

---

# **MESS**

***Release 20200627.14***

**unknown**

**Jun 27, 2020**



# CONTENTS

<b>1</b>	<b>Table of Content</b>	<b>3</b>
1.1	ReadMe . . . . .	3
1.2	System Engineering . . . . .	5
1.3	Software competency . . . . .	24
1.4	PyMESS: MESS with python . . . . .	46
1.5	BLOG indexes . . . . .	54
<b>2</b>	<b>Tensegrity, as inspiration</b>	<b>55</b>
	<b>Index</b>	<b>57</b>



Also knows as: “**Modern Embedded Software Systems**”

---

**Note:** See the [ReadMe](#) on what I mean with ‘*Modern Engineering*’ and ‘*Sovereign Software*’. And for the *Copyright*

---



## TABLE OF CONTENT

### Engels & Dutch

- This site is partly in Dutch.
- En gedeeltelijk in het Engels

## 1.1 ReadMe

### Engels & Dutch

- This site is partly in Dutch.
- En gedeeltelijk in het Engels

**Modern Engineering** Software-engineering is a juvenile profession, with many new insights.

Most are recently *discovered* in generic software development; like ‘the web’. And written in a language (both computer and human), that is very contrasting with the technology-talk of typical “RealTime/Embedded” engineering. And so, often rejected or not seen as relevant.

Although their examples are too dissimilar in many cases, the concepts can be useful. There is no valid reason not to incorporate their modern approach of software-engineering into *our* process.

**Sovereign Software** All **unmanaged**, *background* software enabling modern life!

- Traditional RealTime/Embedded software & their big successors
- Compilers, Kernels, Drivers, code-analysers, ...
- Routers, PLCs, EMUs, the TCP/IP-stack, ...

Traditionally, that “small” software was called “RealTime” and/or “Embedded”. Currently most of that software isn’t ‘*small*’ anymore, nor ‘*embedded (in a product)*’. Furthermore, all *normal* software appears to be realtime; mostly due hardware-speedup.

And, there is a lot of software, like (kernel)-drivers, compilers and SW-tools which isn’t embedded, nor realtime; but should have the quality that is equivalent to that old- school engineering. As everybody depends on it – directly (sw-engineering) or indirectly (a bug in a compiler will effect all end-users!).

### 1.1.1 Copyright

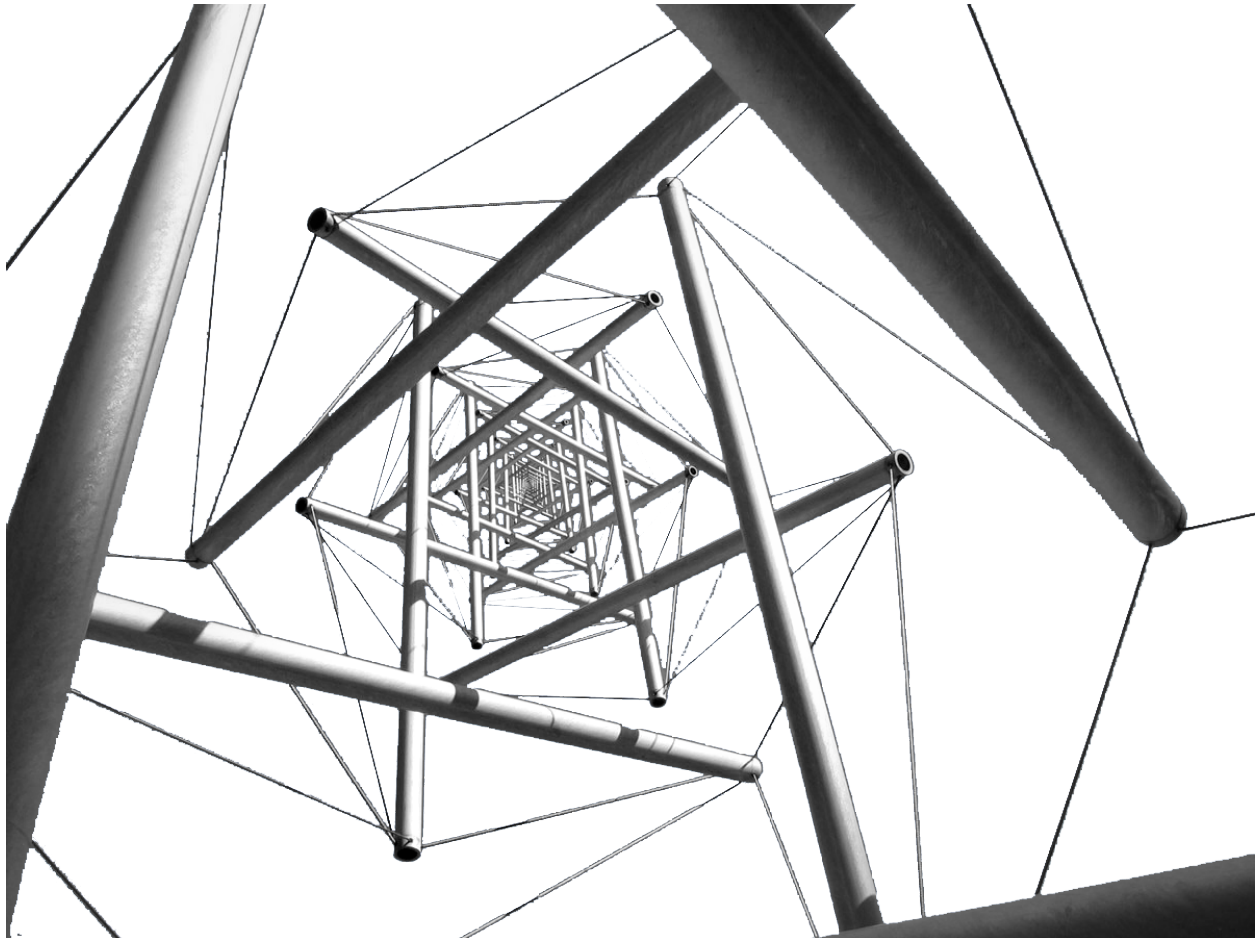
All files and ideas are (C) Albert Mietus. You may:

- Read & study them
- Use the ideas to improve your skills

Please use the *disqus* sections (below) to give your feedback and opinions.

–Albert Mietus

### Tensegrity, as inspiration



**Tensegrity** is a synthesis of the names ‘tensional’ and ‘integrity’. It is based on “*teamwork*” of tension and compression forces. Although the image may look confusing, these structures are very simple. All you need are some poles, some cable, and good engineering. This results in a beautiful ‘tensegrity-tower’ where the poles almost float in the air; as shown [above](#)

It is also a well-known architectural principle for skyscrapers!

For me, it is also an inspiration for Software-Engineering: It should be based on teamwork: a synthesis of creative and verifying people. Together with a methodical way-of-working the amplify each other. Then, the sky becomes a limit, which is easy!



## 1.2 System Engineering

### 1.2.1 Requirements Traceability

**study-time** 1 hour

Requirement-Management, and especially “*Tracking Requirements*” isn’t that complicated, but it needs some discipline. Often that discipline is *replaced* by a tool, which is encouraged by tool-vendors. That hardly ever works.

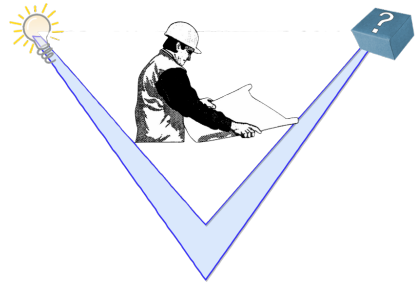
This blog introduces requirements traceability in a conceptual, but pragmatic way. It shows *–as a demo–* what a team has to do (defining requirements and describing the relations) and what the benefit is. It assumes a (more or less) agile way-of-work and shows how it can become lean. A small investment in good requirements definitions makes validations a lot easier.

And yes, it used a tool: a free one; as part of the documentation-system that is used for this side. An structured, version-controllable, textual way to write documentation, including all requirements. With a plugin that can visualize all specifications and their relations.

Those “*particulars*” will be shown too.

---

#### Summary (testers view)



- For a simple development cycle, it easy to trace the validation of the product: *All features should be covered by a test, and all should pass.*
  - When multiple products –or multiple releases– are mandatory, this becomes quickly ambiguous: *Some tests have to pass, and only the essential specifications need test-cases.*
  - Then ‘requirements traceability’ becomes crucial: **It gives insight into which ‘specs’ are required by each product/release and which tests are essential.**
  - This article (as part of a workshop) shows a down-to-earth demo of how you set-up the ‘requirements traceability’ and how to use it.
  - We use the ‘tester view’: Be able to demonstrate the *product does* what was *aimed for*!
-

## Content

### Goals

“Requirements traceability is a sub-discipline of requirements management within software development and systems engineering ...”

“... is defined as “the ability to describe and follow the life of a requirement in both a forwards and backwards direction”

—[https://en.wikipedia.org/wiki/Requirements\\_traceability](https://en.wikipedia.org/wiki/Requirements_traceability)

There are many reasons to enroll requirement-management and -traceability.

- First of all, it’s a good habit: without an objective it hard to succeed. *How can a system pass an acceptance-test when it is not clear what the (current) requirements are?*
- Often more urgent: in many domains, it is demanded by law or standardisation committees. In healthcare, automotive and aerospace “safety” is mandatory. Standard like IEC61508, and (A)SIL more or less demand a process with proper requirement traceability.

With the uprise of lean, agile and scrum, there is a tendency to skip less popular steps (and expensive roles) and to trust on craftsmanship. This implies the traditional ‘requirement management’ approach has to change. And also, that more people should be involved.

This chapter gives you an introduction into ‘tracking requirements’, in a lean, agile fashion.

After this *lecture*, you should understand how to annex (written) requirements and other specifications, to be able to link them with test-cases. You should be able to visualize those relations, to get insight into which tests are needed for a specific product-release. (And the other way around.) Besides, you should be able to apply this manually, for moderately simple products. For big, complicated products, you probably need a bit more experience and the assistance of a standardized process and mayhap a tool, to make it lean.

### The classical Requirements Traceability Matrix

Most literature about requirements-traceability focus on the Requirements Traceability Matrix<sup>1</sup>. This table gives the relation between the (high level) requirements versus all test. As ‘needs’ can’t generate this table (yet), I have created one as example.

With this overview one get insight about which test are needed to verify one requirement. Sometimes it explicitly counts the number of available tests pro requirement too. Albeit that is a nice result, the ATM isn’t that useful in an agile environment. When a requirement is added, ore one changes, the RTM itself gives little help to update the table.

---

<sup>1</sup> Often shorthanded to RTM, or simle “Traceability Matrix”, as wikipedia does.

Table 1: RTM for the *Exact Calculator (CALC2)*

Test\Req	Generic Add (CALC_ADD)	Generic Sub (CALC_SUB)	Generic Multiply (CALC_MULT)	Generic Divide (CALC_DIV)	Big fractional numbers (CALC2_1000ND)
Basic addition test (CALC_TEST_ADD_1)	X				X
Big addition test (CALC_TEST_ADD_2)	X				X
Subtract test (CALC_TEST_SUB_1)		X			X
Multiplication test (CALC_TEST_MULT_1)			X		X
DIV test (demo2 only) (CALC2_TEST_DIV_1)				X	X

## Footnotes & Links

**Wikipedia on RTM:** [https://en.wikipedia.org/wiki/Traceability\\_matrix](https://en.wikipedia.org/wiki/Traceability_matrix)

## Demo: Some calculators

This chapter contains a few demonstrations of the Requirements Traceability concept.

We start with a very simple product: the *Simple Calculator (CALC1)*; with only a few generic requirements and tests. By defining those “needs”<sup>1</sup> it becomes possible to show a (generated) *graph* with the relations between them. Probably, you will be able to implement and verify this simple product without (formal) requirements-traceability; but it gives a nice introduction to the concept.

Then, a new product the *Exact Calculator (CALC2)* is defined, in the same way.

Even it has only one (*puzzling*) additional requirement; that one makes both the implementation and the validation burdensome. And changes the test-cases completely; as shown in *its requirement-traceability graph*.

Likewise, we could add more demo’s with more and more specifications, tests and *need* at multiple-levels; but you will get the idea: By simple ‘defining’ the needs, it becomes possible to get insight; independent of the size of the products or the numbers of specifications.

Finally, an *overall graph* is shown, with all the products, all the tests, and all the connecting *needs*.

## Content

### [CALC1] A simple calculator

The specifications are quite trivial ...

<sup>1</sup> The universal term ‘*need*’ is used for all kinds of requirements, specifications, test-specs, etc. This name comes from the “needs” extension to “sphinx-docs” that we use in this demo.

## The product we need

Demonstrator: <b>Simple Calculator</b> <i>CALC1</i>
tags: demo1  links incoming: <i>CALC_ADD</i> , <i>CALC_SUB</i> , <i>CALC_MULT</i> , <i>CALC_DIV</i> , <i>CALC2</i>
For this demo, a simple calculator is used. It should work with integers; and has only a few requirements. See below.

We use this extremely simplistic example as you will agree on its requirements.

## Some general requirements, for all calculators

Requirement: <b>Generic Add</b> <i>CALC_ADD</i>
tags: general  links outgoing: <i>CALC1</i> , <i>CALC2</i> links incoming: <i>CALC_TEST_ADD_1</i> , <i>CALC_TEST_ADD_2</i> , <i>CALC2_1000ND</i>
All calculators should be able to sum two numbers and show the result.

Requirement: <b>Generic Sub</b> <i>CALC_SUB</i>
tags: general  links outgoing: <i>CALC1</i> , <i>CALC2</i> links incoming: <i>CALC_TEST_SUB_1</i> , <i>CALC2_1000ND</i>
All calculators should be able to subtract a number form another number.

