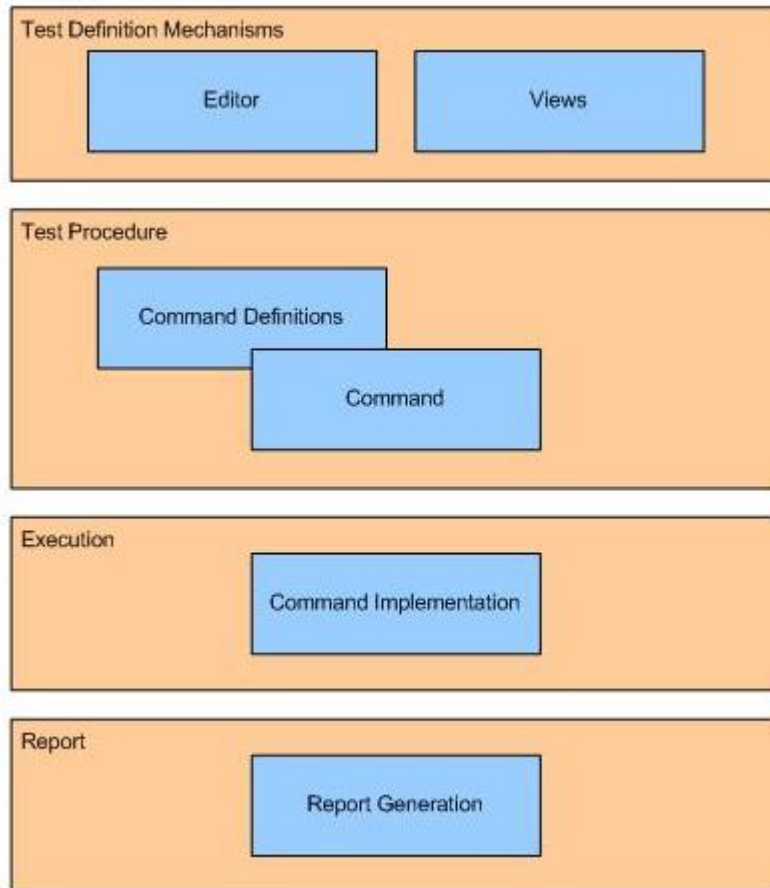


Figure 2: Automation Test Framework Architecture Diagram

2.2 - Test Definition Mechanisms

Test Definition Mechanisms are the means through which the user can create or define test cases. Either by using well-known Eclipse UI components such as Editor or Views, or by creating their own mechanisms, the user can specify what are the requirements, steps, stop criteria and verification factors for their tests. The idea of using commands to represent interactions or test steps allows us to manipulate such interactions by means of abstractions, i.e. Editors and Views.

2.2.1 - Editors and Views

Editors are the primary mechanism for users to create and modify resources (e.g., files). Eclipse provides some basic editors such as text and Java source editors, along with some more complex multipage editors [9]. Being the natural choice for Eclipse users to write

code, Code Editors are expected to be the right choice for Automation Test Framework as well, so users can define and implement tests with them.

Views are typically used to navigate resources and modify properties of a resource. Any changes made within a view are saved immediately. By contrast, editors are used to view or modify a specific resource and follow the common open-save-close model [9]. This way, it should also be possible to use Views in Automation Test Framework. The idea is to have a framework as flexible and useful as possible, attending user's need.

The main Test Definition Mechanism in the Automated Test Framework is the Command Editor. It is a simple text-based editor with syntax highlight and code completion. Features useful for writing code like folding, code completion and syntax highlight shall all also be present. It will also have drag&drop support, so you can choose an image to be used as a parameter of a method. Other Test Definition Mechanisms can be created for specific matters such as a GPS coordinates editor or a camera simulator.

2.3 - Test Procedure

2.3.1 - Description

As stated before, a procedure is a set of steps of a test case. The idea is that every step in a test case is translated to a command. A command can be defined and implemented by the user in order to perform a necessary part or task of the test. Those commands are part of our test description language, GEL (see Section 3.1 for more information about GEL). For example, below we have a simple test case for a user registration feature that we want to describe using a procedure:

- | |
|---|
| <ol style="list-style-type: none">1. Type the name of the user2. Type the address of the user3. Click in the Confirm button4. Verify if the user has been registered |
|---|

Test case 1

Using a test description language, this simple test case could be represented as:

```
type("John Doe")
type(keys.TAB)
type("Eclipse street, #42")
type(keys.TAB)
type(keys.ENTER)
```

Procedure 1-A - Inserting an user in a database

2.3.2 - Command Definitions and Commands

A command can be defined and implemented. Sequoyah will provide in the Automation Testing Framework some basic commands already implemented, such as receiving input from the user, displaying information on the screen, among others. However, the user will be able to extend already existent commands and even to define their own commands. This will be done by means of extension points and extensions.

