Pathways

Release 0.0.4.dev20200627

unknown

Jun 27, 2020

CONTENTS

Fig. 1: **Automatic product validation during lean & agile development**

README (*Pathways: a test-guide for iterative developed software*)

As Pathways is still under heavy research (and development), the documentation is fragmented. Use the list below to find all/most available documentation. Later, all documentation will be structured nicely.

Whenever you have question, or are willing to

help,
please
con-
tact
me

–ALbert
Mi-
etus

CHAPTER

ONE

CONCEPTUAL

1.1 Pathways: a test-guide for iterative developed software

Pathways is a concept and (reference) implementation for automatic verification (or *testing*) of a product. Especially, during iterative development; like Agile and Scrum. It also fits in a Lean (sw-development) approach.

With this *pathways-framework*, testers can write short Automated Test Scripts ('ATS'es); without being a trained programmer. Those test are Lean, and focus on "user acceptance": Does the product behave as specified/expected.

As the test should be domain ("business") specific, several layers of abstraction exist. Typically the are added by the development-team; party by the testers and party by programmers. The framework itself has only a few generic one. Some more examples can be found in the documentation; which include a workshop/training. (The slides are free).

An important design guideline for the pathways-framework is the (maintenance) cost of all those 'ATS'es; that should be minimised. This (almost) implies, the framework itself is advanced: it allows to be extended with plugins; e.g to interface/communicate with the Product Under Test (PUT), (TsT)Bricks and (TsT)Vectors. This make it possible to combine open-source and propriety parts.

An other goal of Pathways is to provide a (free) "play" and "demo" version of this concept. One can use Pathways to learn this concept, but use a own implementation of it. Several of the later exist (all closed, propriety source).

Pathways was conceived after a challenge of a colleague/friend:

> Build a simple version of it.
> So we can play with it. And learn the concept, *easily*.

For more information, please see the website (under construction), or contact me.

–Albert Mietus

1.2 Concept: *Develop* working code, efficiently!

All software development projects create or modify '*code*'. That (source-) code is either delivered directly or delivered after some automated steps, like compiling. Aside of writing that code, there are many other disciplines and essential steps to make sure that code is working correctly. Any mature team should have knowledge of (domain-) requirements, architecture, designing and testing; to name a few.

There are many way to combine those disciplines. Like the traditional *V-Model*; which often results in a quite linear execution those steps. Or *agile*, which uses a cyclic, incremental approach. In all cases, the basic steps are similar and have the same order, only count and size of each individual steps differs. Verification (test-execution) is ignorant of the process; it is always done after the code is made. And hopefully that code is written after the basic specifications are clear.

Even the most primitive approach will deliver 'code', but to deliver '*working* code' we need more; a lot more actually. To verify the code, tests are needed. But, as a test can only *prove* it isn't working; we have to build trust into test-approach too. Somehow, we have to demonstrate that no failing test implies the code is working correctly. This can result is a lot of additional work: especially when all those test have to be re-run for each code-change. This used to be typical for high-quality software. There, a failed test enforces a bug-fix and so a complete rerun of all test. This repeats until no failures are found anymore. Building that level of trust can be very expensive.

To develop *efficiently*, less work is demanded; as no process becomes cheaper by adding labor! So, automation is needed and a more 'smart' way-of-working. *Integrated Agile* is such a concept; this mix of *Lean* and *Agile* and *Scrum* best practice has shown it is possible to develop working code efficiently.

Pathways is inspired by the *ATS* (Automated Test Script) way of working in *Integrated Agile*. It shares the same concept and is enhanced with a framework to quickly adopt it. However, there is only a weak connection: *Pathways* can be used without *Integrated Agile* And it is possible to use *Integrated Agile* without this framework. Several projects have done so. They implemented their own framework; and still reached there effectivity goals. Actually, Pathways started as a demo, when it became to tiresome to explain the concept again and again to projects.

1.3 Pathways glossary

ATS

ATSes

Automatic Test Script

Automated Test Script An *ATS* is a scripted test to *'prove'* (or *disprove*) a part of the developed *PUT* works correctly. As such it delivers only **one bit** of information: verified of fail. This implies it does verify (check) the results of test.

An *ATS* typically consists of a number of steps, like starting a session, doing several operations, and verifying the results. Each step, not only the verification once, have to be successful to get an 'OK'.

ATSfile

ATStest As a single **ATSfile** can contain several **ATStest**s and both are typically called *ATS*, the suffixes *file* and *test* are sometime used distinguish them.

TsTbrick

brick A (test) **brick** is the basic, *reusable*, unit to build (all) *ATS*es. A *brick* offers a conceptual, *human oriented* set of commands to the *ATS*; so the upper-interface of the brick is *independend* of the technology of the PUT.

Typically, the brick's implementation should also not depend on (PUT-implementation & -interfacing) technology depending; it makes the *brick* better reusable.

There are some brick-like unit for that; see: *cobblestone*, *gate*

gate During the executing an *ATS* all kind of commands and results are passed to and from the *PUT*, by an interface called the *gate*. The implementation of a *gate* is typically specific to the technology of that that *PUT*. At the same time, the role of the *gate* is to hide that technology. Such that *cobblestone*s are less depending on that specific *PUT* and *brick*s not at all.

interface Deprecated since version pre-alfa: Old name for *gate*; do not use anymore

cobblestone A *rough* kind of brick, for special cases. By example to extend a `pathways.puts.put`

A real brick has a functional interface; with cobblestone a second, *lower-level* way to interact with the PUT possible. Which isn't used in most tests, but can be convenient in exceptional cases. It better to have a test, that scripted with to many details, then having no option to automate it.

This is best explained with a simple web-calculator. A (human) user can control it by clicking buttons, reading or entering some text, etc. With a cobblestone that level of control is available for a script. Although, a good test isn't build out of this kind of commands. They use more functional commands, like: 'add two numbers'; a typical brick-command.

With a cobblestone those smaller *steps*, like *clicking*, entering test and reading values are possible. Just for in case they are needed to test the exception.

Secondly, cobblestones are often used to script real bricks. By example the 'add two numbers' bricks-command can be implemented by entering a some strings, clicking a few buttons, and reading the result.

PUT

Product Under Test Depending on the context, the term **put** can refer to the actual product that is tested, or to the proxy-object that represents it.

The proxy-object is typically a subclass of `pathways.puts.put`, or one of it subclasses in `pathways.puts`, which can be extended with *cobblestone*s

feed The **feed** part of a test-vector are used as input for the *PUT*.

expected

expected results The **expected results** (or shortly: **expected**) part of a test-vector specify the values that the *PUT* should return. They are (smartly) compared with the *actual results*.

actuals

actual results This are the values the *PUT* outputs and are compared to the *expected results* to verify correctness.

Note that the compare has to be *smart*; not all parts of the output are relevant. By example an (generated) XML-file typically contains a datestamp; which will vary each run. So that part is typically filtered out

conditions New in version Future: Currently no support is available for conditions. Only *feed* and *expected* can be used now.

Some tests may need more complex validations then comparing the *expected results* with the *actual results* after giving the *PUT* an input *feed*. By example, real-time demands may specify a maximum delay. This kind of conditions can be given in the **condition** part of a test-vector.

TstVector

Todo: TstVector

---

fixture

fixtures

Todo: fixture

1.4 More terms

V-Model The famous V-Model shows the relations between *development*-lifecycle activities. There are several definitions and origins of the model, as well as a lot of *misunderstanding*. All model-derivations have multiple hierarchical levels and are based on three kind of activities: *design* (or definition), *integration* (or test) and *validation* (or feedback). The model itself is non-linear, but most implementations of the V-Model are more or less *sequential* in time, however.

The **V-Model** is an extension to the older *Waterfall model*; which is linear. But many people regard them as the same; probably due the typical sequential execution of the V-Model. That *sequential* execution is often seen as a drawback of the V-Model. And used to advocate 'modern' *agile* software development methods; like *scrum*.

See also:

http://en.wikipedia.org/wiki/V-Model This wikipedia article describes the *generic* system engineering V-Model and variations.

http://en.wikipedia.org/wiki/V-Model_(software_development) There is a separate wikipedia article on the software engineering variation of the V-Model

Agile

See also:

http://en.wikipedia.org/wiki/Agile_software_development

Scrum

See also:

http://en.wikipedia.org/wiki/Scrum_(software_development)

Geïntegreerd Agile

Integrated Agile The is the mix of Lean, Agile and Scrum advocated by Albert Mietus

Lean

Todo: DocMe (Lean) XXX

Pathways A **concept** and (reference) *implementation* to execute (system/user) tests to **approve** the (developing) product is working correctly.

CHAPTER

TWO

USER DOCUMENTATION

2.1 User documentation

This 'book' explains how to **use** pathways; with a focus on the test-user

2.1.1 Running your first ATS

Assuming you have installed pathways and the examples (see: *Installing Pathways*), you can run a quick demo:

```
[albert@pathways:]% cd Pathways/examples/calculators/webapp/tst
[albert@pathways:.../webapp/tst]% pw ATS/demo.py --vector ../../shared/tst-data/telop.
↪csv
Final Test result: OK; after 1 ATS and 1 test
[albert@pathways:.../webapp/tst]%
```

It just works . . .

During this test you will see (all very quickly) that the (Firefox) browser is opened, a *very simple* web-calculator is loaded, and the default user is logged-in. Then several additions are done where each results are verified; that is the core of this test. Finally, the user is logged-out to end the session and the browser is closed. Note: On some systems, you may see a (security) pop-up, in which you should confirm it is allowed this python-application acts as server.

You can simply change this test, by using another *vector*. Copy the file `telop.csv` (dutch voor addition) and add some rows, or change the current numbers. You will find the format easy to understand: two numbers (integers only), an empty column to separate the *feed* (inputs) from the *expected results*: the sum of those two numbers. You can edit the file with any editor, or a spreadsheet-programma, as long as you save it in plain csv! Save the file and start the test again with your vector (say `myfirst.csv`):

```
[albert@pathways:.../webapp/tst]% # Specify the path+file to your own file
[albert@pathways:.../webapp/tst]% pw ATS/demo.py --vector myfirst.csv
```

Fake a bug

Now edit the file again, and put a **wrong** anser in a row; an example is given and the end of this section. This *simulates* a bug in the calculator; but is a lot easier. And start the test again.:

```
[albert@pathways:.../webapp/tst]% pw ATS/demo.py --vector buggy.csv
      adding (2, 5) => got 7, expected: 9 -- see log for trace and details
Final Test result: FAILED (See log for details); after 1 ATS and 1 test
```

As expected, the test will fail. The exact message(s) may differ, but it will report the final-test-result as **FAILED**. Usually, it is also show the operation that found the flow. Here it is expecting `2+5` results in `9`, but got `7`. Surely, this is due to the buggy test-vector, but you got the idea.

While the test failed, and to make it possible (for programmers) to debug it, the *PUT* (the PUT (Product Under Test) is here the web-calculator)` is usually not closed. It typically stops right where the failure occurred! This however depends on the *PUT*, and how it is configured. For now, just close the browser (of play around with this simple app, to get a felling for it)

As the output announces there is more information available: each test-run adds log information to the file `-Pathways.log`; (*yes, it starts with a dash*). This file contains information of every step, of every test, for all runs (succesful or not). The amount of information can be configured with the `log` option. You may (and should) remove those log-files on regulair base, or your disk may fill up quickly. Typically, I remove it after a series of succesful-test runs, or just before I want to dive into the details.

Read the logs

For now, remove the logfile and rerun the failing test with lots of logging:

```
[albert@pathways:.../webapp/tst]% rm ./-Pathways # Use `./' prefix to select the file!
[albert@pathways:.../webapp/tst]% pw ATS/demo.py --vector buggy.csv --log=DEBUG
```

Again the test will fail. Now open the logfile. Even for this simple setup it will contain only around 200 lines, depending on the length of the buggy-test-vector and the line you put-in the fake expected value. As can be expected with *DEBUG*, a lot of details is shown.Every tiny step, in the pathway-framework itself and in the libraries it uses, leave a trail. By example, you will find many lines labeled with `remote_connection`. This lines show in details how selenium *talks* with the browser; for now you can ignore them. As the lines labeled with `pathways.puts.webput` abstract that a bit; on those lines you can follow the communication between pathway's and the *PUT*.

