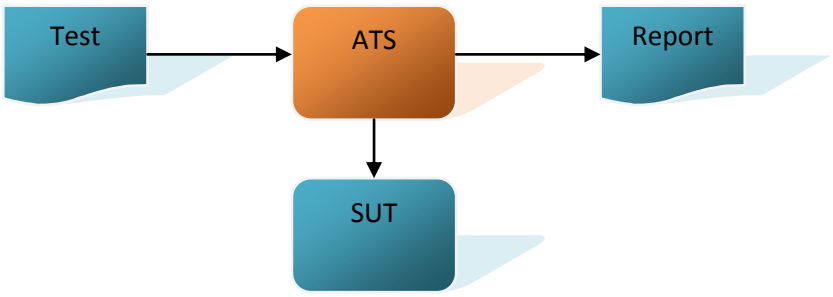
The design of the PowerTools engine

The PowerTools engine is an open source test engine that is written in Java. This document explains the design of the engine, so that it can be adjusted to suit the needs of others. It assumes a basic understanding of automated testing from the reader. The document first explains the basic functionality and some terminology. Then the basic design and the main components are introduced. The remaining sections describe the internals of the main components. To avoid inconsistencies between this document and the code, this document will only contain information that may not be obvious from the code. This includes the overall design and design decisions that span multiple files. For all other things, the code IS the documentation.

The PowerTools engine is the core of the PowerTools test tools. The PowerTools also include interfacing tools that make available other open source tools like Selenium / WebDriver.

The PowerTools engine is released as open source software under the AGPL v3 license. The license details are included with the distribution and are also available at http://www.gnu.org/licenses/agpl-3.0.html. Any comments, questions and requests regarding the PowerTools or its licensing are welcome at powertools@deanalist.nl.

The basics

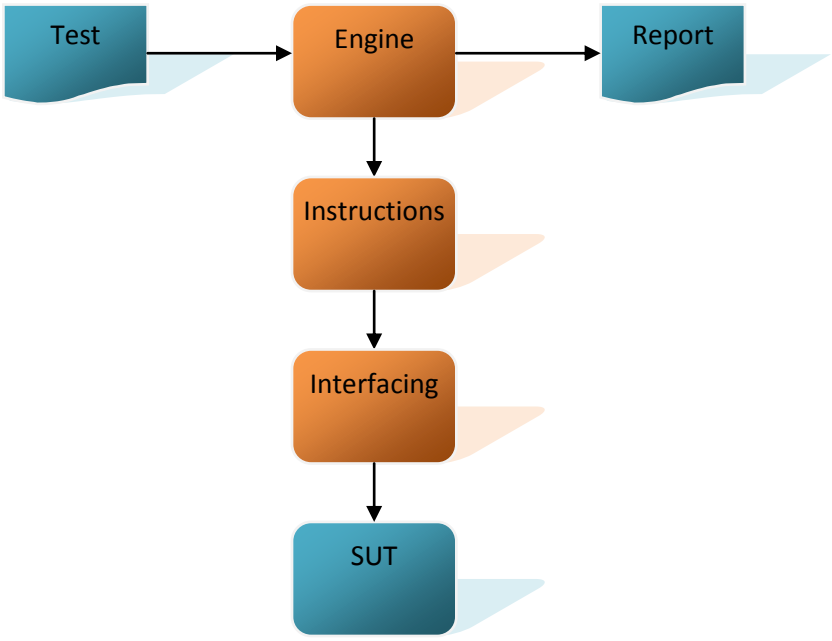


Running an automated test for a System Under Test (SUT) basically involves three tasks: Reading instructions from a test, executing them against the SUT and reporting the results. The complete system that performs these tasks will be called an Automated Test System (ATS). An ATS usually consists of several pieces of software, that work together to perform the functions described above. This software can be divided in three parts: The engine, instructions and interfacing. These will be described in a moment.