According to [9], Extension points are used throughout Eclipse as a mechanism for loosely coupling chunks of functionality. One plug-in declares an extension point in its plug-in manifest, exposing a minimal set of interfaces and related classes for others to use;

other plug-ins declare extensions to that extension point, implementing the appropriate interfaces and referencing or building on the classes provided.

This way, we will have the command definitions as the extension points, and the commands themselves as the extensions.

2.3.3 - Image Based Test Definition

A nice addition to our framework would be image recognition, as such a feature would make possible having more realistic test cases. So if we want a click in a specific region of the screen, we can represent it using a screenshot and have a command like

```
click(somepicture.jpg)
```

Using this approach in our user registration example, we could have a form with fields for user related information and also a confirmation button. In the image below, we have the confirmation button labeled with the text "save", so the user would have to click on this button to finish the registration process.




Figure 3: User registration page with a confirmation button

This way, assuming we have an image file representing the confirmation button from Figure 3, our previous test procedure could become:

```
type("John Doe")
type(keys.TAB)
type("Eclipse street, #42")
click(ConfirmButton.jpg)
```

Procedure 1-B - Introducing image files as parameters

or even

```
type("John Doe")
type(keys.TAB)
type("Eclipse street, #42")
click()
```

Procedure 1-C - Introducing images as parameters

2.3.4 - Reusing Procedures

For another test, assuming that Procedure 1-A is a prerequisite, one can visualize the information related to that registered user with the following steps:

```
execute(procedure_1A)
select(user, "John Doe")
display(name)
```

```
display(address)
```

Procedure 1-D - Displaying in the screen information about an user previously registered in another test

2.3.5 - Using property files

Automation Test Framework will also provide a way to ease the creation of small variations of the same test, or the use of different sets of inputs for the same test. This will be done through the use of property files. For instance, instead of manually changing the data of the users to be registered in a test, we can create a property file containing the information of the users to be registered, using key/value pairs.

```
name=John Doe
address=Eclipse street, #42
name=Jane Doe
address=Eclipse avenue, #43
```

Figure 4: Properties file containing information of users to be registered

Modifying the current example to work with properties files, we would have:

```
read(properties_file)
load(user_name)
load(user_address)
click(ConfirmButton.jpg)
```

Procedure 1-E - Using property files

2.3.6 - Assertions

Until now, we did not explain how the user could verify if the action he is testing was successfully performed, i.e., if the user "John Doe" was registered in the database. We intend to implement this verification using assertions. When we write test cases, we usually validate an assertion before a set of steps is completed. In Test case 1, this assertion could be "A message stating success shall appear".

In GEL this kind of assertion is represented as:

```
assert("The user John Doe has been successfully registered.")
```

or simply:

```
assert(registered.jpg)
```

The complete version of our procedure to test user registration will be:

```
type("John Doe")
type(keys.TAB)
type("Eclipse street, #42")
click(ConfirmButton.jpg)
assert(subscribed.jpg)
```

Procedure 1-F - Introducing assertions

Note: This kind of assertion relies on image recognition.

There are other types of assertions that do not depend on image recognition. For example, in a test for a web-based application, the user can expect as the result of their test a web page containing predefined success or error codes. Instead of relying on the user interface for this verification, the user can check directly the returned code.

```
type("John Doe")
type(keys.TAB)
type("Eclipse street, #42")
click(ConfirmButton.jpg)
assert(success_code)
```

Procedure 1-G - Checking return codes

However, GEL provides a possibility of treatment when a test does not present the expected result. For instance, if in the previous example an error code is returned after the confirmation action, it is possible to print any test related information.

```
type("John Doe")
type(keys.TAB)
type("Eclipse street, #42")
click(ConfirmButton.jpg)
assert(success_code, treatError())
```

Procedure 1-H - Taking some action when an error occurs

In this example, the procedure `treatError()` will be called if an error code is received after a tentative of registering a new user and it will print information related to the test executed.

2.4 - Execution

As described before, Automation Test Framework will allow the user to interfere in the test execution flow, performing operations such as play, stop, resume, record, among other. Therefore, the main focus of this section is to briefly describe how the execution flow control mechanism will be implemented.

We will have basically for elements, as described below.