You will find lines the lines we need near the end of the file. For every failing *ATStest* a traceback is shown – just in case you may need it. But *again*, ignore it for now, the lines around it satisfy to signal the failure-cause:

```
## 2016-05-16 00:41:40,199     DEBUG::pathways.puts.webput           webput[199]read␣
↪         # Read: 7, from display
## 2016-05-16 00:41:40,199      INFO::plugin.cobblestones.webMath   webMath[037]__
↪calc        # put=RekenApp[1013db160]; op=+ feed=(2, 5) =>7
## 2016-05-16 00:41:40,199     ERROR::Pathways::.../bin/pw              run[043]run_
↪ATS       # ATS FAILED (AssertionError): adding (2, 5) => got 7, expected: 9
Traceback (most recent call last):
  File ".../pathways/runners/run.py", line 37, in run_ATS
    status = test_fixer.run_with_fixtures()
  File ".../pathways/core/annex.py", line 239, in run_with_fixtures
    rv = self._ats(**parms)
  File ".../examples/calculators/webapp/tst/ATS/demo.py", line 46, in ATS_add
    assert actual == expect[0], "adding %s => got %s, expected: %s" % (feed, actual,␣
↪expect[0])
AssertionError: adding (2, 5) => got 7, expected: 9
```

(continues on next page)

(continued from previous page)

```
##  2016-05-16 00:41:40,201  CRITICAL::Pathways::.../bin/pw                run[044]run_
↪ATS       #       adding (2, 5) => got 7, expected: 9 -- see log for trace and␣
↪details
##  2016-05-16 00:41:40,201  CRITICAL::Pathways::.../bin/pw                 pw[052]main ␣
↪        # Final Test result: FAILED (See log for details); after 1 ATS and 1 test
```

It shows that pathways read `7` from the '`display`', after it did the '`+` operation' on `2` and `5`. Normally that is correct, but the (buggy) test specified it should be `9`. And so, the *test* **FAILED**.

Fix the bug

In daily use the *DEBUG*-level is usually not needed. As you can see in the example above, the same flaw can be found with the INFO-level. And to find failing tests even as 'CRITICAL' level will do. There are several ways to reduce the number of loglines. For now, you can simply **`grep '::[Pp]athways'`** to find all message from the framework (lowercase *pathways*) and the *ATS*es (uppercase *Pathways*).

Now, rerun the test with a lower log-level and find the same flow:

```
[albert@pathways:.../webapp/tst]% rm ./-Pathways
[albert@pathways:.../webapp/tst]% pw ATS/demo.py --vector buggy.csv --log=INFO # or:␣
↪CRITICAL
```

Finally, fix the flaw! And prove it is corrected by retesting it; repeat when needed until the test is **OK**.

The ATS explained

This demo-test verifies the (web) calculator can add-up two numbers. The *ATS* is only a handfull of lines and a few, short, reusable *fixture* functions. This passage shows the full file, part by part; introducing the basic concepts. Details are explained later.

The ATS itself

Each *ATStest* is function[1] which starts with '`ATS_`'[2]. They are (typically) automatically run; in "file order". The *ATSfile* `demo.py` contains one *ATStest*:

```
1  def ATS_add(calcApp, tstvectors):
2      """A basic test to add 2 numbers and verify the result (as demo)"""
3
4      kb = calcApp.widget('Keyboard')
5
6      for feed, expect in tstvectors:
7          kb.clear()
8          actual = calcApp.add(feed[0],feed[1])
9          assert actual == expect[0], "adding %s => got %s, expected: %s" % (feed,␣
↪actual, expect[0])
```

`ATS_add()` is basically a for-loop over all test-vectors. For each *row* the two input-values are feed to the `calcApp` and the returned (*actuals*) value is compared to the (single) *expected* one. When they are :not equal, the test is aborted

[1] Strictly speaking: any callable is allowed.

[2] This `ATS_` prefix is needed for the auto-run feature. All callables starting with this prefix (in an *ATSfile*) are automatically discovered as *ATStest*

(with the `assert` statement). Before each calculation, the screen is :*clear*ed; as we need that button several times, the keyboard "widget" is searched only once; before the loop is :started.

As you can see, the ATS focus on the "adding" and "calculators". The technical details of how the web-calculator works, how "buttons" are implemented, and how to enter (long) numbers is abstracted in generic modules. This make the ATS highly reusable and stable for all kind of changes.

Fixtures & Bricks

There are several kind of generics. The *brick* is the most used once; but not needed for this trivial example.

The other (main) generalisation, *fixtures*, are used: this are functions to setup and/or teardown a test. They are frequently used for many tests. And 'selected' by passing them as arguments to the *ATStest*.

Setup fixtures

A setup-fixtures is (automatically) called **before** the *ATS* is run. It are functions *decorated* with `@pathways.fixtures.Setup`. The returned setup-value is passed as parameter to the ATS.

```
@pathways.fixtures.Setup
def calcApp(cfg):                       ## NOTE: `cfg` is a 'build-in annex'
    """Connect to the selected reken app. And login"""

    Gate, put = cfg['gate'], cfg['put']
    calculator = Gate(put) # connect to the PUT, using the selected GATE (a plugin)

    calculator.loginOK()

    return calculator
```

First, the *gate* to interface with the *PUT* and the adress of the "put" itself are read from the configuration. Then both are used to connect to the calculator; in the demo above this will fire-up your browser and load the `js_calc.html` page, as specified in the `config.cfg` file. Then a standard account is used to login[4]. Last, the now ready calculator is returned.

This fixture is generic: it can be used in many *ATStest*s. Typically these generic fixture are defined in a separate file and imported. Here it is part of the `demo.py`, to show a complete test in one file.

Also the second setup is generic and quite readable.

TstData fixtures

A testData-fixture is a bit like a setup-fixture, but prepares the test data (and/or test-vector). It's a function decorated with `@pathways.fixture.TstData`. It typically returns a `TstVectors`

```
@pathways.fixtures.TstData
def tstvectors(cfg):
    """Read the feed and expected values from a csv file
       FORMAT:    a,b,,c -- where c === a+b """

    filename = cfg['vector']
```

(continues on next page)

[4] The **OK** suffix is a hint this function will abort when the login does fail. As verifying the login-feature is not part of the objective of this *ATS*; it is assumed that will work. Other *ATSes* will test that, using more basic functions.

(continued from previous page)

```
    data = pathways.TstVectors.from_csv(filename, castTo=int)
    return data
```

Again, the configuration is read. Now, to find file containing the test-data; it is the file specified as argument above. That csv-formated file is read, converted to *int*egers, and passed to the ATS.

Note that the setup-fixtures are selected by the ATS. By it arguments, or by the files that are imported. So when a test needs other data (eg floating-point numbers, or read from another source) that is trivial to do: write *one* similar setup-fixture and use that for **all** those tests.

Teardown fixtures

A teardown-fixture is automatically called *after* the ATS (when mentioned as argument). A stand-alone teardown-fixture should be decorated with `@pathways.fixtures.TearDown`. However, as a setup typically also has a teardown those two can be combined; then they are mentioned only once as ATS-argument. For combined-fixtures, the setup is defined normally. After that, the teardown is defined, with same function-name and decorated with `@<FuncName>.Teardown`; using the `Teardown()` method which is "added" to the fixture defined earlier[3]. An example of this is shown below:

```
@calcApp.Teardown
def calcApp(calcApp):
    """Logoff and disconnect"""

    calcApp.logoutOK()
    calcApp.close()
```

After the test in run, the user is logged-out and the connection to the webapp is closed. In the demo above; this last method closes the browser.

Test Vectors

The essential data for an *ATS* is often stored in a csv-file. Notice the empty column between the feed and the expected results.

The demo above used the following file. You can change/extend the file to get a better test. Without coding any line!

Table 1: vector example (used in the demo above)

# (C) Albert Mietus	Part of	Pathways project	Use at will
#feed		;	expected
2	5		7
1	6		7
0	7		7

[3] Reversing the order of the setup- and teardown-fixture is possible too. When the teardown is defined "above" the setup, that one is defined normally. Always the first one should use the `@pathways.fixture` prefixed decorator and the second the `<FixtureName>` one, both with the correct *Teardown* or *Setup* method.

Simulating a bug

This is an example of a testvector with a flaw, to simulate a failing test.

Table 2: **Buggy** vector example

#a	b	;	som	
1	6		7	
2	5		9	#bug
0	7		7	

footnotes

2.1.2 Installing Pathways

Installing the pathways-framework itself is trivial. It is a small pure-python package, which can be installed in any working **python-3** (3.4 or later) environment. Most users will also need to install the examples. That is a bit more complicated; as they depend on other tools. By example, for the webapp-examples, firefox and selenium are needed.

Installing pathways will install the package itself and a few (command-line) tools, like **`pw`**: a driver to start an (few) *ATS*es.

virtual environment

It is advised to use a python virtual environment. There are several options; for a Unix (MacOS, FreeBSD, Linux etc) *virtualenv* (with *virtualenvwrapper*) are good candidates. Whereas *conda* can be very helpful on windows. Read more on the options on:

virtual environment	RTFD
virtualenv	http://virtualenv.rtfd.org/
virtualenvwrapper	http://virtualenvwrapper.rtfd.org/
conda	http://conda.rtfd.org/
(py)venv	*pyenv is part of python, from 3.3*

Note: Python3 & pip3 ?

The commands below have the **3** suffix, to denote one have to use a python-3 (virtual) environment; and the pip-3 version too. Depending on your environment, you may need to drop the *3*

- When using pip as `python3 -m pip`, the pip as argument has no suffix!
- Below, we use the `--upgrade` argument (always); this make sure an already installed package will be upgraded.

Installing the framework

The R&D distribution of pathways can be obtained from Bitbucket. You can clone the repro, or download the source, there to get a "source" distribution. It is also possible to use a *dist-file*, named like `pathways_framework-0.Y.Z.devDATE.tar.gz`. Currently, that one is only hand-out by Albert (during trainings, etc)) but will become downloadable from Bitbucket soon.

Once a *real* "production" version will become available; then will become available on pypi too.

In all cases, pathways can be installed with pip.

dist-file

Installing the pathways-framework from the dist-file is easy:

```
[albert@Python3:]%  pip3 install --upgrade pathways_framework-0*.tar.gz
```

From source

Installing from source is a bit more complicated, as one has to select the correct branch first; currently that is `dev_ATS.fixtures` (but that will change!). And that install the "directory". Assuming you are the top directory, where the file `setup.py` is located, use:

```
[albert@Python3:]%  hg checkout |branch| ## Select the 'correct' branch
[albert@Python3:]%  pip3 install --upgrade --editable .
```

A 'source-install' is typically installed *editable*, as is done above (with the `--editable` option). Now, the (local) pathways sources can be edited, with direct effect. This is an advantage for developers (of pathways).

Todo: make pathways downloadable

For now: Download the source from Bitbucket!

Tips

- Add python3 to your path (http://stackoverflow.com/questions/3701646/how-to-add-to-the-pythonpath-in-windows-7)

2.1.3 Installing the examples

When the examples (for the tutorials) are *not* needed you can skip this article.

2.1.4 WebApp examples

For the 'webapp' examples a browser is needed, which is automated with 'selenium' (which is part of the *gate*). Several browser are supported, like *Firefox* and *Chrome*. For each browser-brand (& version), a specific browser-driver should be installed; aside of the generic selenium-software and the python (language) bindings.

The generic *webgate* is part of the framework, but should be configured to the selected browser and selenium-driver. And to compete, the examples themselves have to be installed[1].

Warning: Generic selenium software?

The documentation of selenium and related topics (bindings, drivers) is very-outdated.