Requirement: <b>Generic Multiply</b> <i>CALC_MULT</i>
tags: general  links outgoing: <i>CALC1</i> , <i>CALC2</i> links incoming: <i>CALC_TEST_MULT_1</i> , <i>CALC2_1000ND</i>
All calculators should be able to multiply two numbers.

Requirement: <b>Generic Divide</b> <i>CALC_DIV</i>
tags: general  links outgoing: <i>CALC1</i> , <i>CALC2</i> links incoming: <i>CALC2_1000ND</i> , <i>CALC2_TEST_DIV_1</i>
All calculators should be able to divide two numbers.

### Add this is how we test it

As we have defined only general requirements, we only need some generic tests.

Test_Case: <b>Basic addition test</b> <i>CALC_TEST_ADD_1</i>
tags: general  links outgoing: <i>CALC_ADD</i> , <i>CALC2_1000ND</i>
Sum two numbers and verify the result is correct. By example: Add 2 and 5 and check the result is 7

Test_Case: <b>Big addition test</b> <i>CALC_TEST_ADD_2</i>
tags: general
links outgoing: <i>CALC_ADD</i> , <i>CALC2_1000ND</i>
Add the numbers 2222 and 5555 and check the result is <b>7777</b>

Test_Case: <b>Subtract test</b> <i>CALC_TEST_SUB_1</i>
tags: general
links outgoing: <i>CALC_SUB</i> , <i>CALC2_1000ND</i>
Feed two numbers to the calculators, in the right order and verify the result. E.g: <ul style="list-style-type: none"> <li>• Subtract 5 from 7 and check the result is <b>2</b></li> <li>• Subtract 5555 from 7777 and check the result is <b>2222</b></li> </ul>
<b>Note:</b> Here we specify two test in one test-requirement; just to show another style

Test_Case: <b>Multiplication test</b> <i>CALC_TEST_MULT_1</i>
tags: general
links outgoing: <i>CALC_MULT</i> , <i>CALC2_1000ND</i>
You get the idea ...

## Experience practice

1. There are several kinds of ‘needs’.

Here we use the toplevel *Demonstrator* (as it is not a real product), *Requirement* and *Test\_Case*; later we will introduce *Specification* too. More kinds & levels can be configured.

2. Every ‘need’ should have an unique and stable ID; this label is used to link other ‘needs’.
3. Some ‘needs’ are linked to an “earlier/higher” ‘need’.

You can see such an outgoing-link in e.g the requirements (You might need to “open” the details-row)

4. Each outgoing-link will automatically result in an incoming-link on the references need. (Again, open the details-row, to be able to “follow” it in the ‘forward’ direction).

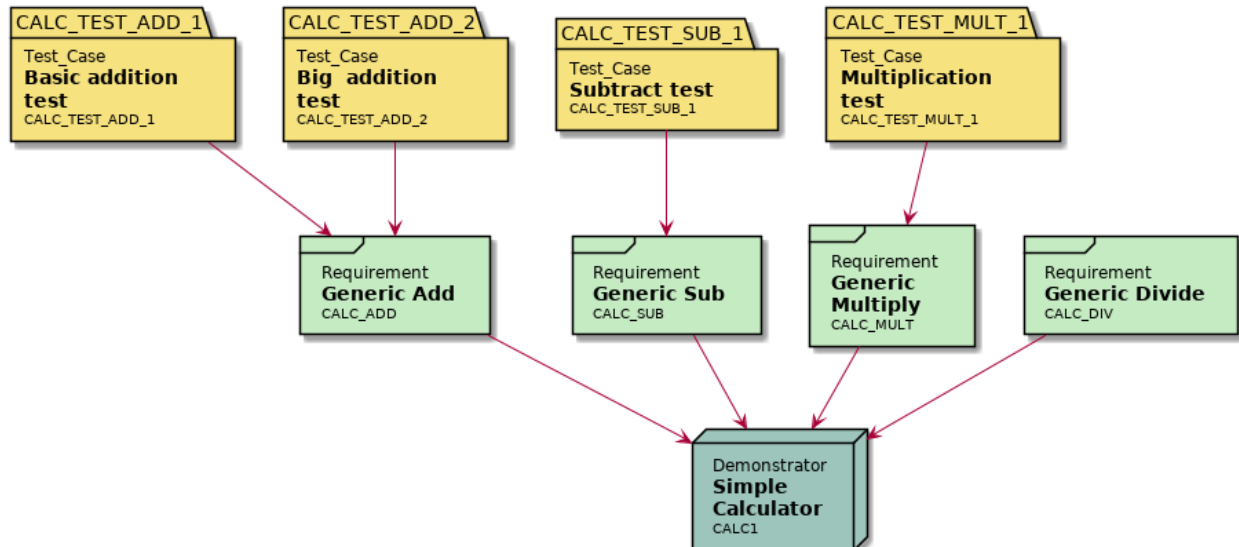
**Hint:** This article uses ‘sphinx-doc’ with the ‘needs’ plugin to define requirement. This is a text-based (and so version-controllable) tool; therefore it is painless to show the details of how it works; that is done in *Particulars for CALC1*

## Requirements Traceability [CALC1]

With the defined ‘needs’, their relations can be shown automatically, or some tables with relevant ones can be listed. Below you will find two examples, for *Simple Calculator (CALC1)*.

### A graphical view (“tree”)

This view shows the relationship between all kinds of specifications. It is fully automatically generated and will be updated when the specifications change.



## Lessons learned

1. This graph clearly shows there are **no tests** for [Generic Divide \(CALC\\_DIV\)](#). This is easily overseen as there are four requirements *and* four tests defined.
2. When the requirement [Generic Add \(CALC\\_ADD\)](#) changes, two tests might need an update.
3. Likewise, for the other requirements, it is directly visual where the relations are.

This very simple demo has only *one* product with *four* requirements and a few tests. There are no product-variant, no product-increments (“new releases”) and no intermediate (or hierarchical) specifications. As a (deliberate) result, its **Requirements Traceability** is simple.

Still, its easy to forget a test; as I did. Did you noticed it?

**Attention:** The “forgotten test” is intentional in this first demo. It will be hot-fixed in the *next chapter*, and you will see it in the other graphs.

Therefore I had to use some “trick” in needs; see *the notes about the forgotten test* for more info.

## The Requirements Matrix (table)

Some people prefer to see the same information in a table; again this one is automatically generated.

ID	Type	Title	Incoming	Outgoing
<a href="#">CALC1</a>	demo	Simple Cal- culator	<a href="#">CALC_ADD</a> ; <a href="#">CALC_SUB</a> ; <a href="#">CALC_MULT</a> ; <a href="#">CALC_DIV</a> ; <a href="#">CALC2</a>	
<a href="#">CALC_ADD</a>	req	Generic Add	<a href="#">CALC_TEST_ADD_1</a> ; <a href="#">CALC_TEST_ADD_2</a> ; <a href="#">CALC2_1000ND</a>	<a href="#">CALC1</a> ; <a href="#">CALC2</a>
<a href="#">CALC_SUB</a>	req	Generic Sub	<a href="#">CALC_TEST_SUB_1</a> ; <a href="#">CALC2_1000ND</a>	<a href="#">CALC1</a> ; <a href="#">CALC2</a>
<a href="#">CALC_MULT</a>	req	Generic Multiply	<a href="#">CALC_TEST_MULT_1</a> ; <a href="#">CALC2_1000ND</a>	<a href="#">CALC1</a> ; <a href="#">CALC2</a>
<a href="#">CALC_DIV</a>	req	Generic Divide	<a href="#">CALC2_1000ND</a> ; <a href="#">CALC2_TEST_DIV_1</a>	<a href="#">CALC1</a> ; <a href="#">CALC2</a>
<a href="#">CALC_TEST_ADD</a>	test	Basic addition test		<a href="#">CALC_ADD</a> ; <a href="#">CALC2_1000ND</a>
<a href="#">CALC_TEST_ADD</a>	test	Big addition test		<a href="#">CALC_ADD</a> ; <a href="#">CALC2_1000ND</a>
<a href="#">CALC_TEST_SUB</a>	test	Subtract test		<a href="#">CALC_SUB</a> ; <a href="#">CALC2_1000ND</a>
<a href="#">CALC_TEST_MULT</a>	test	Multiplication test		<a href="#">CALC_MULT</a> ; <a href="#">CALC2_1000ND</a>

**Hint:** For now, ignore the *links* to [CALC2](#) and [CALC2\\_1000ND](#). Those will be in documented in [\[CALC2\] The exact calculator](#)



## Lessons learned

1. This generated tabular overview kind of act as an index to all “clickable” *needs*. It’s a great page to bookmark.
2. One can even add a status-column (not shown here), or filter on (show only) e.g. test that fails.
3. It gives less insight; therefore it good to have both overviews.

Everybody understands that when the product-definition changes, the implementation will change too. And, that the test-set will change, partially; by example: new tests will be added.

However, some tests may even become obsolete!

So, just re-running all (existing) tests as a regression-test, may not work. The question is: which of all test are related to the changed requirement?

With a table as above, the answer is just one click away.

## Next steps

When we introduce variants, sprint-increments, multiple (sub)components, libraries, versions, releases, etc, the challenge becomes bigger. Especially when a project-teams grows, it might become a nightmare to know which test has to be re-run, when a specification changes.

Unless one uses a (simple) approach as shown above. Then, everybody can just see which *rework* is needed when something “upstream” changes. And, by adding a “status” to each spec, we can even make this visual.

See [\[CALC2\] The exact calculator](#) for a bit more complex example: Adding a product-variant and (only) one extra (non-functional) requirement.

## [CALC2] The exact calculator

This demo is just a bit more complicated then [\[CALC1\] A simple calculator](#): this product-variant has *one* extra requirement.

## A bit more complicated product

Demonstrator: **Exact Calculator** [CALC2](#)

tags: demo2

links outgoing: [CALC1](#)

links incoming: [CALC\\_ADD](#), [CALC\\_SUB](#), [CALC\\_MULT](#), [CALC\\_DIV](#), [CALC2\\_1000ND](#)

This calculator should work with [Fractional Numbers](#), and be exact for very big numbers; as defined in [Big fractional numbers \(CALC2\\_1000ND\)](#)

**Warning:** This implies `floats` are not possible in the implementation

## The extra requirement

Specification: <b>Big fractional numbers</b> <i>CALC2_1000ND</i>
tags: demo2
links outgoing: <i>CALC_ADD</i> , <i>CALC_SUB</i> , <i>CALC_MULT</i> , <i>CALC_DIV</i> , <i>CALC2</i> links incoming: <i>CALC_TEST_ADD_1</i> , <i>CALC_TEST_ADD_2</i> , <i>CALC_TEST_SUB_1</i> , <i>CALC_TEST_MULT_1</i> , <i>CALC2_TEST_DIV_1</i>
The <a href="#">Exact Calculator</a> ( <i>CALC2</i> ) should work with fractions; where nominator and denominator can be very long: up to <b>1000 digits</b> .

## Hotfix the missing test

We also *repair* the missing test in demo1, but only for demo2 (Because it is still a *demo!*).

Test_Case: <b>DIV test (demo2 only)</b> <i>CALC2_TEST_DIV_1</i>
tags: demo2
links outgoing: <i>CALC_DIV</i> , <i>CALC2_1000ND</i>
Subtract $1/3$ from $1/2$ and check the result is <b><math>1/6</math></b> .
<b>Note:</b> This test is was intentionally “forgotten” as explained in the <i>forgotten test</i> . Therefore it is only added for the <a href="#">Exact Calculator</a> ( <i>CALC2</i> ). See <i>the notes about the forgotten test</i> for more info.

## How to test?

The [Big fractional numbers](#) (*CALC2\_1000ND*) requirement is a good example of a “*nonfunctional*” (actually: a non-distributable) specification. It is valid for all other requirements; all parts of the implementation should adhere to it.

Testing this requirement is also different too. The same tests are valid: we have to add, subtract, multiply and divide.

Only, now we have to use other numbers; really big ones!

## Traditionally

In the traditional world, using the TMAP-terms, this approximately come down to:

- Reuse the *logic test*.
- Change a *physical test* (or add one).

## Modern

When using an agile test-automation framework this implies

- The ATS (Automated Test Script) isn't altered.
- Some "Test-Vectors" (or test-data) is added: the big-fractions.

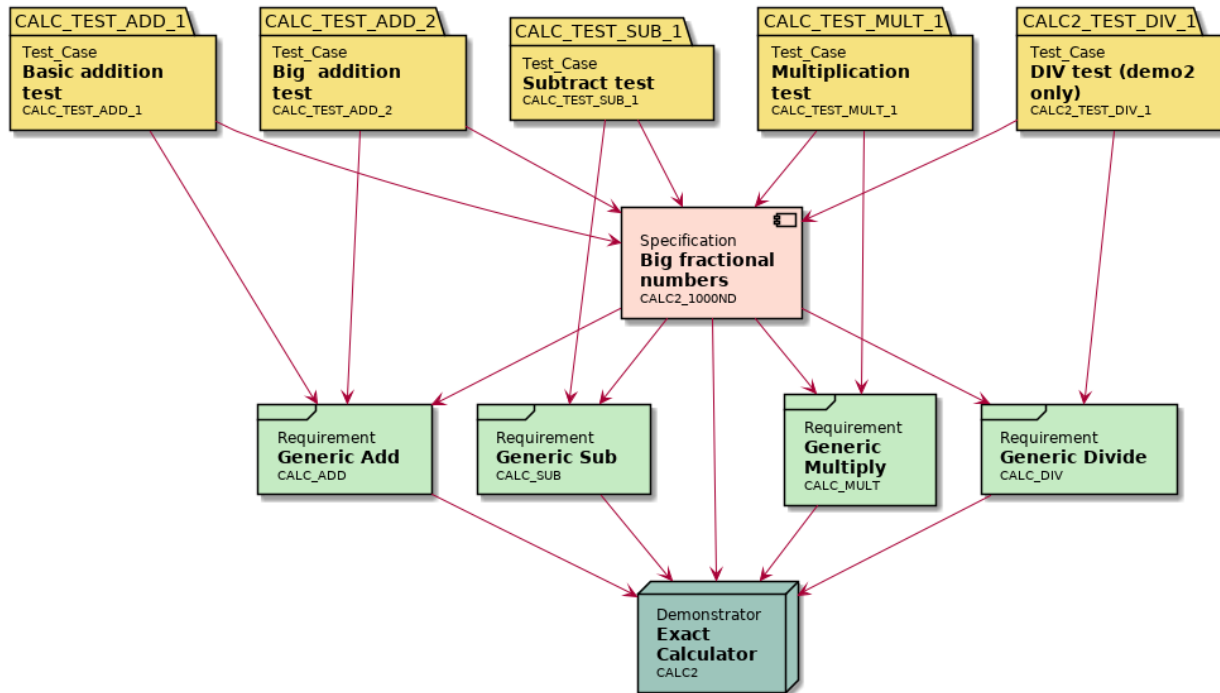
## Experience practice

1. It is possible to have multiply "toplevel" 'needs'. Here, that are *Demonstrators*, but it possible to use *Products*, *Variants*, and/or *Releases* etc, as well.
2. Here, a new kind of 'need' is introduced: *Specification*. As you will see on the next page, it influences not only the implementation, but also testing.
3. In the 'details-row', you can see it has (outgoing) links to many (all) earlier requirements.