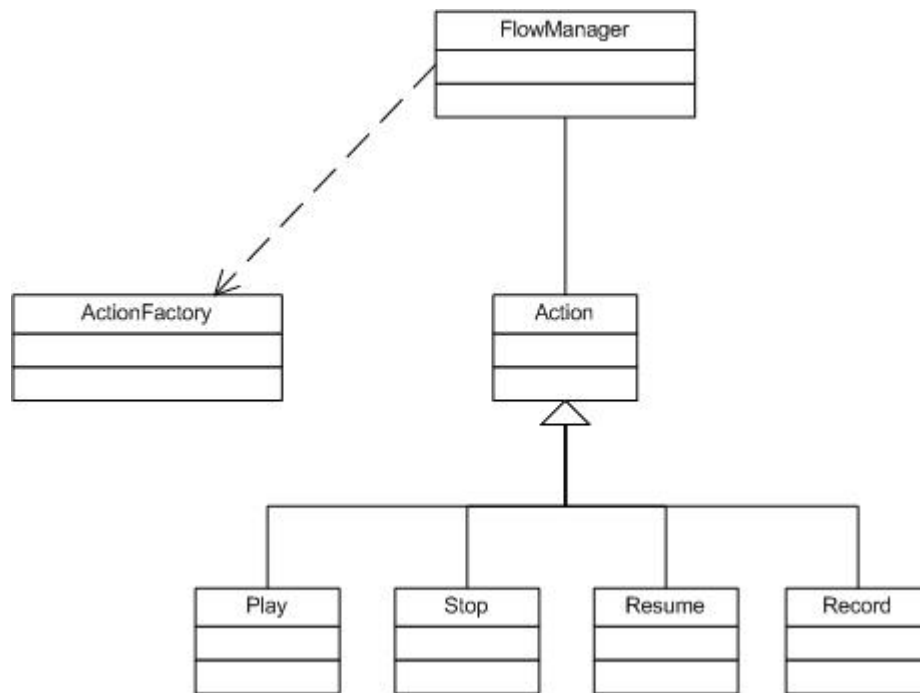


Figure 5: Diagram class for the components involved in the test execution

Action: Abstract class for definition of the several types of actions, such as Play, Stop, Resume and Record.

ActionFactory: responsible for creating the different types of actions.

Flow Manager: responsible for managing the operations that can affect the test execution flow. It will interact with the ActionFactory in order to obtain the Actions to be executed and coordinate their actions.

2.5 - Report

The main component in the Report area will be an Eclipse view with all test status related information. It will basically contain:

- Test execution status, with test execution progress and failed/passed percentage
- Detailed description of individual test execution, with useful information such as test execution history and test execution trace (Java stack trace, for instance)
- Controls for managing test execution related media, such as taking and submitting images, videos and log files from tests.

3 - Technical Approaches

3.1 - The GEL Language

3.1.1 - Description

With the purpose of defining the test cases in Automation Test Framework, we propose a language through which the user will be able to use commands as the steps of their tests.

Being procedures defined as a series of actions or operations [4], and since a test case is a series of steps to be executed or performed, we understand this test-description language as a procedural language.

Such language will be generic and extensible as well. This way, the user can either use the predefined basic commands in their test or define new ones to fit exactly their needs.

For such reasons we decided to call this test-description language as GEL (Generic Extensible Procedural Language). Also, from the definition in Wikipedia [3], "*A gel is a solid, jelly-like material that can have properties ranging from soft and weak to hard and tough*". This is the very essence of our language, the ability of being molded to the most different purposes and scenarios.

3.1.2 - Being generic

In order to be generic, we decided that the commands of GEL will be implemented as extension points. This way, once a command is defined, specific implementations of this command can be defined using different extensions. This approach guarantees that if someone needs a new command they can define it and implement it.

GEL is platform and technology agnostic since it allows different implementations for the same command. For this reason our proposal is that a command will be translated directly to a method of a Java Class. This code translation approach is similar to other languages, such as processing [1] .

3.1.3 - Examples of use

To exemplify the use of GEL, we have below Procedure 1-F being converted to Java.

```
type("John Doe")
type(keys.TAB)
type("Eclipse street, #42")
click(ConfirmButton.jpg)
assert(subscribed.jpg)
```

Procedure 1-F - introducing assertions

```
public static void main(String str) {
    IOUtils.type("John Doe");
    IOUtils.type(IOUtils.keys.TAB);
    IOUtils.type("Eclipse street, #42");
    IOUtils.click(ConfirmButton.jpg);
    AssertionHandler.wait(subscribed.jpg);
}
```

Java Snippet 1 - Procedure 1-C converted to Java

4 - Appendix

4.1 - Showcase

The purpose of this section is to show the potential of this framework by means of cases. Here we list some examples of how Automation Test Framework can be used. With the ability of defining new commands we have the power to represent very different and even highly specific interactions, allowing us to have the most varied test scenarios. By defining and implementing commands specific for each device, several entities can be used in the tests, such as cameras, GPS, circuit boards or even services such as FTP or other web applications.

The diagrams below illustrate the interaction among Automation Test Framework, the mobile application being tested and the entities involved in the different operations.