> Note to the Reader - Docs Being Revised for Selenium 2.0!
>
> —http://www.seleniumhq.org/docs/00_Note_to-the-reader.jsp, Oct 2, 2016

Whereas the current version of selenium is 3.0.0-beta4. And the current firefox-browser (v49) is not compatible with older selenium versions.

At the moment it unclear if (or when) the "generic selenium software" is needed. Most probably, it is not (anymore). Or it comes automatically with the selenium-python language-bindings.

Whenever you installed as described below and it does not work, please install as described on http://www.seleniumhq.org/download/ And let me know, including details as symtoms and versions. –Thanks

Selenium

Tip: Verify script

Pathways comes with a small script (`.../bin/VerifySelenium_Firefox.py`) to verify Selenium is correctly installed, including browser, drivers, and language-bindings; currently only for Firefox (this will change soon).

Run it like:

```
[albert@Python3:]%  python3 {options} VerifySelenium_Firefox.py PATH/TO/DRIVER

It will step-for-step verify all details and give tips when it fails. When all works, the browser is started and
one page is visited. You can visually verify this. At the end, that browser (session) is closed.
```

To show it options: run:

```
[albert@Python3:]%  python4 VerifySelenium_Firefox.py --help
```

- Install the selenium-python bindings:

  ```
  [albert@Python3:]%  pip3 install --upgrade selenium
  ```

[1] That is: download and unpacked. When the "source" distribution is downloaded, the tests are already available (in `pathways-r-d/examples/calculators/webapp/tst/`).

Browser & driver

Firefox

- Install Firefox
- The *Gecko* driver: https://github.com/mozilla/geckodriver/releases

Chrome

- Install Chrome
- Install the ChromeDriver: https://sites.google.com/a/chromium.org/chromedriver/downloads

Other (browsers & version)

The procedure for other browsers is described in the selenium documentation. As browsers and selenium tend to change frequently, is sensible to have a look on that site, as well on stackoverflow.

configure & WebApp test

Todo: make browsers & driver selectable

Warning: Currently, the config files (and even some code!) contain hardcoded paths. That is a known error. . . .

- But as I am the main user, it will do for now. :-)
- A rewrite/update of the `core/conf` is needed.

- download and unpack the examples

  Todo: Make the examples (zip-file) downloadable, and document it.

2.1.5 How to run ATSes?

There are several ways to run *ATS*es. There is no *best* way, that is very depending of the goal and the project. This chapter gives an overview of the options; so you can select the one that fits you best. But remember, you can easily add or combine options.

Using `pw`

Pathways comes with a tool *pw (1)* to run one or more *ATSfile*s from the command-line.:

```
[albert@pathways:tst]%  pw  ATS/demo1.py
...
Final Test result: OK (after 1 ATS)
[albert@pathways:tst]%  pw  ATS/demo1 ATS/demo2.py
...
Final Test result: OK (after 2 ATSes)
```

As the configuration is read before the *ATS* is loaded, the *ATS* may use import modules (like plugins) from non-standard locations, without setting `PYTHONPATH`.

Tip: The `.py` extension in the named *ATS*es are optional, as shown above

---

Use `pathways.autorun()` *(run the ATS itself)*

Most *ATS*es can be started by executing that file. Only the *ATStest*s in that file are executed.:

```
[albert@pathways:tst]%  ./ATS/demo3.py
...
Final Test result: OK (after 1 ATS)
```

Typically it is called from the `tst` directory; to make sure all relative paths are correct.

Attention: To enable this, each *ATSfile* should call `pathways.autorun()` as main routine. As shown in the following code fragment

```
if __name__ == "__main__":
   exit(pathways.autorun())
```

To make sure the exit-code is correctly set, the `pathways.autorun()` should be wrapped in `exit()`

Interactively in Jupyter (IPython) Notebook

It is possible to develop and/or run ATS in Jupyter Notebook (or directly in IPython).

See also:

For now, see http://localhost:8887/notebooks/Pathways/RunATSes_from_Notebook.ipynb

---

Todo: This runner need work; the link above is a HACK

More to follow

There should be options to integrate running the *ATS*es within other frameworks, like:

- cronstab
- nightly build
- continues integration tools
- build tools,
- etc

During such an automatic-run, it should be possible to select tests, determine the order and to determine how to continue after an failure.

This work to be planned. For now, some simple shell (or python) script will do.

2.1.6 Tutorials

2.1.7 Manual Pages

`pw` (1)

A helper script to start one or a few *ATS*es. Typical used on the command-line.

`--vector` `<VECTOR>`
: Specify the testvector(s) to use

`--config` `<FILE>`
: Specify a config file. When not given, a file file the same name as the ATS, but the extension `.cfg` is used

`--log` `<LOGLEVEL>`
: Set the loglevel to get more (or less) log/trace information. Typical this option is set in the *`pw --config`*, but often overiden when running one or a few test; eg. during debugging.

`ATS ...`
: The *ATSfile*s to run. At least one *ATS* is need. When several files are given, they are run in order (as are the *ATStest*s within one file)

pathways config files (5)

Pathways uses multiple 'ExtendedInterpolation' ini-files for almost all configuration; the command-line configuration is an apprehend exception. Typically those files use the *.cfg* extension (not 'ini'). When its basename is equal to the (basename of the) *ATSfile*, it is automatically read and used.

Like other ini-files, pathways cfg-files are sectioned and typical start with a `[DEFAULT]` section. Lines starting with an hash (`#`) are comments and are ignored.

Configurations are stored as a `key=value` format. Both *key* and *value* may contain spaces. However, spaces in keys are not recommended! Spaces around the '='-sign are ignored. Also other leading and trailing whitespace is skipped.

Although the 'key' is case-insensitive, configuration-keys are typically in all-lowercase. Auxiliary keys use mixed- or upper-case. Auxiliary keys are used build values, by interpolation (or expanding) them with `${AuxValue}` in another value.

Config files

Typically, the configuration is combined from several locations.

- When the option `--config <FILE>` is used, only that file is used and all other locations are ignored.

1. A user/global configuration-file called **pathways.cfg**.

   Typically found in the users *HOME* directory, or the directory set by the environment `PATHWAYS_CONF_DIR`, or runner-specific configuration.

2. The shared configuration-files called **config.cfg**. This can be found in several places:

   a) the working directory (where the tests are started), or

   b) the directory where the *ATS*es are located, typically also called `ATS/`.

   Note: currently only 1 file is used: (b) when an ATS is specific, else (a).

   Todo: Fix this

3. The ATS specific file *basename*`.cfg`, in the same directory as the *ATSfile*

Well known configuration-keys

log Threshold for logging. This can be a number between **0** and **50**, or one of the standard names. The lower the number, the more is logged.

- CRITICAL (=50)
- ERROR
- WARNING
- INFO
- DEBUG (=10)

gate

Todo: Redesign the old 3-part gate and document it.

put An URI-style adress of the *PUT*: which product is to be tested. The exact interpretation of the value depends on the *gate*

Todo: extent the list

Custom config-keys

2.2 FAQS

2.2.1 About the workshop (Dutch)

workshop: Aan de slag (FAQ; tips)

language Dutch

Waarom zijn er zoveel dirs/files.

Pathways is volop in beweging; het wordt steeds beter. Zowel de `pathways` module, als de documentatie, de voorbeelden en de workshop. Ook worden regelmatig alternativen *uitgeprobeerd*. Al die files staan in één project: *Pathways*. Maar gelukkig is de structuur vrij regelmatig en makkelijk te doorgronden; zie *TODO*.

Todo:

label *_pathways-file-structuur* plus inhoud maken

En waar begin ik?

Waar start ik de ATS'en op?

Alle *ATS*'sen staan in (een subdirectory van) de directory `ATS/`, die weer in een `tst/` directory staat. Het is belangrijk om de *ATS*'sen op te starten in die `tst/` directory; *niet* in `ATS/`.

Ten eerste kommen daardoor de (standaard) log-file (`-Pathways.LOG`) in die directory te staan. Dat is handig

Maar belangrijker, sommige *ATS*'sen passen het import-path (tijdelijk) aan, bijvoorbeeld om een andere *ATS* als soort van brick te gebruiken. Omdat dit relative paden zijn, is moet de *werkdirectory* constant zijn: `tst/` dus. Dat is wellicht niet altijd even *netjes*, maar tja ...

workshop: Doelgroep (FAQ)

language Dutch

Is de workshop (alleen) bedoeld voor "aankomend testers"?

- Nee; de doelgroep is veel breder: het concept kom uit Lean/Agile/Scrum (Geïntegreerd Agile) aanpak. Iedereen in zo'n ontwikkel/projectaanpak is nodig cq welkom.
- Vooral (ervaren) testers, om ervaring op te doen met moderne automatisch testen, zoals steeds meer gebruikt zal/gaat worden in Lean/Agile/Scrum ontwikkelingen.
- Ook programmeurs zijn wel welkom; zeker die (gaan) werken in een agile omgeving. Waar zij de testers moeten "ondersteunen" door het framework te bouwen/onderhouden/aanpassen
- Er zijn ook oefeningen, die (uitsluitend) gericht zijn op (die) programmeurs

Is *selenium* belangrijk voor 'Pathways'? Of voor de workshop?

- Pathways is **onafhankelijk** van selenium; maar kan die tool als library gebruiken.
- De tester/gebruiker/workshop deelnemer hoef niets van selenium te weten!
- Voor een van de oefeningen wordt (onderwater) selenium gebruik; daarom moet dit (vooraf) geïnstalleerd worden.
- Andere oefeningen werken zonder die "library".
- Of selenium (in het echt) gebruik wordt, is sterk afhankelijk van *PUT* (het product dat ontwikkeld/getest wordt)
- Voor "webapps" kan selenium gebruikt worden, of de browser aan te sturen; er zijn echter alternatieven

Hoeveel Python (kennis) is nodig?

- Ook de tester heeft (beperkt) kennis van Python nodig
- Elke ATS (de test) is geschreven in python. En wordt (meestal) geschreven door de tester
- Maar ATS'sen zijn kort een eenvoudig; ook voor niet programmeurs!
- Een ATS maakt gebruik van 'bricks'; die zijn ook in Python geschreven; maar complexer en vaak door programmeurs gemaakt

Globaal zijn er 3 a 4 "lagen", die een ander niveau van Python nodig hebben:

ATS:

- Korte python scripts, die basis Python-vaardigheden [PY1] vragen.
- Vooral verstand van test(ontwerpen) is nodig

Bricks:

- Iets complexer, soms complex. Meer (algemene) programmeerkennis nodig dan de meeste tester hebben.
- Maar voor programmeurs met een beperkte Python opleiding goed te doen

Gate/config:

- Meer Python & programmeer kennis nodig. Maar is slechts éénmalig nodig per PUT
- Nauwelijks/geen kennis van testen nodig

Pathways/Framework:

- Dit is de core; het ontwerpen & programmeurs hiervan vraag "veel" python en programmer kennis. Het meeste van deze 4 lagen.
- Eigenlijk is dit is een "apart" project/product

In de workshop zijn de **testers** vooral bezig met de ATS-laag, en beperkt met de brick-laag.

Voor **programmeurs** ligt de nadruk op de bricks en gate/config lagen. Gezien het workshop gehalte (en dus korte duur) is dat wel beperkt.

Waarom maakt 'pathways' testen (en product-ontwikkeling) efficiënter?

Door het pathways-concept, kunnen de (alle) *testers* zich concentreren op het ontwerpen van de test(en) ipv het uitvoeren. Bovendien kunnen ze dat lean (efficiënt) doen; omdat de nadruk op (met name het 'schrijf'-deel) van traditionele tools/aanpakken minder is. Voor "eenvoudige" tests (ook afhankelijk van test-ervaring) kan wellicht meteen de ATS geschreven worden. Voor lastigere zaken is die traditionele kennis/aanpak natuurlijk nodig. Maar vaak is de documentatie (& tool) druk minder.

Zo wordt is het 'nooit (zeg nooit nooit) verplicht' om MS-Visio (of soortgelijk) om een stroom-diagram te tekenen. Vaak/soms is dat 'waste' en dus niet 'lean'.

De flow ligt vast in de ATS; en dus kan die "flow" direct in het ATS "geprogrammeerd" worden.