## Requirements Traceability [CALC2]

As in "[*CALC1*] *A simple calculator*", we can automatically generate the an overview of the requirements.

## The tree view



## Lessons learned

1. We can directly see that specification **Big fractional numbers** (CALC2\_1000ND) influences all existing test-cases.
2. Each test *depends* on both a (functional) requirement and a (non-functional) specification (at least in this case).
3. The requirements-relations can become more complicated, even by adding **only one** requirement!

Can you imagine what will happen when we add a handful of requirements to the calculator (*memory, square-root, powers*)? Or, a few more non-functionals (*speed-of-operation, floats*). Then the complexity quickly raises even for such a simple product. And it becomes hard to predict which tests have to be adapted or rerun.

Likewise, when a few requirements become altered in an upcoming sprint: can you predict which tests will have to change? A graph, as above, will certainly help in working that out.

## The table view

ID	Type	Title	Incoming	Outgoing
<i>CALC2</i>	demo	Exact Calcula- tor	<i>CALC_ADD; CALC_SUB; CALC_MULT; CALC_DIV; CALC2_1000ND</i>	<i>CALC1</i>
<i>CALC_ADD</i>	req	Generic Add	<i>CALC_TEST_ADD_1; CALC_TEST_ADD_2; CALC2_1000ND</i>	<i>CALC1; CALC2</i>
<i>CALC_SUB</i>	req	Generic Sub	<i>CALC_TEST_SUB_1; CALC2_1000ND</i>	<i>CALC1; CALC2</i>
<i>CALC_MULT</i>	req	Generic Multiply	<i>CALC_TEST_MULT_1; CALC2_1000ND</i>	<i>CALC1; CALC2</i>
<i>CALC_DIV</i>	req	Generic Divide	<i>CALC2_1000ND; CALC2_TEST_DIV_1</i>	<i>CALC1; CALC2</i>
<i>CALC2_1000ND</i>	spec	Big fractional numbers	<i>CALC_TEST_ADD_1; CALC_TEST_ADD_2; CALC_TEST_SUB_1; CALC_TEST_MULT_1; CALC2_TEST_DIV_1</i>	<i>CALC_ADD; CALC_SUB; CALC_MULT; CALC_DIV; CALC2</i>
<i>CALC_TEST_ADD</i>	test	Basic addition test		<i>CALC_ADD; CALC2_1000ND</i>
<i>CALC_TEST_ADD</i>	test	Big addition test		<i>CALC_ADD; CALC2_1000ND</i>
<i>CALC_TEST_SUB</i>	test	Subtract test		<i>CALC_SUB; CALC2_1000ND</i>
<i>CALC_TEST_MULT</i>	test	Multi- plication test		<i>CALC_MULT; CALC2_1000ND</i>
<i>CALC2_TEST_DIV</i>	test	DIV test (demo2 only)		<i>CALC_DIV; CALC2_1000ND</i>

## Lessons learned

1. The advantage of a table-view is that it will only grow in one direction: it just becomes a bit longer.
2. Even for a big project, it's a great page to bookmark and use as a start-page for all kinds of requirements.

Probably, you like to split it into the kind of 'need'.

It would be great to show a classical [Requirements Traceability matrix](#) (RTM) too. This table shows the relations between all the requirements and all the tests.

**Note:** As 'needs' currently does not support classical *RTMs*, I can't generate/show it here. See [a manually made TRM](#) to get the idea. As you will see: it's easy to read.

However, it's quite hard to grasp deep relations; then the tree above is more helpful.

## Next steps

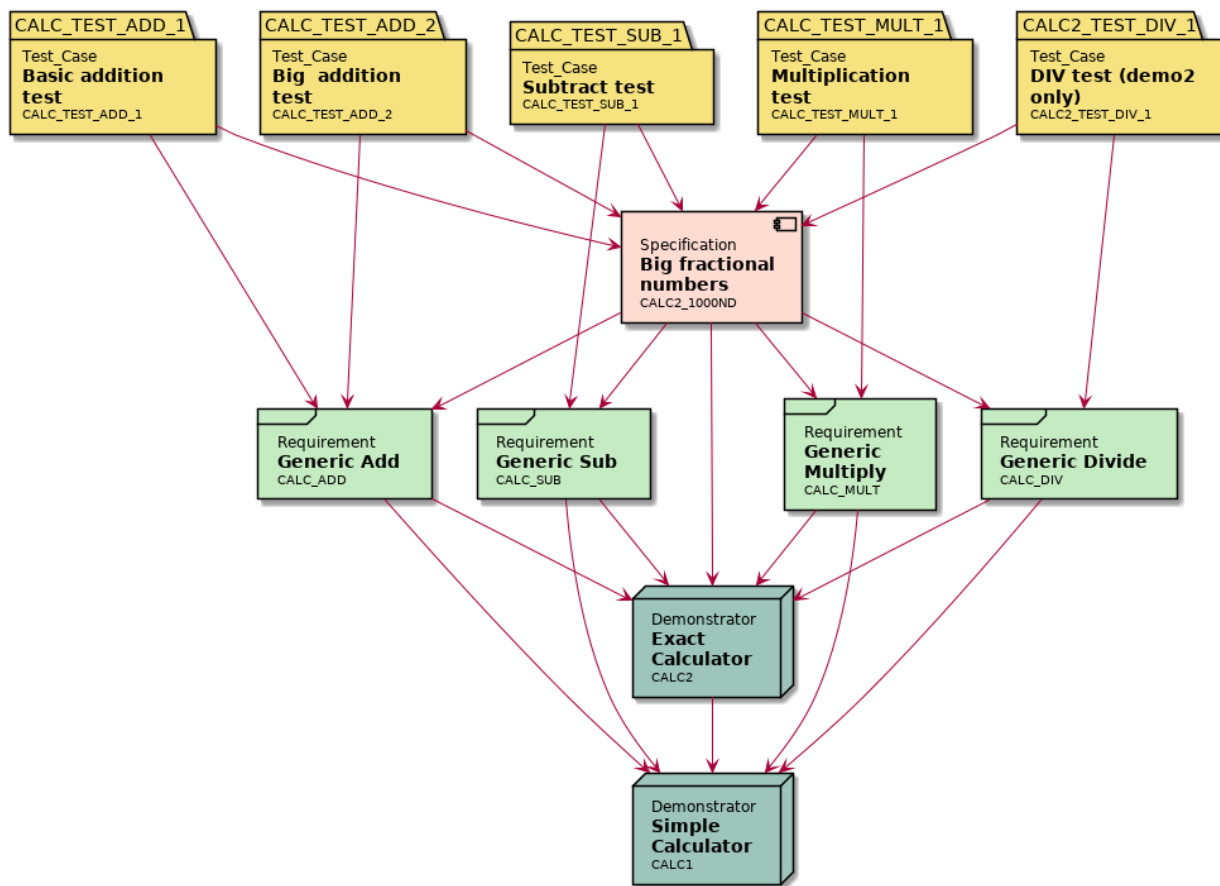
We might add more product-variants, or more sprints to convince you that requirement-traceability is important. The only effect is more pages with (trivial) requirements, specifications or other ‘needs’. And the same to generated overviews; the later cost only a small number of lines, independent of the size of the product. So we will leave that as an exercise.

As a bonus, we will show you one more generated graph: *All needs [BigTree]*, combining the Simple Calculator (CALC1) and Exact Calculator (CALC2) requirements.

Whenever you have more quistions, you can email me Albert.Mietus.

## All needs [BigTree]

We can also show the relations between all product (cq releases) and it *needs* in one big tree<sup>1</sup>.



<sup>1</sup> In general, and strictly speaking, this graph can be a “forrest”: a collection of trees. Mostly, people like to use the term “tree” anyhow. However as this chapter is about requirements and quality, and I’m trying to convince you to be strict, this footnote seems non-trivial ...

## Lessons learned

1. We have at least one test-case for each requirement; as we can clearly see.
2. The “puzzling” specification for **Exact Calculator (CALC2)** have an affinity with both the requirements and the test-cases
3. The (general) requirements for both calculators are equal.
4. Both calculators are akin; **Exact Calculator (CALC2)** is kind of a enhanced version of **Simple Calculator (CALC1)**

This “BigTree” gives a global overview of all ‘needs’ and there relations; which gives insight into product and how to test it. It can be generated when the requirements (and other ‘needs’) are affixed with their (outgoing) links.

Even when the number of ‘needs’ becomes huge, this graph can be drawn quite clear – although you make like to use a big slide of paper.

It will directly shown some “isolated” requirements or test, when you forgot to add a link. Also other “curious looking” parts of the drawing may need some attention. As it remarkable often denotes some mistakes.

## Footnotes & Links

### Footnotes & Links

**Needs:** <http://sphinxcontrib-needs.readthedocs.io>

**Sphinx:** <https://www.sphinx-doc.org/>

## Particulars of ‘needs’

The main objective of this blog is introducing “Requirements Traceability” in general; especially in an agile environment or when a roadmap of product-releases exists. That is done demoing 2 products, 4 requirements, 1 specification, and 5 tests. Also, some generated graphs and tables are shown.

Those ‘needs’ aren’t just text; they are ‘life’ defined and generated by a plugin called *needs* in the *sphinx-doc* documenting tool; which is just to generate this website. Partly, because I like this toolset. But also to show you that is can be done.

And despite this blog is not about (learning) a tool, this section will go a step deeper: It will show you how it is done; without becoming a tutorial on ‘sphinx’ or ‘needs’.

Possible, you like it and would love to play with the demo. Or, you are inquisitive whether this is truly *not text-only*.

The following pages show some excerpts of the “*RST-code*”, that is used to defined all ‘needs’. Feel free to study the ‘sources’ and learn from it.

I will even document the tricks needed for “*The forgotten test*”.

## Particulars for CALC1

---

**Hint:** You should skip the particulars passages, when you not interested in the technical details of this particular tool.

---

### A product (start of the V)

To define the [Simple Calculator \(CALC1\)](#) product the following is described:

```
.. demo:: Simple Calculator
  :ID: CALC1

  For this demo ...
```

You can see this product has the ID `CALC1` and some text. No links are needed, they will be added automatically by requirements, which are described “later”.

### With requirements (one step down into the V)

The requirement [Generic Add \(CALC\\_ADD\)](#) has an ID too, and a link to the products it is valid for; here *CALC1* and *CALC2*.

```
.. req:: Generic Add
  :ID: CALC_ADD
  :links: CALC1;CALC2

  All calculators ... able to sum ...
```

All requirements are described in the same way, as well as each individual requirement can be linked to one or more products (or product-variants). As this demo has (already) two products, and this requirement is valid for both; you see them listed here.

### And tests (the other side of the V)

A test (-case) is also a ‘need’. It is defined in the same approach: a title, an ID and the links to all requirements that are tested by this one. Here, that is only *CALC\_ADD*.

```
.. test:: Basic addition test
  :id: CALC_TEST_ADD_1
  :links: CALC_ADD

  Sum two numbers and verify ...
```

Again, the same construction is repeated for all tests.



## Tracing the requirements

Generating the “requirements tree” as displayed [here](#) is very easy:

```
.. needflow::
   :tags: demo1;general
```

Likewise is showing the table overview:

```
.. needtable::
   :tags: demo1;general
   :style: table
   :columns: id;type;title;incoming;outgoing
   :sort: type
```

See the documentations of needs (<https://sphinxcontrib-needs.readthedocs.io>) for details on all options.

## Particulars for CALC2

---

**Hint:** Again, you can skip the **particulars** passage when you have no curiosity in the technicalities of ‘needs’ itself.

---

## Describing requirements

The describing text of any requirement (in ‘needs’) is standard **rst** (*reStructuredText*). So it can use hyperlinks, forward-references to other needs and even warning-admonitions.

The full textual definition of **Exact Calculator (CALC2)** is:

```
.. demo:: Exact Calculator
   :ID: CALC2
   :tags: demo2
   :links: CALC1

   This calculator should work with `Fractional Numbers <https://en.wikipedia.org/wiki/Fraction\_\(mathematics\)>`_, and be
   exact for very big numbers; as defined in :need:`CALC2_1000ND`

   .. warning::

      This implies ``floats`` are not possible in the implementation
```

## The added specification

Like all other ‘needs’, the specification for **Big fractional numbers (CALC2\_1000ND)** is straightforward. It links to “earlier” requirements.

```
.. spec:: Big fractional numbers
   :id: CALC2_1000ND
   :links: CALC_ADD;CALC_SUB;CALC_MULT;CALC_DIV
   :tags: demo2
```

(continues on next page)

(continued from previous page)

The `:need:`CALC2`` ...