Two important terms that need to be defined up front are ´instruction´ and ´test line.´ An instruction is a test command that the engine knows how to execute. This can be a user defined instruction like ´create client´ or it can be a built-in instruction of the engine like ´define constant.' An instruction can have one or more parameters, for instance the client name or city of residence. A test line is a line in a test that refers to an instruction and provides the arguments – the actual values that the instruction will work with. So a parameter is basically the *name* for what the instruction will work with, and an argument is the *actual value* that the instruction will use. An instruction can look differently for different projects, but the concepts of instruction, test line, parameter and argument apply in each case. The examples below show the same test line that enters a client in two forms.

	name	city	email
enter client	John Doe	Amsterdam	john@doe.com

enter client	John Doe	in	Amsterdam	with email	john@doe.com



A test (execution) engine is a generic piece of software that executes an automated test. It reads the test, invokes the instructions it identifies and reports results. It is generic in the sense that it does not need any information about the system that is being tested. Since it does not directly interact with the SUT, it does not need to know what product is being tested, what this product does, what type(s) of interface it has or what technology was used to build it. All this is taken care of in the instructions and interfacing parts of the ATS. As a result, the same test engine can be used for many different systems or applications, with many kinds of interfaces.

The main features of the engine are:

- It reads instructions from a test line source (like a FitNesse table or an Excel sheet).
- It evaluates expressions that appear in instructions.
- It invokes the code that implements the instructions.
- It generates one or more test reports.

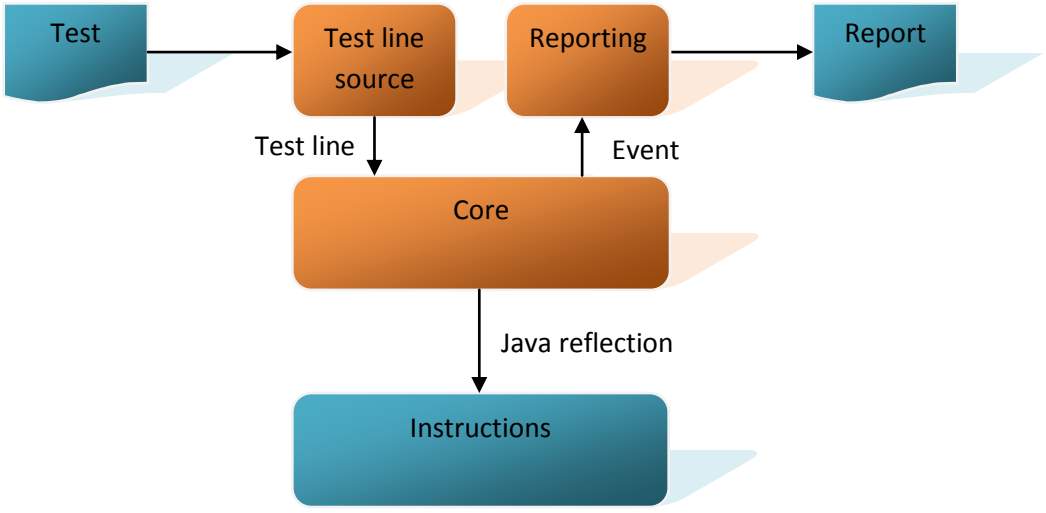
The instructions part of the ATS implements the instructions that can appear in a test. Instructions are invoked by the engine as it encounters them in test lines. The engine needs to know which instructions exist and how to invoke them, but that is all. An instruction can do whatever the tester that needs it wants it to do (as long as that is technically possible). The instructions do not normally access the SUT directly but use the interfacing part for this.

The interfacing part provides access to the SUT, for both actions towards it and for retrieving information from it. Normally, existing libraries are used that either provide direct access or access a tool that takes care of it.

Please note that a number of instructions is built into the test engine instead of the instructions part. These are generic and relatively basic instructions that indicate the structure of the test or support passing data around.

The basic design

The basic design of the engine shows the test line sources, core and reports. Test line sources select the test lines in the input document. They deliver them one by one to the core in the form of test line objects. A test line object contains an instruction name and its arguments. The instruction name in the test line is used to identify the Java method that implements the instruction. This method is then invoked using Java reflection, passing it the arguments from the input document. All relevant events are published in one or more reports.



A test line is made available within the engine through the `TestLine`[1] interface. The first part in the test line, part zero, is the instruction name. The remaining parts contain the arguments.