Al is een kladje als voorbereiding, soms wel handig. Dat kan op papier of whiteboard. Maar uitwerken is niet nodig; het resultaat (de ATS) is (lees: moet!) duidelijk genoeg zijn.

Ook (vooral) het *programmeer*-deel van een project wordt efficiënter. Omdat de testen snel (parallel; of soms eerder dan het ontwikkelwerk) beschikbaar zijn, heeft de programmeur een soort van valnet. Door het veelvuldig uitvoeren van alle testen, weet hij snel of hij geen bestaande features "kapot" gemaakt heeft. Maar ook in hoeverre de nieuwe features al werken. (STTD: Test Driven Development, of Systeem niveau)

2.2.2 Versus: What to use X or Y?

Some concepts –*and/or classes*– look simular, but still are differed. This part of the FAQ explains them.

Cobblestone vs Widget

When a gate to a WebApp is made, one can use both `Widgets` and `Cobblestones` to *define* the `Put`. But what is the difference?

Currently, there is a trivial difference: A `Cobblestone` is available for all kinds of `Puts`, whereas the `Widget` is only available for a `WebPut`.

Note: This may change in a future release however, as there is nothing "Web"-specific to a Widget.

- A Widget is kind of a PUT. It not really a *PUT*, but it can be considered as mini (or partial) PUT. Whereas the *Cobblestone* is more like a *Brick*: a box with functions that can be called and work on te PUT (or Widget).
- The functions within a cobblestone are (typically) automatically available for the PUT; the user (the ATS, or a real brick) doesn't need to know in where it is located:

```
# The add() function, located in the "math" cobblestone, add can be used directly
calc.add()
```

- A Widget has a name. Which should be use explicit before the functionality of that part ca be used:

```
# To login, you have to select the "session page" (a Widget) and login there
ses = calc.widget('Session')
ses.login()
```

- Both a Cobblestone and a Widget are objects; so the can save state in `self`.
- Both are automatically instantiated during the creation of a PUT.
  - a widget can look for its gate via `self.parent` (currently the parent is always the PUT, but that may change, as widgets can contain sub-widgets

- a cobblestone can access it PUT by `self.put`, directly.

Bricks vs Cobblestone

A *Brick* is closer to a tester then an *Cobblestone*. A test is build out of steps, which itself are build of smaller steps; by combining those small steps in a logical "function"; the *Brick* becomes reusable. But principally every tiny step within a brick could be done oen for one; manually if needed. The steps in a *Brick* a conceptual, those in a *Cobblestone* more technical, and closer to the *PUT*.

By example: To add-up two numbers on a calculator, one has to enter the first number, the '+' operator and the second number, etc. However, entering a number has smaller steps: one has to press a button digit by digit! That kind of detail is not essential is most tests. And hidden by "automating" it.

How to press button for butten exactly, is closely related to the *PUT*: on a web-calculator one has to move the mouse around and "click" on the GUI-parts that represent the keys. But on a old-fashioned desk calculator, one really has to move physical button. Besides, pressing those buttons is not related to the "add-up-test". For all other calculations it is exactly the same. So, that part is usually automated inside the *gate*, with *Cobblestone*s.

But automating how to perform an addition-test is quite generic: add the two numbers and verify the result. Typically that result does not depend on the *PUT*. When a tester needs to automate it –so (s)he can use it a a step in a bigger test– it is normally done in a *Brick*.

Note: This is no exact border where to "automate it"; it more an art (or stye) then a science.

The "automation" of splitting a number into digits can be seen a *generic*, for any brand of calculator, both physical, as wel for GUI- and web-app.

At the same time, it is *specific* and closely related to "calculators". After all, the same add-up test can be used to tests "computing servers". They typically require the complete number (or more), in a xmlrpc, SOAP or other message. When one what to both xmlrpc-calculators and web-calculators, one can decide that is all very specific to the web-calculator and put most "automation" inside a :term: calculator.

Tip: One should stive to a balance. When everything ends up at the same place and that become messy, brick it up into the available layers. At the same time, when a simple thing is divided into several places, it there is no cohesion, and once should collect it to 'the right place'.

An pragmatic seperation between *Cobblestone* and *Brick* is "conceptual accessibility". All (tiny) steps within a *Brick* should (conceptually) be possible from a test(er) point of view. Within a *Cobblestone*, there is more (technical) implementation freedom. So, when a 'internal function' or constant is needed within the *PUT* or: *Gate* is needed, it should be implemented in a *Cobblestone*!

Caution: The examples which come with Partways are small! But they use a variety of solutions and layers to introduce (show, teach) them. But don't follow them as an example to brick everything up into very small chunks.

2.3 Training (slides) *OLD*

2.3.1 Pathways Workshop

status archived, old

Automatic Verification in 'Geïntegreerd Agile'

Pathways a **concept** (and reference **implementation**) to *execute* tests to **approve** the (developing) *product* is working **correctly**.

More

- This workshop is part of the *User documentation of Pathways*
- The "html-slides" (hieroglyph) are no longer supported

- Essential for iterative/ & interactive (software) development
- Part of 'Geïntegreerd Agile'; my mix of Lean, Agile and Scrum
  - But surely not the only option, nor part

This workshop

1. Learn the *ATS* & *Pathways concepts* ATS: Automatic Test Script
2. Test some *calculators* A small toy project
   - Play with it
   - Write real tests! 80% testing, 20% Python
3. Adapt the framework Learn to use it in real projects
   - Write *brick*s for reusable tests 45% testing, 55% programming
   - Write plug-ins to facilitate testers 10% testing, 90% programming

2.3.2 Training overview

Some concepts

Why? & Concepts

- Why **Automatic Test Scripts**? Why a **framework**? Why the **Pathways**-framework?
- An overview of Pathways (bricks, TstVectors, ATS, . . .)
- Pathways is for **testers**; but *programmers* can/must help It's a team effort
- A bit of history

Automatic verification

Tip: Testing is activity, an profession, but not a *goal in itself*.

- We need a product *(a software programma)* that works, as we agreed. Or better: as we **aspect**.
- Even when features are added, insights are changed and demands do change, it **should work**!

—some customers

"Lean, Agile, Scrum", effects on testing . . .

Due iterative development		test-execution may become repetitious.	*And so, very expensive*
As requirements do change		the test itself needs to be updated 'a little'	*But quite often*
Programmers and testers sit in 1 team		they may/can/should help each-other!	*This is an opportunity!*

Benefits of Automatic Test Scripts

The *new* tester

A tester is not needed to detect flaws *afterwards*; (s)he is needed to **prevent failures** in the *first place*.

So **thinking** about 'sound' test is essential; not the programming of an ATS. Typically, that is less work then writing the same script in a Word-document.

- Cheap to execution. Verification can be done **daily** Signals when '*development is done*'
- Mostly shorter as ½ A4tje. **Cheap** to write and maintain Focus on '*good tests*'
- Store in **version controll** Expliciet, what is tested, when No need for *lengthy documents*

ATS vs Python vs framework

A tester should focus on *designing* sound tests. Not on executing them, nor on programming them.

Each ATS should 'read' as a classical (manual) test-script

1. First login
2. REPEAT
    1. Enter two numbers and the 'plus' symbol
    2. Read the result on the display
    3. Verify that value is the same a the pre-calculated one Essential: not just a number, the expected one!
3. *Log-off* Only when all steps can be done (and 'verify OK'), the test is OK

Reuse 'bricks', focus on 'domain'; hide technology

Requirements

This kind of *lean*, *agile* and **'KISS'** demands are used as requirements for Pathways.

A framework that isn't easy to use (by testers)

- To be able to make an *ATS* that reads as above, a lot of "programming" en "technology" should be concealed (in the *framework*)
- As many steps contain details steps, that are needed *again-and-again*, those **"bricks"** should be reusable

Aside for 'hiding', the framework and the *brick*s also *centralize* details. A change in the product (eg how to login) should not result in updating many ATS'sen. Only a **one** (small) change, in the corresponding *brick* (or a part of the framework) should be needed!

Testers and developers

But feel free to use it elsewhere Pathways is primary meant for teams that *create working software*; so where 'programmers' and 'testers' work together. And it makes use of the skills of that team

Testers

- All testers should be able to read/modify/understand the (python) test-scripts
- Most testers should be able to write *most of the* the (python) test-scripts
- Only some testers should be able to read/write some of the *brick*s; as this is more technical/complex

Programmers

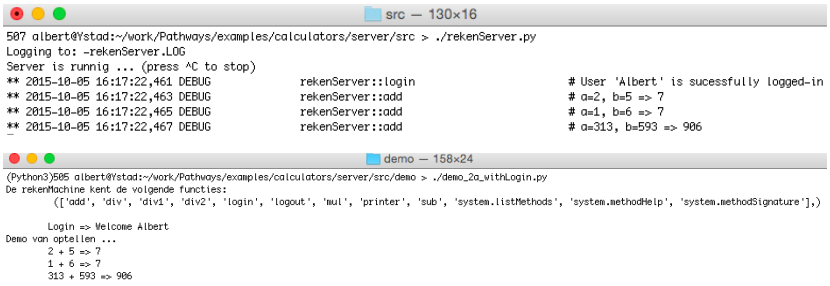
- Typically, **do not** design/write the test-scripts; but can *assis* to code/program them
- Write/Modify those *brick*s the testers need specially the test-API should be requested by testers!
- Maintain the *gate*, the (technical) interface between the framework and the product-under-test (*PUT*)

Overview

— A global overview of the Pathways concept and parts

History

Some calculators, with ATSes



There a two calculator examples

1. A *WebApp*: A html-page, to run in your browser
2. A *xmlrcp* server: a deamon to offload your client

Both have (almost) the same, limited functionality

- Add, subtract, multiply, divide; mainly (positive) integers only
- Both requerer to 'login' first!

Although these 'products' are almost trivial, and have a useless 'login' feature; they are great examples to show the Pathways concept.

XXXXXX

Workshop info

Status

In development

Compulsory knowledge

- **Python** (basic). As all tests, as well as the bricks and the Pathways framework itself is written in Python-3
- Testing and/or programming skills and experience (basic)

Pre-Installed tools

- Python-3 [3.4.1]
- Firefox browser [36.0.4; 36.0.4]
- selenium [2.46.0; 2.47.3] *python bindings to selenium/Firefox*

Pathways [Alpha-2015.09] is needed, but not yet downloadable yet.

Ask ALbert

Workshop Pilot/Help

Goals

- Get feedback/expertise for the real workshop
  - Which (general, background) slides are needed?
  - Complexity **test-exercises**; how much time do they need? How much are needed?
  - Same for **bricks**, **gates**, **plugins**, etc.
  - Tempo, "Prior Art", Topics, ...
- Support to get the Pathways framework up-and-running
  - Port/Install on **Windows** (developed on MacOS)
  - Human interface (1^{st}: cmd-line; optional: GUI, IPython/Jupyter, ...)
  - ...

I'm open for suggestions

2.4 General:

2.4.1 Examples

WebApp example

The plugin

This includes the gate: `plugin.gate.rekenApp`, with Widgets and some *cobblestone*s.

pyreverse diagrams:

classes

packages

Some local bricks (can be empty)

pyreverse diagrams:

classes

packages

And the ATSes

Note: the generated docs from the ATSes aren't very great. Read the ATSes themself, they are meant to be readable!

pyreverse diagrams:

classes

packages

2.5 In Dutch (Nederlandstalig)

2.5.1 *Pathways* demo: Het testen van diverse rekenmachines

Note: The documentation of this example is in Dutch only

Bij een iteratief (software-) ontwikkelproces (zoals *Geïntegreerd Agile*, *scrum*, e.d.) is het belangrijk om regelmatig te testen of het product nog steeds werkt. Zowel nieuwe als bestaande features moeten geverifieerd worden. Ook moeten bestaande testen soms aangepast worden, aan de steeds veranderende specificaties.

Concept & framework

Pathways is zowel het concept *"Hoe die ATSsen te gebruiken"*, als een (eenvoudig) framework om die ATSsen effectief te ontwikkelen.