---

**Tip:**

- There is no *prescribed* order how the individual ‘needs’ can be linked. It kind of feels more natural to link to “higher level” (in the V-model) ‘needs’, and to one that are described “earlier” (in project-time). But when you can link them in any order.
- Similar, a ‘need’ can link to any other ‘need’, independent of its type.

Above we have used a *spec*, to add this requirement; but a normal *req* (requirement) is possible too. You can configure any kind of ‘needs’, as you like.

- You can even *export* ‘needs’ in one document and *import* them in another. For big projects with many levels of modules, and so, specification-documents, this is typical behaviour. In this small calculator example tha is not used.

---

## Tracing relations

To be able to trace whether some test need to be adapted, we only have to add some “links” between the relevant test and the additional (test) specification.

In [Big fractional numbers \(CALC2\\_1000ND\)](#) that is done by adding some (outgoing) links to the existing tests. You may have to open/click the see the details row.

---

**Note:** The incoming links are added automatically.

---

## Inheriting links

Currently, there is no *inherit option*, One can’t specify that the requirements for *CALC1* are also valid for *CALC2*.

- By linking the two `Demonstrators` we get (almost) the same.
- Alternatively, you can just add the links manually.
- (or you can use the *export/import option* and a *simple script* to modify the *json file*)

---

**Tip:** As ‘needs’ is an actively maintained open-source project, ‘inheriting’ may be added in a next release.

Even by you:-)

---

## The hotfix

See *the notes about the forgotten test* for the particulars on how to *forget* and *add* a test in one document.

## The forgotten test

Intentionally, one test is “*forgotten*” for the first demo. However it is needed and “hot-fixed” for *the second one*.

Surely this is not typical in one documentation-release.

Therefor some *tricks* are used to show this in one go. Those tricks are documented here.

By adding some ‘tags’ to the various requirements one can filter on those when generating an overview. And only show the intend ones. This can be useful by-example to labels ‘needs’ for a specific release or product-variant. Here, used a bit more extensive.

## Particulars

As typical each product has it “own” tag: demo1 or demo2. As we have some tests that should be selected on one page, but not on another we use an extra tag: general.

Most test-cases are labeled with general, as are the generic requirements. Whereas the “forgotten” test *DIV test* (demo2 only) (CALC2\_TEST\_DIV\_1) is labeled demo2

```
.. test:: Basic addition test
   :id: CALC_TEST_ADD_1
   :links: CALC_ADD;CALC2_1000ND
   :tags: general

   Sum two numbers ....

.. test:: DIV test (demo2 only)
   :id: CALC2_TEST_DIV_1
   :links: CALC_DIV; CALC2_1000ND
   :tags: demo2

   Subtract ...
```

Now it becomes possible to show the relations with, or without that test.

In the *first demo*, we filter on ‘demo1` and ‘general’. So we get the product, the generic requirements and most tests. But not the forgotten one.

```
.. needflow::
   :tags: demo1;general
```

After we have “hot-fixed” the test, we can simply select all test for the *second graph*

```
.. needflow::
   :tags: demo2;general
```

Actually, we didn’t really “hot fix” it; it was only defined for demo2. But linked to a general requirements.

Normally, you don’t need to use this kind of tricks; it better to not forget a test, or really fix it.

More info? Contact Albert: Albert.Mietus; +31 (0)6 526 592 60. Or use the comment section below.

## 1.3 Software competency

### 1.3.1 HTLD: *HighTech Linux Delivery*

Here you will find some blog-style articles about *HTLD*.

**Warning:** This section is a bit delayed.

#### De Embedded Linux Expert bestaat niet

**reading-time** 4 minuten

Regelmatig krijg ik ‘Embedded Linux Experts’ aangeboden; bijvoorbeeld voor het **HTLD**-team (*HighTech Linux Delivery*). Vaak zijn goede mensen, met Linux ervaring; maar toch niet de mensen die ik zoek. Het blijkt erg lastig voor niet direct betrokken, om de juiste experts te spotten. Daarom een paar hints waarmee je “de” Embedded Linux Expert herkent. Immers, alleen het woordje ‘Linux’ op een CV is onvoldoende!

Zo’n 15 jaar geleden was ik één van de mensen die “**Embedded Linux**” in Nederland introduceerden. Via diverse presentaties en artikelen resulteerde dat in “moderne” expertise. Ook was er een tool: EQSL; Embedded QuickStart Linux. Immers, voor veel ervaren *realtime/embedded* ontwikkelaars was Linux *toen* heel nieuw, heel groot & complex en vooral *verwarrend*. Met de EQSL-CD konden ze een toch snel een Embedded Linux systeem bouwen. Vanaf boekje lezen, tools en opties selecteren, Linux bouwen, het image installeren tot en met opstarten van een custom embedded Linux, en dat in minder dan een halve dag! Dat was in 2006 een hele prestatie; er waren immers niet veel *Embedded Linux experts*.

Opnieuw ben ik bezig met een Linux service, en dus aan het (uit)bouwen van team van Embedded Linux Experts. Nu met een ander 4-letter acroniem: **HTLD**; *HighTech Linux Delivery*. Er is veel belangstelling, en ook veel professionals willen graag meedoen.

#### Wat kan een Linux expert?

Linux wordt nu alom gebruikt en is nog steeds “sexy”. Toch is er nog steeds verwarring. Want wanneer ben je expert? Iemand die een standaard distributie kan installeren op een standaard PC, met twee of drie muis-kliks is duidelijk nog geen expert. Maar ga je al meetellen na duizend-uur, of pas na tienduizend? En hoe zit het met iemand met en 10-jaar ervaring, en Linux ervaring?

Er zijn waarschijnlijk meer dan één miljoen experts op gebied van Linux; waarvan duizenden in Nederland. Maar niet al die experts hebben expertise in het hetzelfde stukje Linux! De overgrote meerderheid richt zich op “klassieke IT”. Vaak zijn het geen *ontwikkelaars* maar *beheerders*. Erg handig om “high availability” clusters (of tegenwoordig: cloud computers) te configureren; maar wellicht niet bedreven in het ontwikkelen van een Linux-*driver*.

Een Embedded Linux Expert kan veel meer dan “Linux” gebruiken! Zo moet zij/hij **ook** een ontwikkelaar zijn. En nog veel meer ...

## Wat kan een Linux ontwikkelaar?

Er zijn heel veel ontwikkelaars, die verstand van Linux hebben. En opnieuw zijn ze niet gelijk. Globaal zijn er 3 soorten linux-ontwikkelaars; afhankelijk van wat ze ontwikkelen. De vraag is:

- 1) Maken ze applicaties die (toevallig, ook) op Linux draaien?
- 2) Gebruiken ze Linux als ontwikkelomgeving?
- 3) Of, ontwikkelen ze Linux zelf?

Ik ben vooral geïnteresseerd in het laatste type. Een Embedded Linux expert moet **ook** de source van Linux zelf beheersen.

De Expert die we zoeken heeft dus verstand van Linux, is ontwikkelaar, en heeft ervaring met de source-code van Linux zelf. Typisch hebben ze ook veel passie voor Linux: het is meer dan werk of een hobby; het is bijna een *life-style*. Een beetje Linux-ontwikkelaar compileert regelmatig zijn eigen kernel, direct van de kernel-sources; liefst met een zelf gebouwde *toolchain*<sup>9</sup>. Vaak vooral omdat het kan, niet omdat het moet; het is immers leuk!

## Wat kan een Embedded Linux Expert?

Is dat complex en ingewikkeld? Ja behoorlijk, voor de meeste mensen. Zoals gezegd, 15 jaar geleden was het zelfs (te) complex voor de meeste, ervaren embedded ontwikkelaars. Inmiddels zijn er duizenden “betaalde “vrijwilligers” die dit met liefde doen. Achter elke ‘distributie’<sup>4</sup> zijn Linux Experts druk bezig met het beter maken van de code, en die te compileren; zodat gebruikers die kunnen installeren op hun PC.

U leest het goed: de “PC”. De meeste distributies, en de meeste Linux ontwikkelaars richten zich primair op de PC. Dat kan een laptop zijn, of een server; of desnoods een cloud computer.

Een Embedded Linux Expert kan veel meer ...

Zij/hij heeft **ook** verstand van en ervaring met Embedded Software; dus van “drivers”<sup>6</sup>, “BSPs”<sup>7</sup> en “bootloaders”<sup>8</sup>. Een Embedded Linux Embedded kan **ook** “Cross Compileren”<sup>5</sup>, met “toolchains”<sup>9</sup> zoals “Buildroot”<sup>11</sup> of “Yocto”<sup>10</sup>; maar ook werken met “GnuMake”<sup>12</sup>, want (Embedded) Linux kan niet zonder “Makefiles”<sup>12</sup>.

## Daarmee herken je die expert!

Ja, dat zijn veel complexe termen bij elkaar. Behalve voor de expert die we zoeken; zijn/hij immers een expert!

Als iemand niet weet wat die termen betekenen, dan is het niet de expert die ik zoek!

Ook verwacht ik van een Embedded Linux Expert, dat zij de “kernel-sources”<sup>1</sup> kan vinden, evenals de “busybox”<sup>2</sup>, en **ook** de rest van “Linux-code”<sup>3</sup> kent.

Natuurlijk loop ik nu de kans dat iemand nu die “aangehaalde termen” in zijn CV gaat zetten. Maar dan heb ik altijd nog strikvragen zoals: “*Heb je ervaring met de meest gebruikte Embedded Linux processor?*”, om het kaf van het

<sup>9</sup> <https://en.wikipedia.org/wiki/Toolchain> Over de software-tools om software te maken en hun (complexe) relaties.

<sup>4</sup> Er zijn meer dan 500 distributies zoals: Ubuntu, Suse, RedHat, Gentoo, etc! Zie oa <https://nl.wikipedia.org/wiki/Linuxdistributie>

<sup>6</sup> <https://nl.wikipedia.org/wiki/Stuurprogramma> Over (device) drivers, ook wel stuurprogramma genoemd.

<sup>7</sup> [https://en.wikipedia.org/wiki/Board\\_support\\_package](https://en.wikipedia.org/wiki/Board_support_package) Over die software die nodig is om (Linux) op uw eigen computer-board te laten werken.

<sup>8</sup> <https://nl.wikipedia.org/wiki/Bootloader> Over software die alle software opstart

<sup>5</sup> [https://en.wikipedia.org/wiki/Cross\\_compiler](https://en.wikipedia.org/wiki/Cross_compiler) Over cross-compileren (“XCC”) en *Canadian Cross Compilers* (Engels).

<sup>11</sup> <https://buildroot.org/>; Buildroot, is een andere toolchain om Embedded Linux te bouwen

<sup>10</sup> <https://www.yoctoproject.org/>; Yocto, is een toolchain om ‘custom’ Linux te compileren

<sup>12</sup> Linux gebruikt GnuMake; zie [https://nl.wikipedia.org/wiki/Make\\_\(computerprogramma\)](https://nl.wikipedia.org/wiki/Make_(computerprogramma)) voor Makefile(s)

<sup>1</sup> De source van de Linux kernel: <https://www.kernel.org>

<sup>2</sup> Alle bekende Unix-tools in mini-uitvoering; vooral voor embedded systemen: <https://www.busybox.net>

<sup>3</sup> Veel andere Linux source code: <https://www.gnu.org>

koren te scheiden. Dat is overgens de “ARM”<sup>13</sup>-CPU, die in vrijwel elke smartphone zit. En daarmee heeft bijna iedereen ervaring. Toch?

### Meer weten?

Zou je expert willen worden bestudeer dan vooral die “aangehaalde termen”; hieronder staan een aantal links naar wikipedia; soms is die kennis alleen in het engels beschikbaar. Ook een paar oude publicaties zijn nog beschikbaar.

Daarnaast levert Googlen op “[Embedded Linux](#)” ruim 100M-hits op! Kennis zat dus, maar is het te vinden?

Ook zal ik komende tijd wat meer blogs publiceren over HTLD; zoals een artikel over waarom Linux-drivers vaak duurder worden dan gedacht. Hou deze plek daarom in de gaten.

Natuurlijk mag je ook altijd contact met me opnemen

— Albert.Mietus

---

### Footnotes & Links

Een paar (15 jaar) oude publicaties over Embedded Linux; ze zijn verouderd, maar soms nog verbazend actueel.

- [13 okt 2005] <https://bits-chips.nl/artikel/snelle-linux-overstap-begint-bij-toepassing/>
- [11 mei 2006] <https://bits-chips.nl/artikel/pts-bouwt-opstapje-naar-embedded-linux>
- [reprints ‘12] <http://albert.mietus.nl/read.IT/Proponotheek/index.html> (reeks van 4)

Een echte Embedded Linux Expert kent deze locaties uit zijn hoofd:

Enkele experts-termen volgens wikipedia:

When you are interested; you can contact me:

— Albert.Mietus; +31 (0)6 526 592 60

## 1.3.2 DesignWorkShops

### ThreadPoolExecutor

#### English & Dutch

This workshop is mostly in Dutch and partly in English; as it is based on an existing Dutch documentation. Only the new parts are in English. Hopefully, more and more part will be translated.

Deze workshop is gebaseerd op bestaande, Nederlandse teksten. Voorlopig zijn alleen de nieuwe delen in het Engels.

- You can use google-translate for now: [Translate](#) (STAGING branch).

**practice-time** 2 \* 1 hour

---

<sup>13</sup> De ARM CPU zit niet alleen in (vrijwel) alle smartphones; ook veel andere embedded systemen gebruiken deze. Zie <https://nl.wikipedia.org/wiki/ARM-architectuur>

This workshop is about the *ThreadPoolExecutor* (‘TPE’ for friends): a pool of workers implemented with Threads. This is a modern, advanced **design-pattern** available in many languages.