The events to report are communicated to the reporting component through the `reports.TestRunResultPublisher` class. Most event types are reported by the engine itself rather than from instruction implementations.

A method that implements an instruction is a regular Java method. But in order for the engine to find it, the method must satisfy two conditions: 1) The class that defines it must be registered with the engine and 2) the method name must be a CamelCase version of the instruction name. When looking for an instruction, the engine will first create a proper method name from the instruction name by CamelCase-ing it: An instruction 'enter client address' must be implemented using a method called `EnterClientAddress(...)`. If a registered class has a method of this exact name, the engine will find it and invoke it. The method can have any number of parameters. It will receive the test line arguments in its parameters, and empty strings for all missing arguments (so optional parameters are supported).

Test line sources

The common functionality for test line sources is part of the `sources` package and the FitNesse related code is in the `fitnesse` package. Test lines may be read from any kind of document, including plain text, XML or Excel. The document types that are currently

[1] All references to code are relative to the org.powerTools.engine package.

supported are Microsoft Excel (both .xls and .xlsx format) and FitNesse Wiki pages, but any document can be used as a test line source if there is a class that will parse the file and return its test lines. Empty lines are discarded. A test line source must extend the abstract class `TestSource` and implement two abstract methods: `void initialize()` and `TestLine getTestLine()`. The `initialize()` method is useful for some test line sources and left empty for the rest.

At any time during test execution, more than one test source can be relevant: If one file invokes another, for example, then all test lines from the second file are to be executed before execution of the first continues. This is implemented using a stack of test sources, with test lines always coming from the top one. A test source is explicitly put on the stack when it is invoked and automatically removed when its test lines have been depleted.

Documents can define procedures, or scripted instructions, if the test line source supports this feature. It must then recognize an instruction that starts the definition of a procedure, create a `Procedure` object and add its parameters and instructions. A parameter can be an output parameter, meaning that a symbol (like a variable) can be passed as an argument and assigned to from within the procedure. The core receives the new procedure using the `ProcedureException`. Once a procedure is registered, the core will invoke it when encountering it in a test by creating a `ProcedureTestSource` object and pushing it on the test source stack.

Test lines from data-driven tests, where lines in a test table do not contain an instruction but only test data, are also based on instructions. While the line in the test source does not explicitly refer to an instruction, the table header above it does imply one. So test sources can use that instruction in the test lines they return. As a result, the core only has to process one kind of test lines.

`ExcelTestSource` uses the API provided by the Apache POI project (http://poi.apache.org) to parse sheets in Excel files. POI enables `getTestLine()` to simply iterate over the rows. It must take care to discard any trailing empty arguments, to avoid accidentally passing too many arguments to an instruction. Neither data-driven testing nor procedures are currently supported for Excel based tests.

The FitNesse wiki pages have three kinds of tables that can contain instructions. Each of them is processed by its own ´fixture´ class, a FitNesse concept, that in turn uses its own test source class. The `initialize()` methods of these test sources process the table header. For a `DataSource`, this means determining from the column names the instruction name to use with all test lines it will return. Procedures are supported using the `InstructionFixture` and `InstructionSource`.

Evaluating expressions

The last thing the engine must do before a test line can be executed is evaluating any expressions. An expression starts with a question mark and can contain literal constants, names of symbols and operators. All other relevant code can be found in the `expression` package.

The evaluation of an expression is performed in two stages. The parsers for both stages are generated using the ANTLR parser generator (www.antlr.org). The first stage is

taken care of by a lexer and parser that are generated from the grammar in `Expression.g`. They check the expression for lexical or syntactical errors and throw an exception if they find one. If the expression is syntactically correct, the parser returns an abstract syntax tree (AST) for it, a data structure that represents the whole expression.

The second stage of parsing is the actual evaluation. It uses the tree walker grammar in `ExpressionTreeWalker.g` to 'parse' the AST. The value of any symbols (constants, variables, etc.) is retrieved through the current scope.

If evaluation of any expression fails, the test line is skipped. So any test line that is passed to the core component has all expressions already evaluated.