Het gebruik van dit framework is niet essentieel; diverse projecten hebben het concept gebruikt lang voordat het *pathways-framework* ontstond.

Het idee voor een framework is later ontstaan. Uit de behoefte om het concept uit te leggen, zonder telkens weer bij nul te beginnen, was een demo nodig: het testen van een rekenmachine. De gemeenschappelijke code om die ATSen te laten werken was de voorloper van het framework.

De vraag: *"Hoe meerdere rekenmachines te testen met dezelfde ATS?"*, en een demo daarvan leidde tot de allereerste versie van het framework.

Hiervoor zijn Automatische Test Scripts (*ATS*sen) essentieel, evenals een concept om die eenvoudig en effectief te maken en onderhouden. En die ATSsen moeten bruikbaar en begrijpelijk zijn voor niet-programmeurs! Een tester moet immers vooral nadenken over het testen; niet met het opschrijven hiervan; noch in (*Word*) documenten, noch in code.

In dit rekenmachines-voorbeeld testen we een paar rekenmachines. Al zijn ze geheel anders ontwikkeld, veel requirements zijn gelijk: 1+1 is altijd 2!. En dus kunnen we veel testen hergebruiken. Dat laat zien hoe een ATS onafhankelijk van technische- cq implementatie-keuzen gemaakt kan zijn.

OLD

Tijdens het (agile) ontwikkelen van een product zal dat product telkens anders worden. Soms komen er features bij, soms moet bestaand gedrag veranderen en vaak zullen stukken 'code' herschreven worden. Toch moet het product blijven werken. En mag het testen daarvan niet te veel tijd kosten.

Een test moet dus herbruikbaar zijn en blijven werken, ook als de ontwikkelaars andere keuzen maken. Elke ATS zal dus moeten abstraheren van technische details.

Doordat Pathways abstraheert van de implementatie(techniek), zijn testen herbruikbaar.

Het is dus mogelijk om meerder

Het Pathways concept abstraheert van implementatie (de techniek) van het te verifiëren product. Ofwel, een goede *ATS*(set) kan meerdere producten testen, als die (bijna) gelijke features hebben.

Dit laten we hier zien door een aantal eenvoudige rekenmachines te testen. Het optellen van twee getallen geeft altijd hetzelfde resultaat. Met één ATS kan dus zowel een (javascript) **webapp**licatie, een (xmlrpc-) **server** als andere soorgelijke tools getest worden.

Demo: RekenServer

De **RekenServer** is een soort van heel eenvoudige rekenmachine. Uitgebreid met bijzondere features, zoals inloggen en printen, om ook het testen van dit *gedrag* demonstreren.

Een typische sessie verloopt als: inloggen, sommen uitrekenen, die berekening printen en weer uitloggen.

Overzicht

- *Specificaties*

Specificaties

De specificaties zijn zeer beperkt en in een paar korte sprint te realizeren. Het doel is immers om het pathways concept te demonstreren.

Berekenen

- Alleen getallen tot en met 100
- Alle operaties zijn op 2 getallen
- Alleen *add* (optellen) is vereist
- Nice to have: *subtract*, *multiply* en *divide*

In- & Uitloggen

Inloggen gebeurd met een naam en een wachtwoord. De naam mag niet leeg zijn, en voorlopig moet het wachtwoord `GeHeim` zijn. Verdere controle is niet nodig

De RekenServer mag **geen** functionaliteit (anders dan inloggen) bieden zolang men niet ingelogd is. De gebruiker is verplicht om uit te loggen. Inloggen als kan niet als al er al iemand ingelogd is.

Printen

Ook printen is zeer eenvoudig; er wordt niets naar de printer gestuurd. Doel is demonstreren dat de RekenServer alle berekening onthoud zodat die later geprint kunnen worden.

Nu moet, als reactie op het print-commando, een eenvoudig tekstbestand gedownload worden. In dat document moet de gebruikersnaam staan, de datum en tijd van het print-commando en alle berekening tot nu toe. Zoals in onderstaand voorbeeld:

Voorbeeld van een print-out

```
User:        <naam>
Timestamp:   <datum en tijd in ISO notatie>
<lege regel>
Berekeningen:
    <getal1> <operatie> <getal2> ==> <uitkomst>
    <getal1> <operatie> <getal2> ==> <uitkomst>
    <etc>
```

CHAPTER

THREE

THE FRAMEWORK ITSELF

3.1 Framework documentation

3.1.1 Pathways: a test-guide for iterative developed software

This Pathways-framework is a reference implementation for effectively verifying (or *testing*) the correct behaviour of a product. Especially during iterative development, like scrum or *'Integrated Agile'*, the developed product changes constantly, as does the features (and other requirements) of the product. So, it is vital to constantly verify all existing and new features. This has to be automated else the load of manual testing will grow quickly above the capability of the team.

Also the maintenance of all those *Automated Test Script*s (or *ATS*) should be minimal. This is possible by using known software-design-principles, like locality.

3.1.2 Overview

The two images give an global overview of the pathways-package; they are automatically generated from the code with *pyreverse*

UML class diagram

UML packages diagram

3.1.3 Design

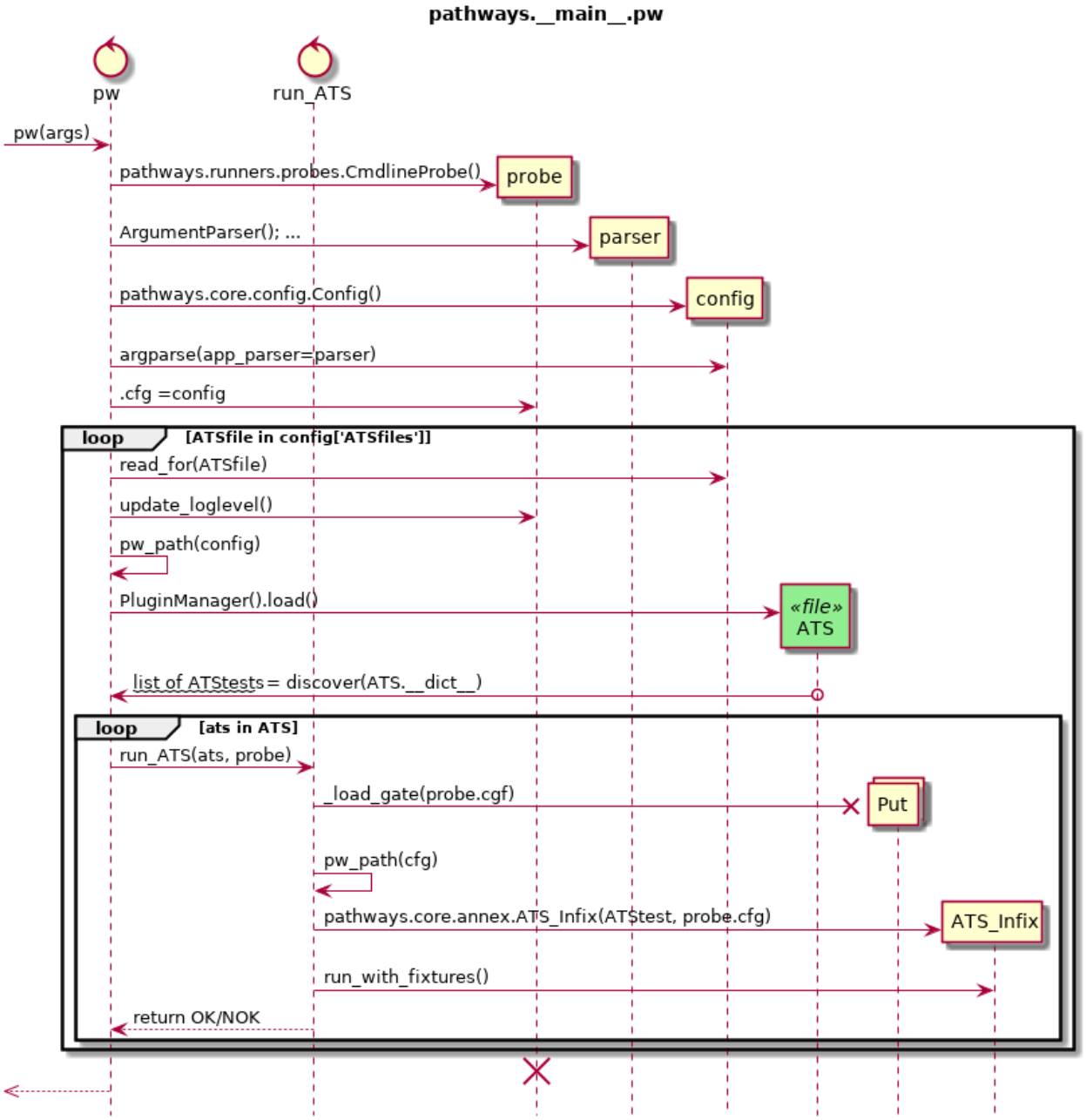
Design of the Pathways framework

Running some ATSes

These diagram show the working of code dated *Nov 2018*. They are made to find out how to gate/puts can use the `Config` (also see: *Put/gate & pathways.core.config.Config*). And to document the working.

Running `pw`

One can run one-or-more ATSfiles with one-or-more ATStests with `pw`. This ultimately run `run_with_fixtures()`; which is shown *below*.



See also:

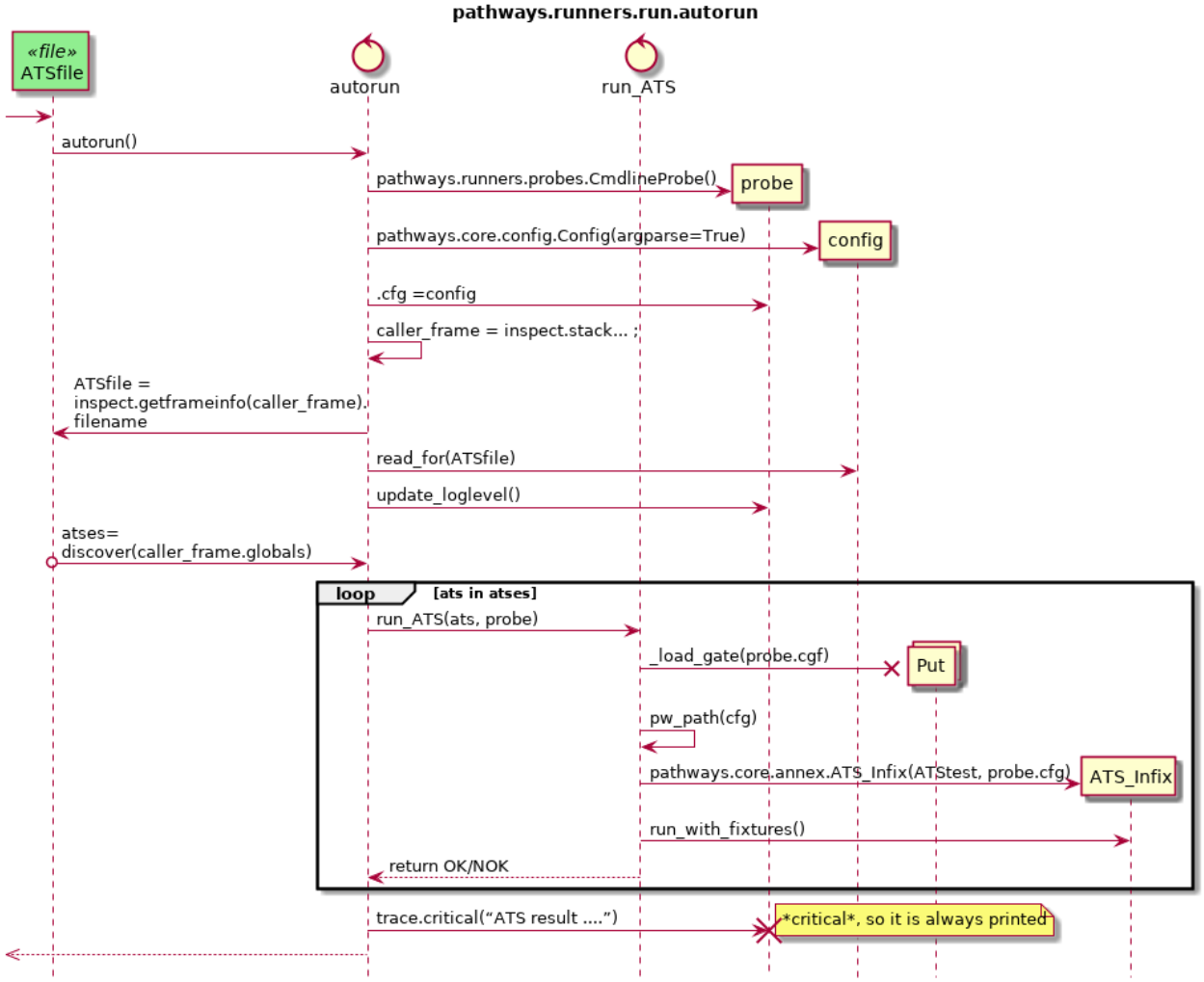
Source

- `pathways.__main__.pw()`
- `pathways.runners.run.run_ATS()`
- `pathways.runners.run.discover()`

Autorun

Alternatively, most ATS(files) can also be *autorun* –using trailer as shown. Although the sequence is a bit different, it will result in *running* `run_with_fixtures()`.

```
if __name__ == "__main__":
    exit(pathways.autorun())
```



See also:

Source

- `pathways.runners.run.autorun()`
- `pathways.runners.run.run_ATS()`

Running with Fixtures

When running an ATS(test), it `@Setup`, `@Check` and `@Teardown` fixtures are run before and after the ATS. And each Fixture can have this set of fixtures itself. The generalised in `annex`, and `fixtures`.