You will get an introduction to the concepts, to be able to use the ‘TPE’. Also, we study Python implementation, to practice **design-analyse**.

The example code is both available in Python and Java; more languages will follow. As the python-TPE is open-source and easy to read, that one is used for the design-analyse. In that exercise, you will analyse code to create some design-diagrams afterward.

Last, you may use this concept in a next DesignWorkShop . . .

As usual, this workshop has some sections named *Questionnaire*; those questions are meant to sharpen the mind. Thinking about the answer is more important than the answer itself.

Some exercises will have a possible elaboration; often that isn’t only good one. Again: think about it. And learn!

—Have fun, Albert

## Introduction

### Why concurrently?

Vaak is het wenselijk om (a) veel *soortgelijke taken tegelijkertijd* uit te voeren. Dit (b) op de *achtergrond* te doen; zodat het ‘hoofdprogramma’ beschikbaar blijft, bijvoorbeeld voor interactie. En (c), dit werk te verdelen meerdere processors; om zo de doorlooptijd te verkorten.

Deze *concurrency* kan met diverse primitieven gerealiseerd worden. Bijvoorbeeld met meerdere computer-nodes, met meerdere processen, of met meerdere *threads*. De juiste keuze is sterk afhankelijk van de relative (communicatie) overhead: hoe kleiner de taak, hoe minder overhead toegestaan is. Daarom zijn threads populair. Maar ook het opstarten van een thread kost tijd; te veel tijd voor de kleinste, repeterende taken. Het *WorkersPool* patroon is een geavanceerde, generieke aanpak, die de minste overhead kent. Zeker als deze geïmplementeerd word met threads. Dan mag de taak zo klein zijn als *één functie*!

### WorkersPools

Al zijn er ook vele andere redenen om een ‘WorkersPool’ te gebruiken. Zoals:

### Robuustheid

Door werk te verdelen over meerdere processen, of nodes, is een zekere ‘*redundancy*’ ingebouwd. Met behulp van een ‘broker’ kunnen mislukte (gecrashde) taken vaak opnieuw opgestart worden.

### Schaalbaar

Door (steeds) meer *werkers* in te zetten is de oplossing erg schaalbaar. Een bekend voorbeeld hiervan is de (apache) webserver. Deze maakt gebruik van het WorkersPool principe op meerdere niveaus. Zowel met threads & processen op één systeem, en middels ‘load-balancers’ tussen systemen en “in de cloud”.

## Eenvoudig

Hoewel de (thread) WorkersPool complex is om te implementeren, maakt een goede realisatie (lees: interface) het gebruik van concurrency eenvoudig. Een goed voorbeeld hiervan is [Grand Central Dispatch](#) van Apple. Deze technology is o.a. gebaseerd op de WorkersPool en goed geïntegreerd; waardoor vele (zelfs beginnende) programmeurs dit principe (onbewust) gebruiken.

GCD is ook bekend als *libdispatch* en beschikbaar voor o.a. [FreeBSD](#), [Linux](#) en onderdeel van de (open-source) [Swift-Libraries](#).

## Questionnaire

1. When is the overhead of starting a thread to high?
2. Why does a worker-pool (with threads) improve this?
3. Can you imagine other kind of worker-pools (So not with threads, but with ...). Give at least 2 others

## Concept

Elke **worker** werkt concurrent (tov elkaar en het hoofdprogramma) en kan meerdere keren een (soortgelijke) taak *na elkaar* uitvoeren. Zodra hij een taak krijgt toegewezen, werkt hij uitsluitend (en typisch zonder onderbreking) aan die taak en maakt het resultaat zo snel mogelijk beschikbaar. Daarna wacht hij op een nieuwe taak. Het aantal worker is typisch instelbaar; waarbij het de kunst is om altijd minimaal één worker vrij te hebben.

Een typisch implementatie heeft een queue voor nieuwe taken (de *work\_queue*), een queue met resultaten en een set van workers. Zodra een worker *beschikbaar* komt, pakt deze de volgende taak uit de *work\_queue*, gaat aan de slag en zet het resultaat in de tweede queue. Typisch in een *forever-loop*, waarin hij alleen pauzeert als er niets in de *work\_queue* staat.

Een wat uitgebreidere implementatie abstraheert van de queues. Die hebben immers locking nodig, wat kan afgeschermd worden in de nette API-functie. Ook worden steeds vaker **future**-objecten gebruikt; die bevatten “*het toekomstige resultaat van een asynchrone berekening*”. In sommige programmertalen kunnen de taken zelfs “inline” gedefinieerd worden. Dit alles maakt het concept complexer, maar handiger in gebruik.

Fig.  
1:  
Concept  
met  
Threads,  
afkomstig  
van  
wiki-  
media

## The TPE API

Conceptually, a TPE is very easy to use. There are two main classes; the *TPE* itself, and some *FutureObjects*.

### **class TPE**

Abstraheert een *WorkersPool*, inclusief queues, etc.

*Gemoduleerd naar de Python implementatie ``ThreadPoolExecutor``; maar veel eenvoudiger.*

Het (maximaal) aantal worker moet bij het instantiëren van de class opgegeven worden.

**submit** (*fn*, *\*args*, *\*\*kwargs*)

Voor een nieuwe taak op en return een *FutureObject*. De functie *fn* zal asynchroon uitgevoerd zal worden als *fn(\*args, \*\*kwargs)*.

### **class FutureObject**

*Gemoduleerd naar de Python implementatie ``Future``; maar veel eenvoudiger.*

**done** ()

Return *True* als de taak afgerond is; anders *False*. Wacht nooit op een taak.



**result** (*timeout=0*)

Return het resultaat van de bijbehorende taak, als dat al beschikbaar is. Als die taak nog niet afgerond is, dat zal daar maximaal *timeout* seconden op gewacht worden. Als de taak dan nog niet afgerond is, zal *None* terug geven worden

## Use (Examples)

This entry gives a very simple example, in various languages.

The key methods are *TPE.submit()* and *FutureObject.result()* [*.get()* in some languages]. Also *FutureObject.done()* [or *.isDone*] is a important method.

There are many more methods, depending on the implementation. Bit without using the above ones, you not using it properly.

There are basically three steps:

1. Create the TPE object. This will start the workers (threads) and do the plumbing
2. Submit *callable*s (functions) to the TPE. Which will distribute it to an available worker, or queue it temporally.
3. Use the result **later** by reading the returned *FutureObject*.

When calling *TPE.submit()*, a *FutureObject* is returned **directly**. This is kind of a placeholder; it doesn't contain the result yet! When the submitted function (eventually) returns, the return-value is stored in that placeholder.

One call monitor the availability of the result by *FutureObject.done()*; only when it is *True* the result is available. Or, one just wait for it by calling *FutureObject.result()*. Surely, then it is done.

Note: one can also give a wait-limit to *result()*. See you language and/or implementation documentation for details

## Python

```
#!/ python
import time, random
from concurrent.futures import ThreadPoolExecutor as TCE

TRIES=10
MAX=6

def taak(geal):
    time.sleep(geal/MAX)          # Simulate a complicated calculations ...
    return geal, geal * geal      # ... returning the number and it square

def demo():
    workers = TCE(max_workers=4)  # (1)
    results = {}

    for n in range(TRIES):        # (2) Submit all tasks; (with random number)
        results[n] = workers.submit(taak, random.randint(1, MAX)) # Store the Futures

    for n, f in results.items():  # (3) Print the results (in order)
        done = f.done()
        print("{n}: {g} ==> {r} ({done})".format(n=n, g=f.result()[0], r=f.
        ↪ result()[1],
```

(continues on next page)

(continued from previous page)

```

done="direct" if done else "waited"))

if __name__ == '__main__':
    demo()

```

You can run this examples by (using python-3)

```

[albert@MESS:1]% python examples/TPE_demo_1.py
0: 5 ==> 25 (waited)
1: 3 ==> 9 (direct)
2: 6 ==> 36 (waited)
#...

```

## Java

```

import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.Future;
import java.util.concurrent.ExecutionException;

public class TPE_demo_2 {

    private static final int TRIES      = 10;
    private static final int WORKERS    = 4;

    public static void main(String[] args) {
        demo();
    }

    private static void demo() {
        ThreadPoolExecutor workers = (ThreadPoolExecutor) Executors.
↳newFixedThreadPool(WORKERS); /* (1)
        List<Future<Long>> resultList = new ArrayList<>();

        /* (2) Sumbit all task, with a random input
        for (int i = 1; i <= TRIES; i++) {
            Future<Long> future = workers.submit(new DemoTask((long) (Math.random() *
↳10)));
            resultList.add(future); // And save future result
        }

        /* (3) Now print all results; wait on the when needed [.get() does]
        for(Future<Long> f : resultList) {
            try {
                System.out.println("Future done (Y/N)? :" + f.isDone() + ".\tResult_
↳is: " + f.get());
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        /** Stop the workers
        workers.shutdown();
    }
}

class DemoTask implements Callable {
    Long number;

    public DemoTask(Long number) {
        this.number = number; // JAVA: save the input of this
        ↪ "function"
    }

    public Long call() {
        try {
            TimeUnit.SECONDS.sleep(number/4); // Simulate a complicated calculations
            ↪ ...
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return number * number; // ... returning the square
    }
}

```

To compile this example, compile it first (int the examples dir)

```

[albert@MESS:2]% javac TPE_demo.java
[albert@MESS:3]% java TPE_demo_2 DemoTask
Future done (Y/N)? :true.    Result is: 1
Future done (Y/N)? :false.   Result is: 49
Future done (Y/N)? :false.   Result is: 64
Future done (Y/N)? :true.    Result is: 9
#...

```

## Exercise

De eersten opgaven gaan over het gebruik van een *WorkersPool*. Het zijn een paar eenvoudige, korte vragen om de hersens op te warmen.

NB Hoewel er een random-factor in het programma zit, zijn de antwoorden voorspelbaar!

Use the *python demo* as reference.

1. Bestudeer het voorbeeld-programma.
  - a. Wat is het (geprinte) resultaat?
  - b. Hoelang draait het programma (ongeveer)?
  - c. Wat kun je zeggen over wanneer ‘waited’ en wanneer ‘direct’ geprint word?
2. Waarom is een *WorkersPool* oplossing efficiënter (qua processor belasting en generieke overhead) dan een oplossing waarbij threads direct gebruikt worden?

- a. Wat kun je zeggen over de relatie tussen het aantal workers en het aantal cores/processors op je systeem? Is er een relatie?
3. Can you port this *use example* to C/C++
  - a. Plz give it a try. You will find the links to the documentation in *ThreadPoolExecutor* (at the ref:bottom <TPE\_links>)

## Design Analyse

In this second part of the workshop, we are going to study the **design** of the TPE.

Designing a TPE is (too) complex; that is not within the scope of this *DesignWorkShops*. Though understanding the design is...

A great (embedded) Software Designer should be able to extract the design from the existing code, even when it is undocumented and badly described. Especially then, your skills are vital.

So, let us practice them.

You are going to create “the” design of the python-implementation of the TPE. Both the static and dynamic (behavioural) UML-diagrams are needed.

---

### Note:

- This exercise is *not* about the cleanness of the UML-diagrams; it about the design itself.
  - Use a “whiteboard” to make the designs, not a fancy tool! When coworkers do understand it will do!
  - Optionally, you can eventually (later!) draw them nicely in plantUML.
- 

## Static Analyse

### Class Diagrams

#### Future (object)

Create the class diagrams for the python **Future (object)**.

---

### Note:

- The term *future* is used multiple times in python. You need the one in `concurrent`-package!
  - Possible to your surprise, there is no “thread” specific one. Only the *base* version exists, which works for threads and processes.
  - See file `concurrent.futures._base.py`: [https://github.com/python/cpython/blob/master/Lib/concurrent/futures/\\_base.py](https://github.com/python/cpython/blob/master/Lib/concurrent/futures/_base.py).
-

## TPE

Create the class diagrams for the python *ThreadPoolExecutor*.

The code can be found online: <https://github.com/python/cpython/blob/master/Lib/concurrent/futures/thread.py>.

## Other diagrams

Create all other (static) diagrams that you may find useful.

## Questionnaire

1. Why is the Futures class-diagram so much simpler are the ThreadPoolExecutor one?
  2. Which class is more important? Conceptually?
- 

### See also:

- *Pyreversed class diagrams* for my elaboration on the static analyse

## Dynamic Analyse

Create the dynamic (behavioural) UML diagrams that are needed to define the proper use of (a simplified) TPE.

Again, use the python implementation, but restrict yourself to the interface as defined in *The TPE API*

---

### See also:

- *DesignBoard* for my elaboration to come to a sequence-diagram
- 

### Tip: WoW

1. Use the (latest) Python-3 code; it quite good readable; eve when you are not a python-expert
  - Take the simplified (conceptual): ref: *TPE\_API* as a starting point; additional functionality that the real code included does not have to be included. Of course essential things must being described. The choice of “essential” is yours.
  - You may use the documentation of all classes that used; when needed.
2. Start with a short quick analyse, make notes and try to understand it. Then continue with other parts of the analyse and repeat.
  - Don’t use “UML-tool” in the first few steps; possible not even UML-diagrams.
  - Simplify the code (on a whiteboard), use lines & arrows to show values (parameters, key-variable, ect) are *passed around*.
  - Guess the important parts, like “somewhere the return values should be stored”, and “where are the threads created”; and make sure the are on that sketch
3. Summary all in (draft) UML diagrams. Check it is complete and not to detailed
  - Remove all details from the design, that aren’t essential to the design itself. And a part of the freedom to implement it (the programmer)

---

### Done?

Show this *minimal design* to a peer-group. And ask them to explain the design.

- You are not allowed to explain it, nor to answer questions!
- Whenever there are questions, there are unclear parts!

Then you should know: You have to improve your design!