4.1.1 - Web Applications

With Automation Test Framework, the user can test mobile applications that interact with some sort of web-based service, such as sending an HTTPRequest to register a new user in a website.

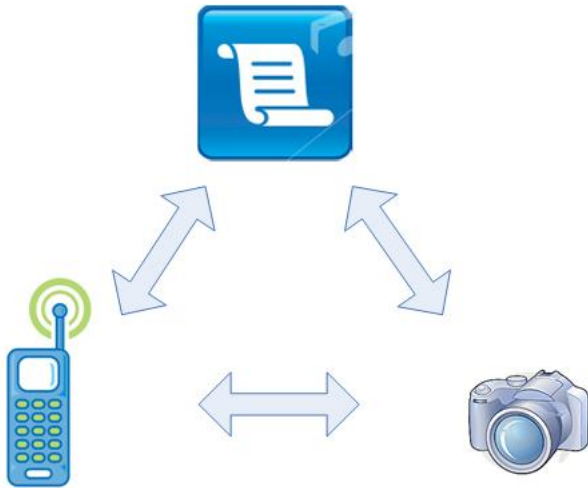


Some example commands are listed below:

```
post(url, httprequest)
get(url, httprequest)
```

4.1.2 - Camera

We want to test an application that interacts with the camera from a mobile device, taking a picture and then sending it via MMS. We could have a command implementation that interacts with a PC camera and displays its content in a Eclipse view. Another option would be working with a simulator/emulator. This way the user could plug a camera to a PC USB port and then test their application as in the actual device.



Some example commands are listed below:

```
takePicture()  
click(aPicture.jpg)  
display(aPicture.jpg)
```

4.1.3 - GPS

We want to test an application which uses GPS data. A possible Test Definition Mechanism could be a view with a map, and a route in this map would generate a procedure like the one described below. This application can interact with GPS, whether by using a stub service or a GPS simulator.

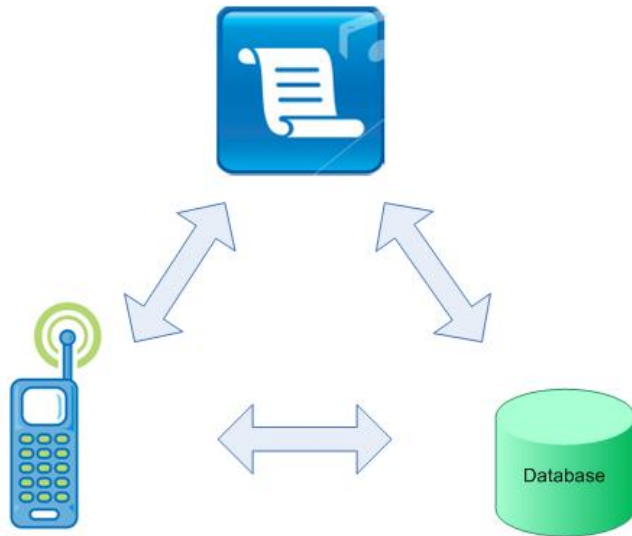


Some example commands are listed below:

```
setGPS(-22.812877, -47.061224)  
setGPS(-22.813075, -47.061074)  
setGPS(-22.813262, -47.060983)  
setGPS(-22.813441, -47.060891)
```

4.1.4 - SQL

The user can define commands for SQL instructions, such as insert, update and commit.



Some example commands are listed below:

```
open(databaseConnection)
insert("John Doe", name)
insert("Eclipse street, #42", address)
commit()
```

4.1.5 - FTP

The user can define commands for interacting with FTP servers and sending and receiving files.



Some example commands are listed below:

```
open(url, port)
```

```
setMode(binary)
put(file)
get(file)
```

4.1.6 - Circuit Boards

The user can test an application that communicates with a prototype board connected via serial or USB ports, sending and receiving bytes or files.



Some example commands are listed below:

```
openPort(portNumber)
readData()
writeData(byteArray)
readData()
display(data)
```

5 - References

- [1] <http://www.processing.org>
- [2] http://en.wikipedia.org/wiki/Test_automation
- [3] <http://en.wikipedia.org/wiki/Gel>
- [4] <http://en.wikipedia.org/wiki/Procedure>
- [5] http://en.wikipedia.org/wiki/Data-driven_testing
- [6] http://en.wikipedia.org/wiki/Modularity-driven_testing
- [7] http://en.wikipedia.org/wiki/Keyword-driven_testing
- [8] http://en.wikipedia.org/wiki/Model-based_testing
- [9] Clayberg, Eric; Rubel, Dan, "Eclipse: Building Commercial-Quality Plug-Ins", Addison-Wesley Professional, June 27, 2004 (<http://www.qualityeclipse.com/>)