Notes

- The call to `pathways.runners.run._load_gate()` returns a Gate (`pathways.puts.put.Put` instance), with is not used yet!
- Probably, the `Put` class should be renamed to `Gate`.
- `pathways.core.annex.GenericFixture.run_with_fixtures()` and `pathways.core.annex.ATS_Infix.run_with_fixtures()` are similar, but not the same. The later is a wrapper around the ATS and will run `self._ats`. The generic supports multi-phases (*setup*, *check*, *teardown*) and calls `run_phase()`

3.1.4 The packages

CHAPTER

FOUR

BUREAULADE (DESK-DRAWER: ALL KIND OF STUFF)

The documents below are mostly in Dutch

4.1 Development (the project)

4.1.1 FEATURES (accepted)

Note: doc-style

The are written down as documented as if the features already is implemented. They are not, as long they are listed here.

Init/Config file

Hint: mostly done

During startup an init/config-file is read; when available. That file can contain global setting, that acts defaults for the parameters that are normally set as parameter. Like: `--gate`, `--put`, `--log` and `--vector`. (But without the dashed).

Those values override the defaults in the ATS, but are overriden by the command-line options.

There are several options

Syntax:

1. Use the ini-file syntax (a text file); the (default) extension is **.cfg**
2. Use a *sphinx-style conf.py* file (python code – *this allows ifs and calculations*)
3. Allow both; where (IDEA) both are read, and the python setting override the ini-file one

Place/Location

I) The file can be specified as parameter `--conf path/to/file`.

Then the below ones will be skipped!

II) A file with the same (base)name as the ATS-test (typical the ATS-filename), along that ATS-test (so the same dir).

So a *ATS* in the `ATS/path/demo.py` can use the config of `ATS/path/demo.cfg`

III) The `config.cfg` file in same directory as the ATS-file. This file is only valid for *ATS*'es in the same dir; an *ATS* in a subdir will *NOT* read/use it!

IV) The global file `.pathways.cfg`, in the user's home directory (`$HOME`, or its windows equivalent). Alternatively, the directory to find that file can be specified in the environment: `PATHWAYS_CONF_DIR`.

When several places are used, the top one override (value by value) the lower one. When the parameter `--conf-only` (with a file; or by the file as set by `--conf`) is used, the other ones are ignored

---

Todo: Describe the config format: sections, keys and `${key} expansion`

The **[ATS]** section is used (and, the default section)

Note: No ATS/shared.cfg The idea to have a `shared.cfg` file in the `ATS/` directory is abandoned.

This ATS-dir (or "top dir") is only a naming convention; there is no way to calculate (find) it. It has to be specified; but then, it is useless, not better then using a `--conf`. option

Use '`$HOME/.pathways`' option instead

Auto run ATS

Todo: AutoRun ATS_* (Need details)

All ATS-test-functions should have a name that start with `ATS_`; (not *main* as it used to be).

Then, those tests can be runs automatically (in alphabetic order, by dir/subb/file/func-hierarchy)

Pre/Post

Todo: Specify Pre/Post (setup/teardown)

It should be possible to run some code before and after each ATS.

Example: the code in **`examples/calculators/server/tst/connect_test.py`** shouldn't be in a shell-script but (impliciet) in an ATS

split/make package workshop, etc

Currently, all of Pathways is one bug *development structure*. That is too complicated for a workshop. So it has to spilt, or better: 'distributed' differently.

Probably the best way is to use make/makefiles to do that; possible some file has to move around

A. Make an installable pathways package (more or less the pathways-dir)

B. Make a –or a few– small workshop bundle(s). Both in demo mode and/or practice mode. In 'demo-mode' has all (needed) files; in practice-mode, you have make ATS/brick yourself; or extend them

C. Documentation; mainly: not in package, nor in WS-bundle. Possible WS and full

GATE/PUT

GATE vs PUTS

The *put*, the product under test, is typically an *address* to reach a (one?) thing; the *gate* is *code* that abstracts the gate from generic Pathways-code.

So, it kind of wrong the baseclass for the "gate" is "`Put`"; as are all current subclasses. This should be renamed!

Config

Put/gate & `pathways.core.config.Config`

A gate (plugin) should be able to use the config values. This is a feature, but also bug (alike)

E.g the webPut should use a BIN_PATH value to locate the `geckodriver`

Cmdline gate

MultiHead Gate/Put

In some cases, it would be very nice to have a gate/put with 2 (or more heads). Eg. to test a (MQTT, DDS, ...) channel. Both for functionality and performance.

4.1.2 IMPLEMENTED FEATURES

Version info (done)

Add version nummer to pathways-framework

Also see BUG *Version calculation (solved)*

New in version 0.0.2.dev20151007: The constantes `pathways.version` (a string) and `pathways.version_info` (a 5-tuple: Major, Minor, Maintenance, PreReleaseTag, PreReleaseNo) contain version info.

That info is also used as version sting to build the dist and archive files. It can be retrieved by using **`pathways/__version__.py`** as script. Typical with the option `--version_str`; which print it as string (like *0.0.2.dev20151007*).

4.1.3 Ideas (new)

Filters

As the expect-part of a test-vector contains only the essential results of a test we need to *filter* the returned test-data

4.1.4 ToDo's

Todo: Move _bricks back to bricks in `Pathways/examples/calculators/webapp/tst/` And solve this issue

---

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/DevelopmentProject/BUGS.rst, line 67.)

Todo: Describe the config format: sections, keys and `${key} expansion`

The **[ATS]** section is used (and, the default section)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/DevelopmentProject/FEATURES.rst, line 51.)

Todo: AutoRun ATS_* (Need details)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/DevelopmentProject/FEATURES.rst, line 70.)

Todo: Specify Pre/Post (setup/teardown)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/DevelopmentProject/FEATURES.rst, line 81.)

Todo:

label *_pathways-file-structuur* plus inhoud maken

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/FAQS/Workshop.nl/aanDeSlag.rst, line 15.)

Todo: Fix this

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/UserDoc/ManualPages/cfg.rst, line 39.)

Todo: Redesign the old 3-part gate and document it.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/UserDoc/ManualPages/cfg.rst, line 59.)

Todo: extent the list

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/UserDoc/ManualPages/cfg.rst, line 65.)

Todo: make browsers & driver selectable

Warning: Currently, the config files (and even some code!) contain hardcoded paths. That is a known error. . . .

- But as I am the main user, it will do for now. :-)
- A rewrite/update of the `core/conf` is needed.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/UserDoc/install_examples.rst, line 103.)

Todo: Make the examples (zip-file) downloadable, and document it.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/UserDoc/install_examples.rst, line 113.)

Todo: make pathways downloadable

For now: Download the source from Bitbucket!

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/UserDoc/installing.rst, line 75.)

Todo: This runner need work; the link above is a HACK

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/UserDoc/run_ATSes.rst, line 62.)

Todo: TstVector

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/glossary.rst, line 109.)

Todo: fixture

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/glossary.rst, line 113.)

Todo: DocMe (Lean) XXX

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pathways-rd/checkouts/latest/docs/doc/more_terms.rst, line 43.)

Note: Thise items are automatically collected from the pathways source.

Enhance the docstrings (or rst-files?) with one of:

.. todo::
    #. Some work ...
    #. More work that need done

or

.. todo:: DocMe (<where/what>)

4.1.5 BUGS that need fixing

clear_first

In .../calculators/webapp/tst/plugin/cobblestones/webMath.py **clear_first** is True (always using the default). That is wrong, and not allowing 1+2+3

Better to do that is the AST, or brick

(Only) Removing it will break the example-test

4.1.6 BUGS, which are mostly fixed

RekenServer: ImportError: No module named 'plugin'

No of the ATSes will run, as they can't import *plugin.cobblestones*

This depends on the setting of *PYTHONPATH*. When it contains a empty path (a colon/semicolon on it own), the cwd is on the PYTHONPATH. As that is "tst" dir, the plugin dir is found. Without that (semi)colon, the dir is NOT on the path, so not found, so an error is encountered.

Note: In both cases (tst)ATS is on sys.path (no 0). But that is the wrong dir

Error

```
>>> ATS/logInUit.py --vector ../../tst/tst-data/sessie.csv --log=WARNING
|Traceback (most recent call last):
|  File "ATS/logInUit.py", line 7, in <module>
|    from plugin.cobblestones import sessie
|ImportError: No module named 'plugin'
```

Workaround

For now, `pathways/__init__.py` add the current/tst dir to the path.

Listing 1: pathways/__init__.py

```
# Copyright (C) ALbert Mietus, SoftwareBeterMaken.nl; 2014, 2015. Part of Pathways␣
↪project -*- coding: utf-8 -*-
## Runners:
from  .runners.run import autorun


## Vectors:
from  .data.tstVectors import TstVectors

## verify
from .core.verify import verify

###
### HACKS
###

# THIS IS A WORKAROUND, to find the 'plugin' directories (and other local modules␣
↪also)
import sys
sys.path.append("") # Add the current dir to the search-path
```

Paths

Due to *RestructureExamples* several directories can have the same name, like:

1. examples/calculators/**webapp**/tst/*bricks*, and
2. examples/calculators/tst/*bricks*.

When both are on the loadpath (*PYTHONPATH*) only the first is searched for. A module as **bricks.add** isn't found when it located in the second "bricks"; even when the first doesn't have an *add* module.

Workaround

For now `Pathways/examples/calculators/webapp/tst/bricks` is moved to `Pathways/examples/calculators/webapp/tst/_bricks`, as it doesn't have important code

---

Todo: Move _bricks back to bricks in `Pathways/examples/calculators/webapp/tst/` And solve this issue

---

4.1.7 Known BUGS

pathways.puts.webput

The selenium driver doesn't work with **pypy3**. Firefox is started but remains blank. After some time Firefox disappears ...

```
(PyPy3)$ cd /Users/albert/work/Pathways/examples/calculators/src/webapp/tst
(PyPy3)$ PYTHON=pypy3 make test # or test1
@@ FAILED to run the test, due Message: 'Can\'t load the profile. Profile Dir: /var/
↪folders/......
```

examples/calculators/webapp/tst/plugin/cobblestones/webMath.py

Function `Webmath.__calc()` has a `clear_first` parameter, which is always True. This is considered as a **WRONG HACK**.