---

### Elaboration

This last section of the *ThreadPoolExecutor DesignWorkShops* contain some elaboration and answer of me.

### Pyreversed class diagrams

Extracting class diagrams from code is easy; there exist many tools. For python, `pyreversed` is the most used one.

The readability of the diagram is strongly depends on the tool, the selected options, and mostly: the **quality** of the design/code. Do not expect nice diagrams, when somebody created the code without a proper design!

### `concurrent.futures.thread.ThreadPoolExecutor`

The class-diagrams below are automatically generated; using *pyreverse*. Two analyses are made:

- Showing all details (right)
- Showing only the “public” parts

PUB_ONLY (DEFAULT)	ALL

### Questionnaire

1. Which of the two diagrams are the most useful?
  2. How *bad* (if at all) is it to have “too many” details”?
  3. Which details should not be included? (When you do the analyse manually)
- 

### See also:

#### Docs

- <https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor>
- <https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.Future>

#### Code

- <https://github.com/python/cpython/blob/master/Lib/concurrent/futures/thread.py>

- [https://github.com/python/cpython/blob/master/Lib/concurrent/futures/\\_base.py](https://github.com/python/cpython/blob/master/Lib/concurrent/futures/_base.py)

**Version** The Python3.4.1 code is used to generate the diagrams above.

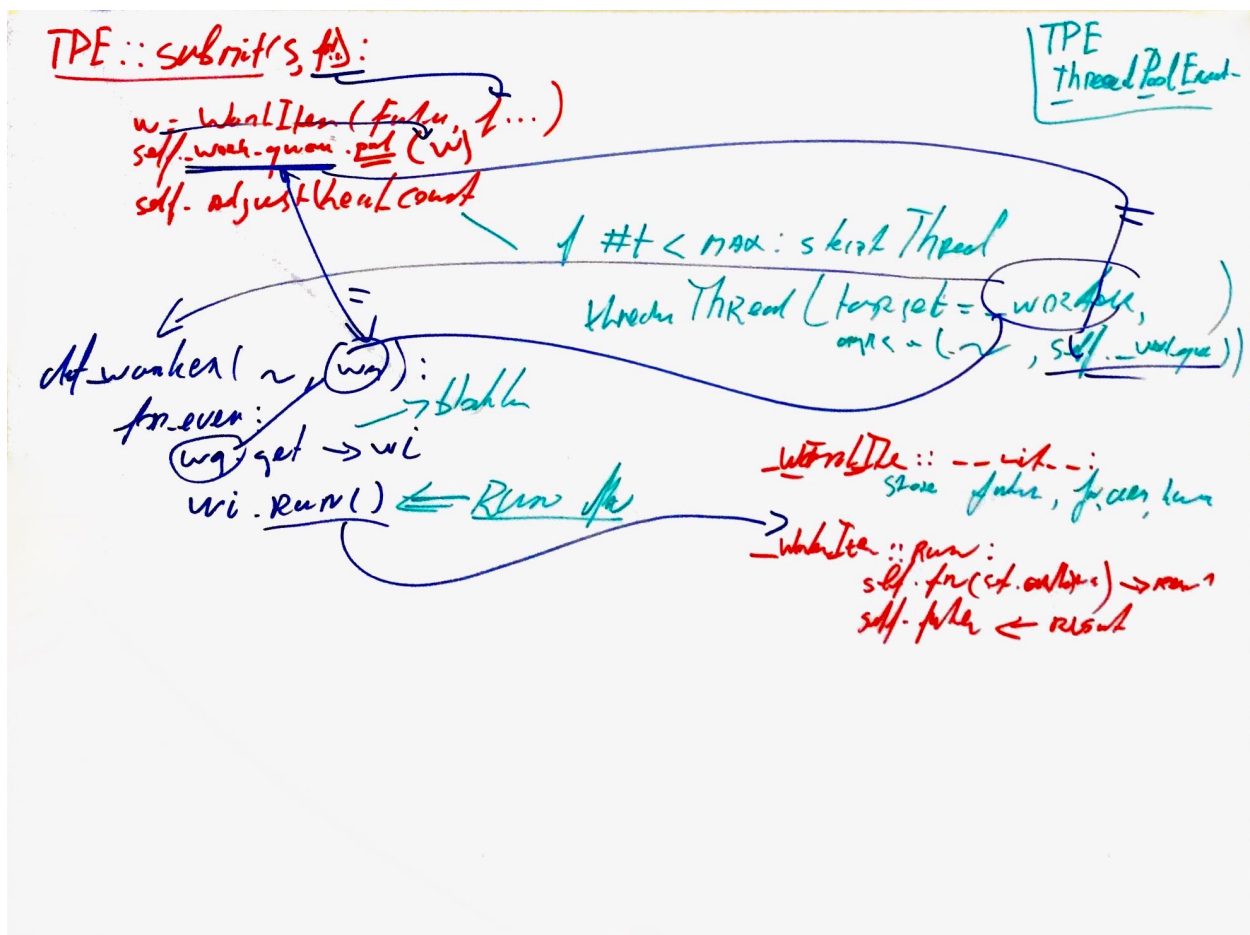
## DesignBoard

The dynamic behaviour can't be extracted from the code; one has to *run* it. Often this is done by manually following the most important functions. (So, not by executing it – that gives way to many details).

A very convenient procedure is to write-down the essential lines of the method. And of the methods that are being called. Use “arrows” to denote the (data & control) flow and to connect the pieces. Elaborate when needed.

By using a big whiteboard and many colors (and a wiper!) the “flow” can be visualised.

See my analyse in this picture below.



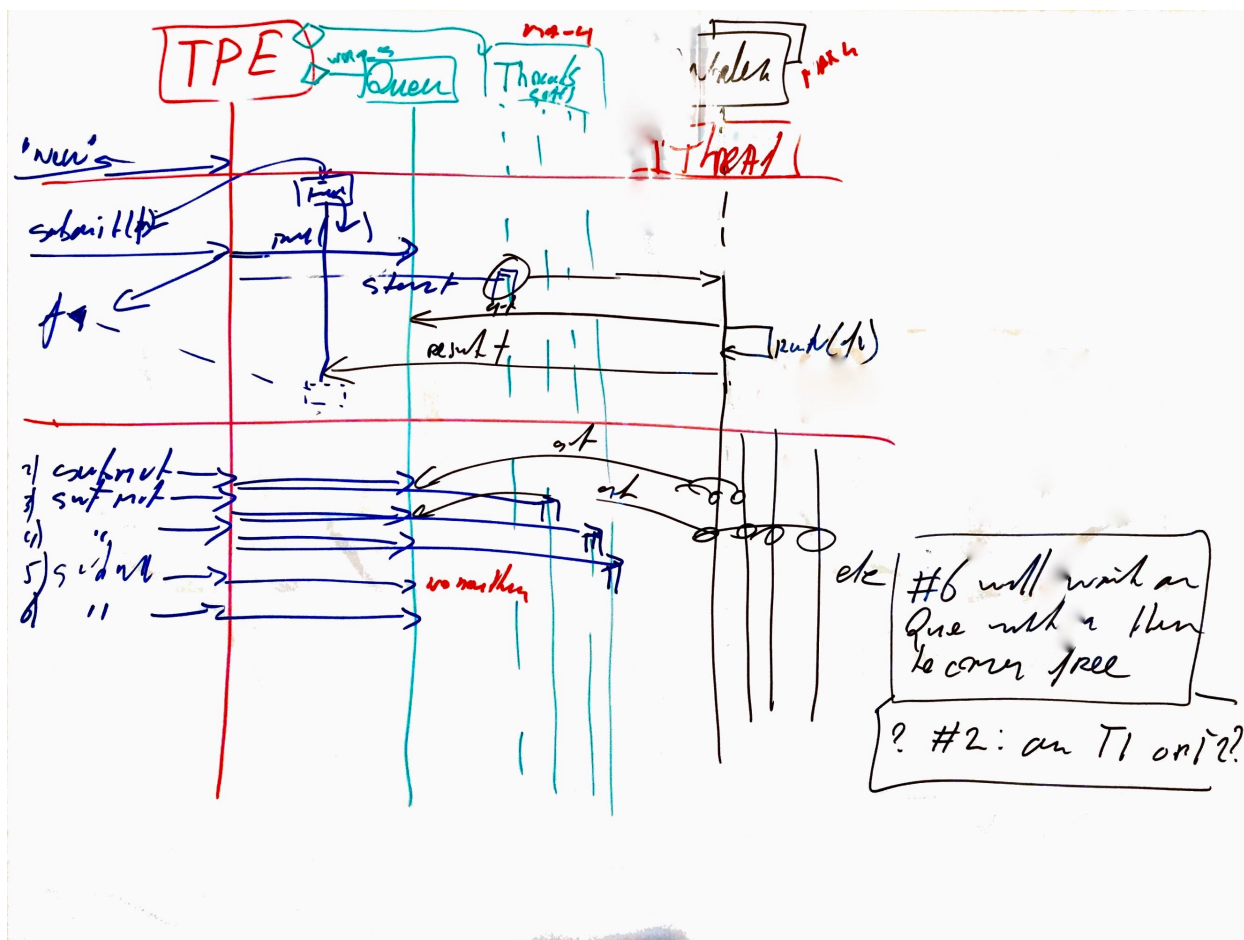
Next, this “curly flow” can be converted to a sequence-diagram. Preferable on a 2nd whiteboard. As shown below.

Last, and optional, we can convert it to (plant)UML:

TPE-sequence-backup

See also:

**ThreadPoolExecutor (python)** De referentie voor deze opdracht. <https://docs.python.org/3/library/concurrent.futures.html#threadpoolexecutor>





**Future-Objects (python)** Meer over (python) future-objects is te vinden op: <https://docs.python.org/3/library/concurrent.futures.html#future-objects>

**ProcessPoolExecutor (python)** De proces variant heeft (vrijwel) dezelfde interface (en gezamenlijke code). Voor liefhebbers is de documentatie te vinden op: <https://docs.python.org/3/library/concurrent.futures.html#processpoolexecutor>

## Java

- Ook Java heeft een `ThreadPoolExecutor`; zie: <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html>
- En kent het Future-object, (o.a.) als interface. Zie: <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html>

## C#

- In C# bestaan wel Thread-Pools: <https://docs.microsoft.com/en-gb/dotnet/articles/csharp/programming-guide/concepts/threading/thread-pooling>
- Maar lijken Future-object niet te bestaan. Zie in bovenstaande de opmerkingen over .. *and return values*

## C++

- Er lijkt weinig ondersteuning in C++ voor worker-pools en futures.
- *Boost* (<http://www.boost.org>) levert wel een aantal oplossingen (als library-code). Onderstaande links verwijzen naar de documentatie daarvan.
  - `basic_thread_pool`
  - `futures`

**Blocks in C** Met deze C-extensie zijn functie inlines te definiëren. Het zijn een soort van *lambda*-expressies, die gebruikt worden als argument in `GCD.dispatch()` functies. O.a de *CLang* ondersteund dit.

Zie [https://en.wikipedia.org/wiki/Blocks\\_\(C\\_language\\_extension\)](https://en.wikipedia.org/wiki/Blocks_(C_language_extension))

---

**Hint:** This extension does need a *runtime* (library) too.

---

## Pub/Sub

### HighTech vs Gooogling

When you search for *Pub/Sub*, you will find over 312M hits! Most are about cloud-computing and on how to exchange messages between servers via a '*broker*'.

Often it is infrastructure engineering based.

This workshop isn't about that. We focus on *Modern, Embedded Software Systems*: How can a (HighTech) Software Engineer use this pattern to create better software.

**practice-time** 2 \* 1 hour

The publish-subscribe *architecture-pattern* is very popular to route messages, according to *WikiPedia*. It is used much more generic, however. This workshop shows how to use it in an embedded application, or another single process application.

Then it becomes a HighTech (embedded) software **design pattern**.

We start with a simple implementation of **Pub/Sub**. You can experiment with that (Python) code and/or port it to your favorite language. Then, you are asked to **design** a cached, distributed version. Which is a bit more ambitious. And even though the result is probably *pointless*, it's a great Design-Exercise and a lot of fun!

It will help you to understand Pub/Sub, help you to use it in embedded software. Eventually, you may need Pub/Sub in a distributed environment. Then, it is better to use one of the existing ones, like **DDS** (Data Distribution Service); which more efficient and even RealTime!