Reporting

The reporting component is found in the `reports` package. The `TestRunResultPublisher` class publishes the events it receives to all registered report objects. The events have been grouped in four interfaces:

Interface name	***Type of events***
`TestSubscriber`	Test run
`TestCaseSubscriber`	Test case
`TestLineSubscriber`	Test line
`TestResultSubscriber`	Test line result

A report that is interested in the events in a particular interface can implement that interface and register itself with the publisher. This allows a separate report with only the error messages, a log with all details, or any other combination of events.

For the report that FitNesse displays after running a test, the `FitNesseReporter` in the `fitnesse` package determines success or failure for each instruction by keeping track of. The `BaseTestSource` reports the result in the FitNesse table using green or red background and links the instruction name to the relevant part of the HTML report.

The core

The core component receives test lines from the test line source component, invokes the code that implements the instructions in the test lines, and passes various events to the reporting component. It also provides a runtime API to instructions code.

Since every project has its own combination of instructions, test document format and reporting wishes, you have to indicate how you want your engine to work. By subclassing an engine class, you get to select these. One or more test document formats can be supported by creating one or more `run()` methods that select the desired test line source. New types can be added if necessary. The report format(s) can be selected in the constructor by adding the reports to the publisher. Here also, new types can be added if necessary. The classes that implement instructions are specified to the `InstructionExecutors` object so that it can find instruction methods.

Several built-in instructions are provided in the `BuiltinInstructions` class, but these instructions are not made available by default. A specific set of these instructions can be supported by creating a new class with basic instructions (that reuse the existing `functionality`) and registering it.

If it is important that state is retained between calls to the `run()` methods, the engine class should be a singleton class.

When a test is run, the `run()` method of the engine will keep requesting and executing test lines from the specified test line source until they run out. Since the test line source will only return a valid test line, the `execute()` method can look up the method to invoke without further checking of the test line. The `Executor` object it receives contains all relevant information for invoking that method. It also converts the arguments to the types that the method expects (if these are basic types like `int` and `float`), so parsing of (numerical) parameters by the instruction itself is not needed.

Exchanging data

One way to pass data around between instructions is to let an instruction save it in a place that is accessible to other instructions, for instance data members of the instruction class. While such mechanisms can obviously work just fine, it has the disadvantages that 1) it would have to be implemented again and again for many situations and 2) it is not always clear in the test case how data is passed around, which increases the maintenance effort. The engine offers a simple but powerful mechanism that supports the data exchange needs of many situations by offering 'symbols.'

A symbol is a logical name for a value. It is the same concept as that of variables in a programming language, but made available in tests as well as in the code that implements instructions. Some of the built-in instructions that are normally made available in test cases are 'define constant,' 'define variable' and 'set.' These allow a tester to give a logical name to a value that will be used in a test. The value of a symbol is available in expressions. A simple use of this mechanism is making configuration items like URLs available anywhere in the test. A more powerful use of it is returning results from instructions back to the test, as in the simple example below.

define variable	accountNr		
create account	John Doe	john@doe.com	accountNr
send welcome email	?accountNr		

The 'create account' instruction receives a variable name to store the generated account number in. The ´send welcome email´ instruction receives the account number as it was stored in the accountNr variable.

This mechanism works for structured data as well as for single data values. Unlike constants and variables, structures can hold many data items, as in the below example.

define variable	accountNr		

define structure	person		
set	person.firstName	John	
set	person.lastName	Doe	
set	person.email	John@doe.com	
create account	person	accountNr	
send welcome email	?accountNr		

This functionality is not only available from a test, but also from instruction code. The runtime allows creating a symbol and setting and retrieving the value of a symbol. One place where this is needed is the ‘create account’ instruction that will need to set the variable that should receive the generated account number.

The runtime

The core provides internal classes and instruction classes access to some of its basic features using a runtime object. Any instructions class with a constructor with a `RunTime` parameter will receive the runtime object upon creation (using reflection). The instruction methods can use it to report errors, warnings and debug info. The runtime It also supports defining symbols like constants and variables and manipulating existing symbols. To facilitate interaction between instruction sets, it allows sharing objects.