This results in clicking the 'clear' button; before each calculation. Which results in correct additions. But that is not how normal people act. Without this (or by setting it for False) the calculation are wrong (as the old value is the start of the next input)

```
>>> 5 + 7 ==> 12
>>> 1 + 2  ==> 123 # WRONG
The '1' is appended to the string '12', becoming 121. Adding 2 (``+ 2``) results in␣
↪123
```

4.1.8 Temporally code

dev: Generic ATS for all calculators

I'm busy with generic ATS scripts for calculator/webapp and calculator/server. It's kind of hard. And there are all kind of environmental relations. Sometimes selenium wil not start/run/load, (see below). And hardcoded paths, techno dependencies, ect.

So a lot of (temporally) changes and files are added

EXTRA files

A few extra (failing) test:

```
| examples/calculators/tryout/server.combi_g.sh
| examples/calculators/tryout/webapp.telop_g.sh
| examples/calculators/tryout/_shared.sh  *support*
```

These following '**_g**' files are modifications of the normal files:

```
| examples/calculators/src/webapp/tst/ATS/telop_g.py
| examples/calculators/src/server/tst/ATS/combi_g.py
```

Running the extra tests

The output is shortened

```
  (Python3)$ (cd examples/calculators/src/server; ../rekenServer.py &) # run the␣
↪server
  (Python3)$ cd examples/calculators/tryout

  # DEBUG=yes in env (pre prepend) will add debug-logging

  (Python3)$./webapp.telop_g.sh
  ## 2015-01-10 00:08:17,323 ERROR                 xmlrpcPut::xml_call
↪# The rekenserver does not support widget; details: <Fault 1: '<class \'Exception\'>
↪:method "widget" is not supported'>
  @@ FAILED to run the test, due a TstException: ('The rekenserver does not support
↪%s; details: %s', 'widget', <Fault 1: '<class \'Exception\'>:method "widget" is not␣
↪supported'>)
  [ ...  the webapp is OK ... ]
  === All test are run OK

  (Python3)$ ./server.combi_g.sh
  Testing SRV: --gate=/Users/albert/work/Pathways/examples/calculators/src/server/
↪tst:gate.rekenMachine:RekenServer --put=http://localhost:37421
  === All test are run OK
  Testing APP: --gate=/Users/albert/work/Pathways/examples/calculators/src/webapp/
↪tst:gate.rekenApp:RekenApp --put=file:///Users/albert/work/Pathways/examples/
↪calculators/src/webapp/js_calc.html
 ## 2015-01-10 00:05:19,159 WARNING                  sessie::_loginout_check
↪# LOGIN/OUT FAILED (on message): expect=('True', 'Welcome *'), got=(True, '0');␣
↪found=
 @@ One or more tests FAILED
```

4.1.9 FIXED BUGS

Version calculation (solved)

With the current setup, the version-info in calculated. *Both* during creation of the package, as well when installing. The latter is *WRONG*. Aside of the FileNotFoundError; the 'devYYYYMMDD' is not correct (build- vs install-time)

Error:

```
>>> C:\Pathways\pathways_framework>pip install -U C:\Pathways\pathways_framework\.
|pip install -U C:\Pathways\pathways_framework\.
|Processing c:\pathways\pathways_framework
|   Complete output from command python setup.py egg_info:
|   Traceback (most recent call last):
|     File "<string>", line 20, in <module>
|     File "C:\Users\postemer\AppData\Local\Temp\pip-_3eaouq2-build\setup.py", line␣
↪15, in <module>
|       (Major, Minor, Maintenance) = (int(n) for n in open("__version__").
↪readlines()[0].split('.')[:3])
| (etc)
```

Solution

Pre-calculate the (all) version-info, add the file in the package; when relevant. See std-docs. And extent to use that info also for the filename if the Pathways-all file.

Secondly, add version-info into pathways (also see Feature: *Version info (done)*)

4.1.10 Design Notes

RestructureExamples

Goals

- Remove the *src* dir between `calculators` and `server` cq `webapp` - the `src` and `tst` should be at same level
- Create a <*plugin*> dir; for gate, cobblestones etc

directory-overview

examples/ **OLD**	examples/ **NEW**
examples/calculators	examples/calculators
examples/calculators/doc.nl	examples/calculators/doc.nl
examples/calculators/**src**	
	examples/calculators/tst
	examples/calculators/tst/**<plugins>**
	examples/calculators/tst/**ATS**
examples/calculators/bricks	examples/calculators/**tst**/bricks
examples/calculators/tst-data	examples/calculators/tst/tst-data
examples/calculators/*src*/server	examples/calculators/**server**/src
examples/calculators/*src*/server/demo	examples/calculators/server/src/demo
examples/calculators/*src*/server/tst	examples/calculators/server/tst
	examples/calculators/server/tst/**<plugin>**
examples/calculators/*src*/server/tst/gate	examples/calculators/server/tst/**<plugin>**/gate – *file or dir*
examples/calculators/*src*/server/tst/ATS	examples/calculators/server/tst/ATS
examples/calculators/*src*/server/tst/bricks	examples/calculators/server/tst/bricks
examples/calculators/*src*/webapp	examples/calculators/**webapp**/src
examples/calculators/*src*/webapp/demo	examples/calculators/webapp/src/demo
examples/calculators/*src*/webapp/tst	examples/calculators/webapp/tst
	examples/calculators/webapp/tst/**<plugin>**
examples/calculators/*src*/webapp/tst/cobblestones	examples/calculators/webapp/tst/**<plugin>**/cobblestones – *file or dir*
examples/calculators/*src*/webapp/tst/gate	examples/calculators/webapp/tst/**<plugin>**/gate – *file or dir*
examples/calculators/*src*/webapp/tst/ATS	examples/calculators/webapp/tst/ATS
examples/calculators/*src*/webapp/tst/bricks	examples/calculators/webapp/tst/bricks
examples/calculators/tryout	examples/calculators/tryout
examples/doc	examples/doc

4.2 Bureau Lade

4.2.1 Pathways: "*Automatic product validation during lean & agile development*"

Attention: Onderstaande stond in de Sogeti-HighTech Nieuwsbrief; (31 December 2015)

Een poos geleden werd ik door een collega uitgedaagd: "Jij, als Pythondocent, kan toch wel je Automatisch TestScript concept uitwerken tot een algemeen beschikbaar voorbeeld?". Ze had natuurlijk een goed argument. Want al is mijn idee voor testautomatisering in agile projecten al een paar keer gerealiseerd, die waren allemaal klantspecifiek en (dus) niet beschikbaar voor een volgend team. Een algemeen beschikbaar voorbeeld zou dat veel makkelijker maken. "Nu kost het telkens veel moeite om die mensen te laten ervaren hoe goed het werkt", zei ze nog.

Dit heeft geleid tot Pathways: een referentie implementatie, als voorbeeld, om met behulp van automatische testscripts (ATSen) het ontwikkelen van softwaresystemen goedkoper te maken. Of beter, daar ben ik mee bezig. Gelukkig hebben een paar *beschikbare* collega's al een paar keer geholpen. Door mijn knutsels te proberen (*natuurlijk* werkte het niet bij hun ...), te brainstormen over hoe het eenvoudiger kan en door een soort van pilot workshop te volgen. Want ook dat is een doel: er moet een hiervoor een training komen.

Die hulp heeft, zoals verwacht, tot de nodige verbeteringen geleid. Maar ook tot meer werk, want het kan altijd beter en mooier. Ik kan dus nog steeds hulp gebruiken. Gelukkig vindt iedereen die geholpen heeft het Pathways-concept goed; dat geeft energie om door te gaan met het onderzoek.

Eén van de belangrijkste uitgangspunten voor Pathways is dat het **eenvoudig en goedkoop** moet zijn om testen te ontwerpen. Elke tester moet die mini python-scripts kunnen lezen, begrijpen en schrijven. Een triviale test (toevoegen) mag immers maar een paar minuten duren; de noodzaak om eindeloze documenten te schrijven moet (dus) voorkomen worden. Neem een rekenmachine als voorbeeld: hoe moeilijk is het om het optellen van twee getallen te testen? In het voorbeeld hieronder kan het in 5 regels, plus een handvol testvectoren. Zelfs zonder uitleg zal vrijwel iedereen het begrijpen. Het toevoegen van een extra test(case) is triviaal: gewoon één regel toevoegen aan een csv-file.

En de meeste testcollega's konden met onderstaande voorbeeld direct aan de slag om ook testen voor aftellen, delen en vermenigvuldigen te programmeren.

Feed		Expect
a	b	a+b
1	2	3
3	4	7
-12	42	30

```
def ATS_add(rekenapp, tstvectors):
    "A basic test to add 2 numbers and verify the result"
    for feed, expect in tstvectors:
        actual = rekenapp.add(feed[0], feed[1])
        assert actual == expect[0], \
          "adding %s => got %s, expected: %s" % (feed, actual, expect)
```

Natuurlijk is er soms meer werk nodig. Zonder in details te treden: `rekenapp` en `tstvectors` in dit voorbeeld zijn **fixtures**. Ook dat is telkens een handvol regels code, die zorgen voor de (herbruikbare) setup- en teardown van de test. Daarnaast gebruikt elke ATS één of meerdere **test-bricks**; die vaak geschreven worden met hulp van de programmeurs in het team. In dit voorbeeld *vertalen* die het (logische) `add`-commando naar (fysieke) gebruikershandelingen: bijvoorbeeld muiskliks op de toetsen van de webrekenmachine. Maar ook het teruglezen van het resultaat op het display.

Ook zijn bricks herbruikbaar. Het is zelfs mogelijk om complete testen te hergebruiken! In mijn voorbeelden kan ik ook een xmlrpc-rekenserver testen, gewoon door een andere set 'bricks' te gebruiken. Als het goed is geeft die hetzelfde resultaat; alleen de aansturing is anders.

Als iemand een robotje voor mij kan bouwen om de toetsen van een echt rekenmachientje te bedienen, kunnen we ook die testen opnemen als standaard voorbeeld. Wie helpt?

Ook als je wilt helpen als tester, programmeurs, elektronica-specialist, documentatie-schrijver of gewoon als hobbyist, neem gerust contact op! Ik heb nog ideeën zat! Ook zal ik proberen regelmatig een update te geven in deze nieuwsbrief.

— ALbert

4.2.2 NOTES

To build docs (on windows)

date September 29, 2016

Run:

```
> cd .../pathways-r-d/docs

> sphinx-build.exe -q -c . -b html   doc/ __result/html
Using std_conf
...\pathways-r-d\docs\doc\PathwaysFramework\index.rst:44: WARNING: toctree glob␣
↪pattern 'apidoc/*' didn't match any documents
...\pathways-r-d\docs\doc\PathwaysFramework\index.rst:34: WARNING: image file not␣
↪readable: PathwaysFramework\./pyreverse/classes_Pathways_Framework.svg
...\pathways-r-d\docs\doc\PathwaysFramework\index.rst:40: WARNING: image file not␣
↪readable: PathwaysFramework\./pyreverse/packages_Pathways_Framework.svg
...\pathways-r-d\docs\doc\index.rst:84: WARNING: toctree contains reference to␣
↪nonexisting document 'symlinks/examples.doc/index'
...\pathways-r-d\docs\doc\index.rst:93: WARNING: toctree contains reference to␣
↪nonexisting document 'symlinks/calculators.doc.nl/index'

# OPTIONAL: (Not needed, although some cross-links may be invalided)
> sphinx-build.exe -q -c . -b slides doc/ __result/slides
(Similar messages as above)
```

Errors, to solve

- BUG: Add the "external" std stuff to repro!
  - now, it a symlinks
- install sphinx extension (for docs)

  > pip install -r requirements.txt
- Symlinks

  Images (deleted for now)
  - docs/doc/_static/Needle_Tower_doorzichtig.png
  - docs/doc/_static/SwBMnl.png

  ??
  - docs/doc/_static/exports

  For the sidebar; delete seems to solve it :-)
  - docs/_templates/globaltoc.html
  - docs/_templates/relations.html
  - docs/_templates/searchbox.html
  - docs/_templates/sourcelink.html

VerifySelenium_Firefox.py (on windows)

date October 7, 2017

- geckodriver-v0.9.0 works
- geckodriver-v0.10.0 **DOES NOT**. However, the TIP/message is not correct:

```
$ python3 ./pathways/bin/VerifySelenium_Firefox.py  TMP/geckodriver-v0.10.0.exe
Starting browser ...
Selenium reported an error: Message: Service TMP/geckodriver-v0.10.0.exe␣
↪unexpectedly exited. Status code was: 1
. Possible, your url is incorrect?
However, selenium might be working. Fix your URL to be sure
```

4.3 INKOMEND

Some docs are (partly) written by others and emailed to me. The are temporally stored here. And will be updated and moved eventually

4.3.1 Some calculators, with ATSes

There a two calculator examples

1. A *WebApp*: A html-page, to run in your browser
2. A *xmlrcp* server: a deamon to offload your client

Both have (almost) the same, limited functionality

- Add, subtract, multiply, divide; mainly (positive) integers only
- Both requerer to 'login' first!