But, by knowing how to implement it and are able to design it, will help you to use it for the better

## Introduction

An almost trivial example of PubSub is the daily newspaper:

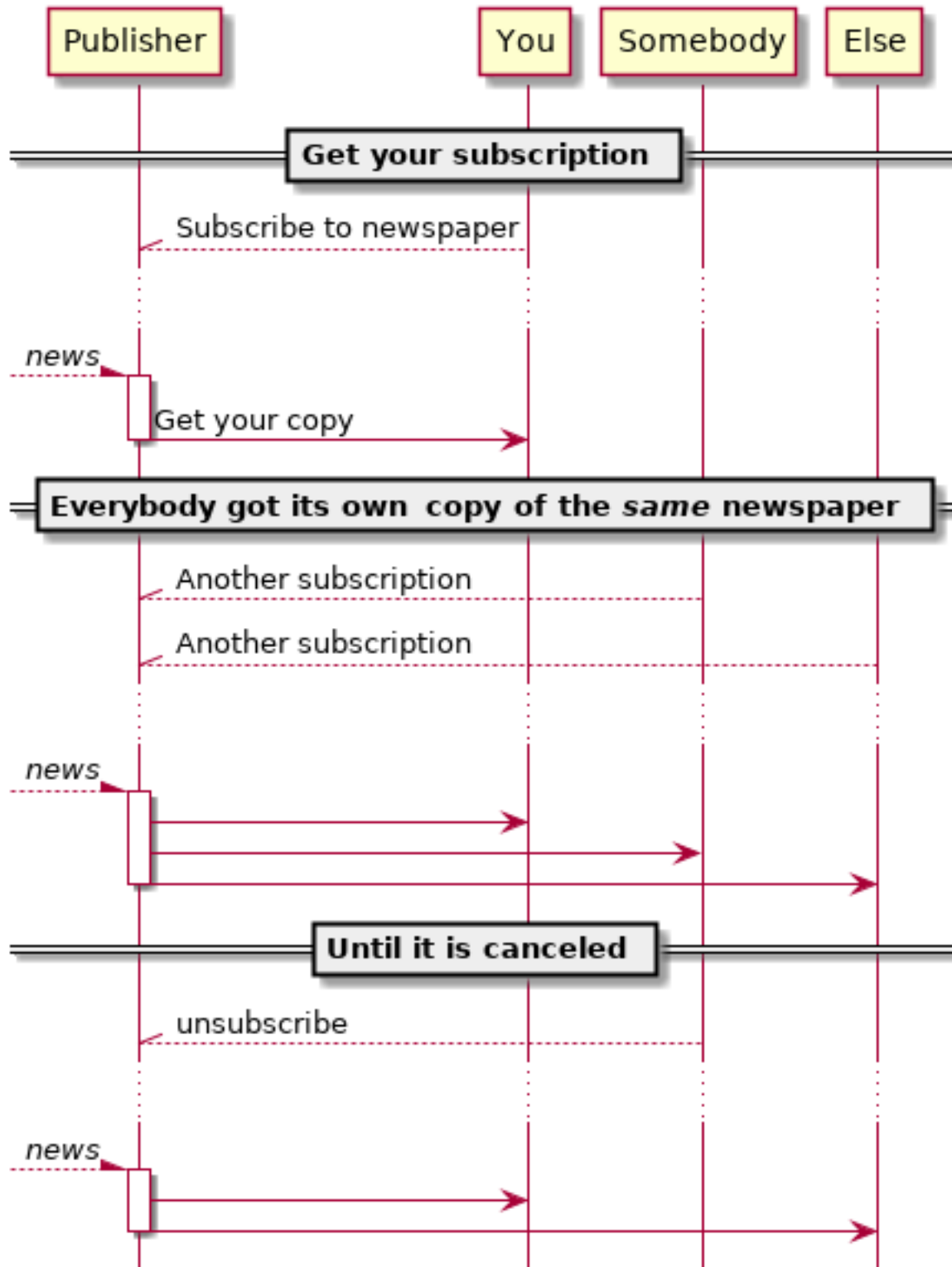
Once you have a subscription, you automatically get each new publication.

Typically, you are not the only subscriber; the *same* 'newspaper' is sent to all subscribers: everybody got his *copy*. And, whenever somebody cancels his subscription, all others still get there update daily.

Also notice: your copy is independent of that the neighbors. And, until you subscribe, the publisher does not know you.

In software-engineering, we call that **uncoupled** or *loosely coupled*, and that is great.

As we will explain in the next page.



## Advantages

For software-engineers, Pub/Sub has many advantages. The most obvious one is *decoupling*, another one is *scaling*. It is also simple to *Use*.

And, with Pub/Sub you will automatically use ‘*Dependency Inversion*’, one of the **SOLID** principles; as well as ‘*Dependency Injection*’, which often simplifies testing.

## Coupling

By and large, software-routines have to pass information. From one function to another, from one class to another, or from one module to some other module(s). Especially this latter case is annoying when it is implemented by calling a method of that other module. Then, we speak about **tight** (and *static*) **coupling**: the module effectively can’t perform without the other. When that “other” module is a stable, generic *library*, it is often considered as acceptable. Although it can disturb your (unit)-testing; by making it slow.

But how about two modules, that are under construction?

Then, both are not “stable” (as they might develop) and being dependent on unstable modules is bad. You can’t test independently, you may need to revise when the other is updated, etc. Well, you know the troubles . . .

To overcome this, the modules should be *uncoupled* or *loosely coupled*: Both modules are not allowed to call a function/method of the other one. (Which is easy:-). But still, pass information; which might seem impossible at first.

This is possible, as the modules do not depend on each other; instead, they both depend on the generic *pubsub*. *Topic*, as we can see on the *next page*

## Scaling

Now and then the same data is needed by multiple “consumers”. That number of “users” may even grow in future releases. A sensor-value, by example, that was initially only used in one or two routines, may become relevant input to many new, fancy features.

Imagine a module that handles (pushing) the brake. Initially, it was only needed to slow down the car. Nowadays it will switch off the cruise control, also. Perhaps, in the future, that same data might influence the volume of the radio; or is needed to automatically “e-call” 112, when there is a serious road accident. Or . . .

With Pub/Sub, it is easy to distribute data to more and more modules. Even to modules that aren’t yet imagined when you write that sensor-routine! Your current module only has to use `pubsub.Topic.publish()`, and that future module can get that data using `pubsub.Topic.subscribe()`; easy!

## Questionnaire

We have shortly introduced two advantages, now you have to *think*

1. Are there **disadvantages** to this *design-pattern*?
2. Can you mention some other **advantages**?

## Use

As typical with *Design Patterns* there are many ways to implement it. Here we focus on the “Topic” approach. It decouples modules by defining a generic interface and act as a kind of “man-in-the-middle”.

Both the Publisher and the Subscribers share a common *Topic* instance:

```
from pubsub import Topic
t = Topic("Just a demo")
```

## Publishers

Publishing a value is very simple; assuming *t* is a *pubsub.Topic* instance:

```
t.publish("Hello World")
```

## Subscribers

Each subscriber should register a callback, which will be called “automagical” when a new value is available:

```
t.subscribe(demo_cb)
```

Where *t* is topic (an instance of *pubsub.Topic*) and *demo\_cb* is the *callback*. This can a function or other kind of callable. Multiple subscriptions are possible, by registering another:

```
oo = Demo()
t.subscribe(oo.demo_oo_cb)
```

## callbacks

A callback is a callable, that should process the new value. In essence, it is just a function (or method) with the correct signature. A trivial example is:

```
def demo_cb(value, topic):
    print("Function-Demo:: Topic: %s has got the value: %s" %(topic, value))
```

It can also be a method when you prefer an OO-style:

```
class Demo:

    def demo_oo_cb(self, val, topic):
        print("Method-demo: I (%s) got '%s' from topic %s" %(self, val, topic))
```

## See also:

- *pubsub.AbstractType.Publisher*
- *pubsub.AbstractType.Subscriber*
- *CallBack Signature*

## Threads

You might be wondering about threads: are they needed, essential or even possible?

The simple answer is: It's an "(I) don't care!"

It is also depending on the implementation. The shown implementation does not need, nor use threads. Remember, the (main) goal is to **decouple** (modules) and make it a *scalable* solution. Effectively, the `Publisher` is calling the *callback* of the `Subscribers` (in a loop); like in a conventional, *direct call* solution.

That *callback* will run in the same thread as the `Publisher`, though it can schedule some work on another thread. For example, with a *ThreadPoolExecutor*.

Notwithstanding, it might be beneficial to include a *ThreadPoolExecutor* (or any other concurrency concept) within the implementation of `Topic`. Then, the runtime of `t.publish()` can be controlled; even become *RealTime*.

## Questionnaire

1. Why can the runtime of `t.publish` be unbound?

Give an example. (in this implementation).

2. Why isn't a threading implementation **not** always better?

Give an example of on when `t.publish()` with threads is slower as the current one

## The Pub/Sub API

This is the API (and implementation) of a simple *Topic* class; as used in *Use*.

## Topic

### **class** pubsub.Topic

A *Topic* is like a channel to distribute information (events), in a **Pub/Sub** environment.

This will decouple the *pubsub.AbstractType.Publisher* from the *pubsub.AbstractType.Subscriber* (in both directions).

- On one side is a `Publisher` that provides 'data' (**value**'s, *events*, ...).
- The other side has `Subscribers` who subscribe to the topic (with **callbacks**).
- Both `Publisher` and `Subscriber` are abstract types; there is no concrete class (needed).
- Any module that calls `publish()` is called a `Publisher`; likewise, a `Subscriber` is anyone calling `subscribe()`.
- Commonly there is only one `Publisher` (for a given `Topic`); that is not mandatory, however.

### **Topic.publish(value, force=False) :**

This method is called by the `Publisher`, whenever a new **value** is to be shared. When **force** is *False* (default), the **value** will only be distributed when it differs from the previous **value**. To force distribution, set **force** to *True*.

### **Topic.subscribe(callback) :**

This method is called by all `Subscribers` to *register* a **callback**, which is called on new 'data'.

The passed **callback** (any *callable*, like a *callback\_function\_type()* *callback\_method\_type()*) will be called when 'data' is available. It has a signature like:

```
def callback([self,] value, topic):
```

Where **value** is the ‘data’ passed to `publish()` and ‘**topic**’ is the *Topic* instance, use to route it.

When the callback is a method, the *self* parameter is automatically remembered by Python. For function-callbacks, leave it out.

## Supporting types

Both the Publisher and the Subscribers are *Abstract Types*.

**class** pubsub.AbstractType.Publisher

Any code that will call `pubsub.Topic.publish()`. It can be a class, a module or just code ...

**class** pubsub.AbstractType.Subscriber

Everybody calling `pubsub.Topic.subscribe()`. Typically, the *Subscriber* has a **callback** function/method too. See *callbacks* for an example.

Often, it has a method that acts as the callback.

## callbacks

The generic signature for callbacks is simple:

`pubsub.AbstractType.callback_function_type(value, topic)`

`pubsub.AbstractType.callback_method_type(self, value, topic)`

## Demo (life)

This section gives a demonstration of a simple (python) implementation of *pubsub.Topic*. It is available in two forms:

1. A *revealjs* slide-deck of a Jupiter/IPython notebook
2. An interactive notebook. This is the same notebook, which you can edit and run on public “[binder](#)”

You can also [download the notebook](#), it is available on GitHub.

---

### Note: Same source, variable version

Due to practical details, the slide-deck below and the interactive notebook might be out-of-sync. Although it coming from the same source, it can be another revision; making the slides is partial manual.

---

## Slides

---

### Tip:

- Use ‘**space**’ to go to the next slide.
  - Use ‘?’ to see all keyboard *shortcuts*.
  - Click on the edge to open it in a new tab (and use ‘F’ for full-screen).
- 

## Interactive notebooks

Run-on binder: [try it yourself](#)

---

**Tip:** Or download [Jupyter](#); e.g by [Anaconda](#) (especially for Windows), or `pip`

Anaconda downloads: (*v2020.02;Python-3.7*)

- [Window-64](#).
  - [macOS](#).
  - For other/newer releases, and other platforms, see <https://www.anaconda.com>
- 

And play with it on your PC using this [notebook](#). With the benefit, you can save your changes.

Surely, you can also Copy/Past the code, and use your favourite editor.

## Practice

This last section of this [Pub/Sub](#) workshop is short. It only contain some practice (ideas). Two are on practicing your design skills, the other two are more coding-oriented.

## Design Analyse

Given the python implementation, including the shown “Use-Cases”: Analyse the design in detail:

1. Make a (quick) static design-analyse.

Resulting in some package- and class- diagrams.

2. Make a (draft) dynamic design-analyse.

At least a sequence-diagram for each of the “use-cases”



## Port to C/C++

### 3. Can you port (or re-implement) the python examples to C/C++?

Surely, you have to change some details; as a generic data-type (“value”) is not available. It is fine, to use a *string-type*. And just “print” it in the demo-callbacks (like I did).

## Design a cached, distributed one

The shown (conceptional) implementation works for a single process; optional with threads. In this exercise, you are going to extent that for “network use”; although it will be a simple, conceptional one. Many existing protocols and frameworks do exist already! The goal is *not* to challenge them, nor to *use* them.

The goal is to practice your design skills!

So, this is a (fun) design-exercise. You should be able to make a (full, conceptional) design in about an hour. That does imply many fancy options should be left-out:-)

4. Extent the current interface to allow pub/sub between multiple processes; optionally running on multiple computers (on the same network).
  - a. The current API (*Topic*, `publish()` & `subscribe()`) is not allowed to change. Adding parameters to the class initiation (“the constructor” in C++) is allowed. Adding extra methods is also allowed (but see below!).
  - b. All existing Use-Cases **should** keep working (both the shown one, as many others).
    - i. The main methods (`publish()` & `subscribe()`) **should** remain *exactly* the same.  
No more parameters!
    - ii. The default behavior **should** be “local” (not distributed).
  - c. There is no requirement for performance. But it is expected that a second “network-get” **will** be resolved locally. So, use a cache to minimize networking
  - d. The networking **should** use standard TCP/IP networking (“sockets”). No other network libraries/frameworks are allowed.
    - A simple “serialise” **will** do. Assume, all computers/processes use the same endianness and other encodings.
    - Again, use “strings” (only); then this part is easy.

---

### Hint: *Deamon* & *lib*

An easy way to design this is to foresee *one* processes handle all the administration (the core of *Topic*); including “calling” all the callbacks.

This is typically called a **daemon**, or *services* on Windows.

To hide all the networking stuff, arrange a (small) library, that acts as *facade* and provides the (extended) *The Pub/Sub API*.

---

## Implement it

5. To check your design above is great, now implement it.

Or better: implement the design of you co-trainee, and ask him to implement yours!

Remember, a design is a communication-tool: A great design contains exactly those details that your coworker needs to implement is as it is meant to be, but no more. (S)He should have some freedom to optimize implementation-details.

## Internal notes

---

### Todo:

- Use ‘needs’ to specify the requirements
- 

## 1.4 PyMESS: MESS with python

### 1.4.1 Training snippets

#### dPID: A Python ‘homework’ exercise

**practice-time** 1 hour

This is an optional exercise for the python-3 workshops: program a **discrete PID**-controller.