Although these 'products' are almost trivial, and have a useless 'login' feature; they are great examples to show the Pathways concept.

XXXXXX

sectionauthor Erik Jan, Albert Mietus

4.3.2 Installation

1. install firefox
2. install python (latest version)
3. install PIP (usually installed with Python, but not always)
4. add python to your path (http://stackoverflow.com/questions/3701646/how-to-add-to-the-pythonpath-in-windows-7)
5. add PIP to path (same as previous step, but now the folder C:/Python3x/Scripts
6. Install selenium (open command window and tupe: 'pip install selenium'

7. Unpack the Pathways-.zip file

webapp1

Edit the ini file `.../calculators/webapp/tst/ATS/config.cfg`

server1

Edit the ini file `.../calculators/server/tst/ATS/config.cfg`

```
### (C)  Part of Pathways project

[DEFAULT]

# My file/dir base
#######EDIT ME########
TOPd        :        C:\Pathways\pathways_all\
#######STOP HERE######
CALCd        : ${TOPd}examples/calculators/

# some locations, extending/using the shared.cfg setting

PLUGINd              : ./
SHARED-DATAd  : ${CALCd}/tst/tst-data/

# Log a lot, but not to much
log = INFO

[ATS]
gate    = ${PLUGINd}:plugin.gate.rekenServer:RekenServer
put        = http://localhost:37421
```

4.3.3 First Run

webapp2

1. open command window
2. navigate to yourpathwaypath/examples/calculators/webapp/tst
3. type **`python ATS/telop.py --vector ../../tst/tst-data/telop.csv`** The test should run and a firefox window opens where you see the calculator. In the end you should see '==ALL test are run OK' in de commandline

server2

Here the a deamon/service kind of product is tested; which is support to run *forever*. So to test it, we have to start it manually:

a. Open a (new) command window any shell on unix, 'cmd' on windows

b. Navigate to `.../calculators/server/src/`

c. Start the server: **`python ./rekenServer.py`**

Now, we can execute the tests, as normally. This is done is a separate window/shell/cmd-box

- Open a (new) command window any shell on unix, 'cmd' on windows
- Navigate to `.../calculators/server/tst/`

Several test are available:

- **`python ATS/logInUit.py --vector ../../tst/tst-data/sessie.csv`**
- **`python ATS/logInUit.py --vector ../../tst/tst-data/login.csv`**
- **`python ATS/optellen.py --vector ../../tst/tst-data/telop.csv`**

All these test should pass and should report **–ALL test are run OK**

Tip: When using a proper shell, the 'python' part is not needed. Using **`ATS/<test>.py`** has the same effect, with less typing.

Note: For those that didn't add python to the `PATH`; the full path to the python has to be specified. It has to be **python-3!** (3.4 or later)

Hint: It is possible to check whether the server runs, is reachable, by running command:*python connect_test.py*. When it give errors, the ATSes will fail to.

4.3.4 The ATS File

In this part the ATS file will be explained. The ATS consists of a few parts, of which the ATS itself is the most importent, since that is the actual test. The other parts are an setup and teardown function. An ATS file can consist of one or more ATS's, setup and teardown functions. It is also possible to have a general setup and teardown function as well. Therefore, a test set can have a structure like below. Note that the ATS specific setup and teardown functions are optional.

1) General setup function
    1) Setup 1
    2) ATS 1
    3) Teardown 1

    ...

    4) Setup n
    5) ATS n

6) Teardown n

2) General teardown function

ATS Function

In the ATS used as an example a caculator is tested, which has different functionalities, such as a login and printscreen function and off course the general functions a calculator should have. The calculator that is tested is a simple calculator. Therefore, it can only substract, add, multiply and divide whole numbers (integers). The ATS, of which parts are shown, has as goal to test the calculation functions only is shown below.

```
    logging.info("einde test")


def ATS_1(calc, vector, teardown):
    logging.info("start test 1")
    lastVal = 0
    for feed, expect in vector:
        som, num  = feed(0), feed(1)
        if som == "+":
            actual = calc.add(num)
        elif som == "-":
            actual = calc.sub(num)
        elif som == "/":
            actual = calc.div(num)
        elif som =="*":
            actual = calc.mul(num)
        else:
            print("foutje in het *.csv bestand")
            logging.error
        if actual != expect:
            logging.error #deze functie zou ook de som moeten printen die is␣
↪uitgevoerd dus: (lastVal, som, num, '=', actual, '!=', expect
        else:
            logging.succes #geen idee of dit bestaat. Anders gewoon een print('test␣
↪is OK!') of iets dergelijks. Zou wel mooi zijn als deze bestaat
```

In this function the external test data is used to perform calculations with the calculator. The result of this calculation is compared to the expected results in the test data.

Vector function

From the test data file, which is a comma seperated file, the test data is imported. The test data has the following structure `+,2,,3`.

4.3.5 The ATS

In this part the Automatic Test Scripts (ATS) will be explained. It is assumed a general understanding about the pathways concept is acquired. Furthermore, a global understanding of testing in general is needed. A tester will typically work with ATS files.

Generally an ATS-file consists of a few parts, of which the ATS itself, a setup function and a teardown function. The constists ATS is the actual test. The setup function is an initialization of the test and the teardown function will shutdown the test. An ATS-file can consist of one or more ATS's, setup and teardown functions. It is also possible to have a general setup and teardown function. Therefore, a test-set can have a structure like below. Note that the ATS specific setup and teardown functions are optional.

- General setup function
  - Setup 1
  - ATS 1
  - Teardown 1

    ...
  - Setup n
  - ATS n
  - Teardown n
- General teardown function

In the ATS-file used as an example a caculator is tested, which has different functionalities, such as a login and printscreen function and the general functions a calculator should have. The calculator that is tested is a simple calculator. Therefore, it can only substract, add, multiply and divide integers. The ATS-file used as an example has as a goal to test the calculation functions only.

The ATS Function

```
    logging.info("einde test")


def ATS_1(calc, vector, teardown):
    logging.info("start test 1")
    lastVal = 0
    for feed, expect in vector:
        som, num  = feed(0), feed(1)
        if som == "+":
            actual = calc.add(num)
        elif som == "-":
            actual = calc.sub(num)
        elif som == "/":
            actual = calc.div(num)
        elif som =="*":
            actual = calc.mul(num)
```

(continues on next page)

(continued from previous page)

```
        else:
            print("foutje in het *.csv bestand")
            logging.error
        if actual != expect:
            logging.error #deze functie zou ook de som moeten printen die is␣
↪uitgevoerd dus: (lastVal, som, num, '=', actual, '!=', expect
        else:
            logging.succes #geen idee of dit bestaat. Anders gewoon een print('test␣
↪is OK!') of iets dergelijks. Zou wel mooi zijn als deze bestaat
```

In this function the external test data is used to perform calculations with the calculator. The result of this calculation is compared to the expected results in the test data. The function that has been defined needs three inputs: 'calc', 'vector' and 'teardown'. These refer to the setup and teardown functions, with the help of fixtures.

Fixtures

Fixtures are a specific method where one can point to specific funtions. For example in the ATS function shown below, three parameters are included; calc, vector and teardown.

```
    logging.info("einde test")
```

The parameter 'calc' points to the function calc shown below. Using fixtures you are able to point to this whole function, instead of just one parameter. Therefore, calling upon the ATS_1 will first execute the 'calc' function and will return the calc value as a parameter to the ATS_1 function. The same principle holds for the teardown and vector function as well.

```
import logging

#onderstaande functie is een initializatie voor de test
@fixture.setup
def calc(inifile):
    gate = from_inifile(gate)       #weet dat dit niet klopt, maar weet niet hoe
    put = from_inifile(put)
    calc = tstObj(gate, put)        #kiest de rekenmachine
    calc.login                      #logt in op de rekenmachine
```

Vector function

From the test data file, which is a comma seperated file, the test data is imported. The test data has the following structure: **+,2,,3**. The data within this file is used to make calculations on the calculator which is tested. The first two characters are operations and the third character is the expected result. After a calculation the result of the calculator is compared to the expected result.

Caution: Ik zie geen reden meer om dit gescheiden te houden van de calc functie

Calc function

The Calc function is a fixture as well and it is an initilization for the test. In the ATS-file used as an example, the calculator that is tested is chosen. Furthermore, the Calculator will be made ready for the test itself. Note that for the test described in this document it makes sense to place the login function in the setup function.Would one like to test the login function explicity, it should be placed in the ATS.

```
import logging

#onderstaande functie is een initializatie voor de test
@fixture.setup
def calc(inifile):
    gate = from_inifile(gate)       #weet dat dit niet klopt, maar weet niet hoe
    put = from_inifile(put)
    calc = tstObj(gate, put)        #kiest de rekenmachine
    calc.login                      #logt in op de rekenmachine
```

4.3.6 Start-up

Steps to complete

1. install firefox
2. install python 3.x
3. install PIP (usually installed with python, but not always on windows machines)
4. add python to your path (http://stackoverflow.com/questions/3701646/how-to-add-to-the-pythonpath-in-windows-7)
5. add pip to path (same as previous, but add the folder C:/Python3x/Scripts
6. install Selenium, in command line 'pip install selenium'
7. unpack the Pathways folder
8. edit the ini file in yourpathwaypath/examples/calculators/webapp/tst/ATS/config.cfg

Error

```
>>>
@@ FAILED to run the test, due Message: Component returned failure code: 0x80520001␣
↪(NS_ERROR_FILE_UNRECOGNIZED_PATH) [nsIWebNavigation.loadURIWithOptions]
Stacktrace:
    at _loadURIWithFlags (chrome://browser/content/browser.js:12204)
    at loadURIWithFlags (chrome://browser/content/tabbrowser.xml:6155)
    at loadURI (chrome://global/content/bindings/browser.xml:119)
    at loadURI (chrome://browser/content/tabbrowser.xml:3622)
    at FirefoxDriver.prototype.get (\file:///C:/Users/postemer/AppData/Local/Temp/
↪tmpzhcta5xy/extensions/fxdriver@googlecode.com/components/driver-component.js:10515)
    at DelayedCommand.prototype.executeInternal_/h (\file:///C:/Users/postemer/
↪AppData/Local/Temp/tmpzhcta5xy/extensions/fxdriver@googlecode.com/components/
↪command-processor.js:12617)
    at DelayedCommand.prototype.executeInternal_ (\file:///C:/Users/postemer/AppData/
↪Local/Temp/tmpzhcta5xy/extensions/fxdriver@googlecode.com/components/command-
↪processor.js:12622)
    at DelayedCommand.prototype.execute/< (\file:///C:/Users/postemer/AppData/Local/
↪Temp/tmpzhcta5xy/extensions/fxdriver@googlecode.com/components/command-processor.
↪js:12564)
```

(continues on next page)

(continued from previous page)

Solution

Edit the ini file 'config.cfg' in the ../webapp/ATS/ directory - remove the line 'home . . . ' - set the path in after 'TOPd' to the path of pathways

INDEX

Symbols

A

B

C

E

F

G

I

L

P

S

T

V