A basic *class definition* is given; which has to be tested and implemented. By starting with the *test-part*, which is advisable anyhow (the TDD approach), the exercise starts simple.

A few *test-examples* are also given. This file can be used as ‘*template*’ to write your own tests.

#### dPID articles

##### dPID: The dPID class

**status** RC-1.0

This article shows the `dpid.PID` documentation, as specified in *the python file*.

It specifies the *interface* to a discrete-PID-controller. Study it to understand how the class should be used. Then start writing python code to test it.

#### **Attention:** Its’ a **python-coding** exercise

When part of this controller is 100% clear, just assume it is working correctly. And fill in the details as needed. Use that as base for your test-code.

- At least, eventually, the class and the code are consistent. So, future changes will not invalidate current (assumed) behaviour; without notice.
- During normal development, such details should be incorporated into the doc-string; possible after discussion and/or approval

- For the ‘homework-goal’ the exact working isn’t that relevant. Fill in details as needed; **focus on writing python-code!**

**class** `dpid.dPID (P, I, D, min_result=None, max_result=None)`

A simple discrete-PID controller, as an exercise.

This PID-controller can be initialised with constantes for P, I and D; which can’t be changed afterwards. Optional, a minimum and maximum output value can be given; both during initialisation, and later.

The controller has two **inputs**: `setpoint()` and `measured()`, and one **output**: `result()`. Those inputs can be set/updated independently. Similarly, the `result()` can be read at any-moment. As the controller will *remember* the timestamp a values changes (and knows when the result is read), it will always give the correct output. Thus, that output value does depend on the timestamp it is requested!

The `setpoint()` is considered as a step-function: between two changes, it will remain the last value. The `measured()` value however should be considered as a continuously linear-changing value. So, between two updates of this value, the dPID-controller will interpolate linearly.

When a `result()` is read during such a period; the PID-controller can’t predict the next-measured-value, however. Therefor, it will (for that single read) assume the measured-value is the same as last-time.

When a maximum and/or minimum value is set, the `result()` will be clipped to that value when needed. Without a min/max, the `result()` is unlimited.

---

**Hint:** As this class is part of an exercise; no implementation is given.

During the training one should update this file to implement the class **without** changing the interface.

All (numeric) input & output values are either integers or floats.

---

**setpoint** (`sp`)

Set the setpoint: a numeric value.

**measured** (`value`)

Give the controller an update on the actual *measured* (or simulated) process-value.

The controller will assume a linear progression between the last update and the current one

**result** ()

Return the actual result value

**set\_min\_max** (`min_result=None, max_result=None`)

Change the minimum and/or maximal result value. Used to clip the `result()`

## dPID: Exercise

**status** pre-alpha

## TDD-approach

### Test first

First, write some test-files with test-functions that verify the correct working of the `dpid.dPID` class as specified. Depending on your knowledge of PID-controllers these test may (functionally) vary. The primary goal here is not to program (including testing) a great PID-controller; but to practice your python-skills.

---

**Hint:** time-dependent

The `dpid.dPID` class is discrete; it only calculates the `dpid.dPID.result()` when requested. This implies the (return) value of `dpid.dPID.result()` will depend on when it is called (relative to the other methods).

- So, timing may be relevant in your (test)-code. One can use `time.sleep` to control that. Use a floating-point parameter for sub-second resolution. Typically, the function is accurate in the milli-seconds range.
- Compare using a small *MARGIN*, to allow derivations due e.g. timing. See the *examples*.

**See also:**

Convenient python functions

- <https://docs.python.org/3.5/library/time.html?highlight=sleep#time.sleep>
  - <https://docs.python.org/3.5/library/functions.html?highlight=abs#abs>
- 

## Instructions

- Use the `pytest` framework, to shorten test-code.
  - *Pytest introduction* is a summary with everything you need for this exercise.
  - You can use *Some (py)test examples* as *template* for your own files.
- Start by running `pytest` as shown.
  - Reproduce the shown output first, before adding your own files.
  - Remember; the output text will differ slightly; for paths, dates etc.
- Add a (one) new file: `test_<something>.py`.
  - Copy the start of the file (above the first function); update the copyright.
  - Write a single test-function:
    - \* Create a `dpid.dPID` instance with known, **simple** P, I and D settings.
    - \* Give it a `setpoint()` and `measured()` value.
    - \* Request a `result()`.
    - \* Assert the returned value is (almost) equal to the pre-computed number.
- Run `pytest` again.
  - It should run both the existing example, and the new test-file.
  - The test should fail! As the *empty* class always returns 0, that is easy.
  - Watch for *syntax* and other errors! All, but `AssertionError`, should be resolved in your code
- Repeat, either by adding a test-function to that file, or adding more test-files.

- When needed, you can add auxiliary functions; just don't use the *test*-phrase in its name.
- Or continue first with the implementation part. And add more test later (for other functions).
- Each test should be a little more complicated as the existing ones.
- Or better: start with the trivial once. Then the almost-trivial, etc.
  - \* Start testing a “*P-only*” PID-controller
  - \* Then an “*I-only*”, then a “*D-only*”. After which you test a simple combination
  - \* etc.

## Code second

When a part of the functionality is tested (or at least: there is test-code for), you can start implementing the *dPID* class. Keep is simple. The only objective is to **make one failing test pass**.

## And improve (refactor)

### dPID: The code

**status** RC-1.0

The code of the test-examples and the (empty) *dpid.dPID* class are shown here. They are exactly as the python-files; but for the highlighting.

### Some (py)test examples

```

1  # Copyright (C) 2017: ALbert Mietfus, SoftwareBeterMaken
2  # Part of my MESS project
3  # Dropjes licencie: Beloon me met dropjes naar nuttigheid
4
5  import pytest
6
7  from logging import getLogger
8  logger = getLogger(__name__)
9
10 from dpid import dPID
11
12 def test_P():
13     MARGIN = 0.5
14
15     c = dPID(1,0,0)
16
17     c.setpoint(10.0)
18     c.measured(10.0)
19     out = c.result()
20
21     assert (-1*MARGIN) < out < MARGIN, "result (%s) should be close to zero (MARGIN=
    ↳ %s)" % (out, MARGIN)
22
23 def test_clip():
24     c = dPID(1,2,3)

```

(continues on next page)

(continued from previous page)

```

25
26     c.set_min_max(min_result=10)
27     c.set_min_max(max_result=10)
28
29     for sp in range(-100,100,10):
30         c.setpoint(sp)
31         c.measured(0)
32
33         got = c.result()
34         assert got == 10, "Both min and max are clipped to 10; so result should be 10!
→. But it is: %s" % c.result()

```

### The class (empty)

```

1  # Copyright (C) 2017: ALbert Mietfus, SoftwareBeterMaken
2  # Part of my MESS project
3  # Dropjes licencie: Beloon me met dropjes naar nuttigheid
4
5
6  from logging import getLogger
7  logger = getLogger(__name__)
8
9  class dPID:
10     """A simple discrete-PID controller, as an exercise.
11
12     This PID-controller can be initialised with constantes for ``P``, ``I`` and ``D``;
13     which can't be changed afterwards. Optional, a minimum and maximum
14     output value can be given; both during initialisation, and later.
15
16     The controller has two inputs: :meth:`.setpoint` and
17     :meth:`.measured`, and one output: :meth:`.result`. Those inputs can
18     be set/updated independently. Similarly, the :meth:`.result` can be read
19     at any-moment. As the controller will remember the timestamp a values
20     changes (and knows when the result is read), it will always give the
21     correct output. Thus, that output value does depend on the timestamp it
22     is requested!
23
24     The :meth:`.setpoint` is considered as a step-function: between two
25     changes, it will remain the last value. The :meth:`.measured` value
26     however should be considered as a continuously linear-changing value. So,
27     between two updates of this value, the dPID-controller will interpolate
28     linearly.
29
30     When a :meth:`.result` is read during such a period; the PID-controller can't
31     predict the next-measured-value, however. Therefor, it will (for that
32     single read) assume the measured-value is the same as last-time.
33
34     When a maximum and/or minimum value is set, the :meth:`.result` will be
35     clipped to that value when needed. Without a min/max, the :meth:`.result` is
36     unlimited.
37
38     .. hint:: As this class is part of an exercise; no implementation is given.
39
40

```

(continues on next page)

(continued from previous page)

```

41     During the training one should update this file to implement the class
42     **without** changing the interface.
43
44     All (numeric) input & output values are either integers or floats.
45
46     """
47
48
49
50     def __init__(self, P,I,D, min_result=None, max_result=None): pass
51
52     def setpoint(self, sp):
53         """Set the setpoint: a numeric value."""
54
55     def measured(self, value):
56         """Give the controller an update on the actual *measured* (or simulated)
57         ↪ process-value.
58
59         The controller will assume a linear progression between the last update and
60         ↪ the current one
61         """
62
63     def result(self):
64         """Return the actual result value"""
65         return 0.0 # XXX
66
67     def set_min_max(self, min_result=None, max_result=None):
68         """Change the minimum and/or maximal result value. Used to clip the :meth:`.
69         ↪ result`"""

```

## Downloads

You can download these files directly from [bitbucket](https://bitbucket.org)

- [https://bitbucket.org/ALbert\\_Mietus/mess/raw/default/pyMESS/training/dPID/test\\_examples.py](https://bitbucket.org/ALbert_Mietus/mess/raw/default/pyMESS/training/dPID/test_examples.py)
- [https://bitbucket.org/ALbert\\_Mietus/mess/raw/default/pyMESS/training/dPID/dpid.py](https://bitbucket.org/ALbert_Mietus/mess/raw/default/pyMESS/training/dPID/dpid.py)

## Pytest introduction

**status** Beta

By using `pytest`, it becomes simple to run one, several or all test-functions. It has many advanced features, which are not needed for this exercise; but feel free to visit the website.

`Pytest` uses *autodiscovery* to find all tests. This makes all test-scripts a lot shorter (and easier to maintain), as the “*main-trick*” isn’t needed in all those files.

Without `pytest` all test-files should have a section like:

```

if __name__ == "__main__":
    test_P()
    test_clip()
    ...
    # list ALL your test here!

```

Effectively, `pytest` will automatically discover all test-functions; and execute them as-if that section is added, with all test-function listed (in file-order).

## Example

- Installation of `pytest` is trivial; use:

```
[Albert@pyMESS:] % pip install pytest
```

- Running all tests (in 1 directory) is trivial too:

```
[Albert@pyMESS:../dPID] % pytest
===== test session starts _
↪=====
platform darwin -- Python 3.4.1, pytest-3.0.4, py-1.4.31, pluggy-0.4.0
rootdir: /Users/albert/work/MESS,hg/pyMESS/training/dPID/dPID, inifile:
collected 2 items

test_examples.py .F

===== FAILURES _
↪=====
_____ test_clip _____
↪_____

    def test_clip():
        c = dPID(1,2,3)

        c.set_min_max(min_result=10)
        c.set_min_max(max_result=10)

        for sp in range(-100,100,10):
            c.setpoint(sp)
            c.measured(0)

            got = c.result()
>         assert got == 10, "Both min and max are clipped to 10; so result_
↪should be 10!. But it is: %s" % c.result()
E             AssertionError: Both min and max are clipped to 10; so result should_
↪be 10!. But it is: 0.0
E             assert 0.0 == 10

test_examples.py:33: AssertionError
===== 1 failed, 1 passed in 0.09_
↪seconds =====
```

---

**Note:** expect *AssertionErrors* (**ONLY**)

- As the class isn't implemented, one should expect *Asserts* during those (initial) runs.
  - Make sure you find *AssertionErrors* only; no syntax-errors etc! They denote mistakes in your code!
- 

- The used test-file (test\_examples.py) can be found [here](#)



## Conventions

To make this (*autodiscovery*) possible, one has to fulfill a few conventions:

1. All (python) files containing test-functions, should start with `test_`
  - Alternative: end with `_test.py`
  - Typically, *I* use the prefix for black-box and glass-box tests. And the suffix for white-box tests.
2. All test-functions should have a name starting with `test_`
  - No other function should *not* use that prefix!
3. Test-functions are called without arguments
  - We don't use/need `fixtures` here; which look like function-parameters. So, define all test-functions without parameters!

## OK or NOK: Assert on failure

Every test should result in a single-bit of information: OK nor Not-OK. Sometimes it may be useful to log (print) intermediate results; that can't replace the OK/NOK bit however.

With `pytest` this is easy: use the `assert statement`!

Typically a test ends with an `assert`. However, it's perfectly normal to have many `asserts` in one test-function; each one acts as a kind of sub-test. When a test succeeds hardly any output is generated; preventing cluttering of the test-reports.

When the first `assert-expression` results in `False` the test Fails. Then that `AssertionError` is shown with some context. Giving the programmer feedback on which test fails and why.

**Warning:** `Assert` is NOT a function

In python `assert` is a keyword with one or two expressions.

Don't use it as a function; which is a common (starters) mistake. Then, it is read as a single expression: a tuple with two elements. Which is always `True`. So the `assert` never fails!

Typically, the second expression is a string explaining what is expected. And so, documents that part of the test.

## See also:

### Links

- <https://pytest.readthedocs.io/>
- [https://en.wikipedia.org/wiki/PID\\_controller](https://en.wikipedia.org/wiki/PID_controller)
- [https://en.wikipedia.org/wiki/Test-driven\\_development](https://en.wikipedia.org/wiki/Test-driven_development)

## 1.5 BLOG indexes

### Posts

- *Pub/Sub* (2020/03)
- *ThreadPoolExecutor* (2020/03)
- *Requirements Traceability* (2020/02)
- *De Embedded Linux Expert bestaat niet* (2019/03)
- *dPID: A Python ‘homework’ exercise* (2017/10)

Also see the [Draft](#) ones (when available)

### 1.5.1 Major categories

#### Opinion

- *De Embedded Linux Expert bestaat niet* (2019/03)

#### Lecture

- *Requirements Traceability* (2020/02)

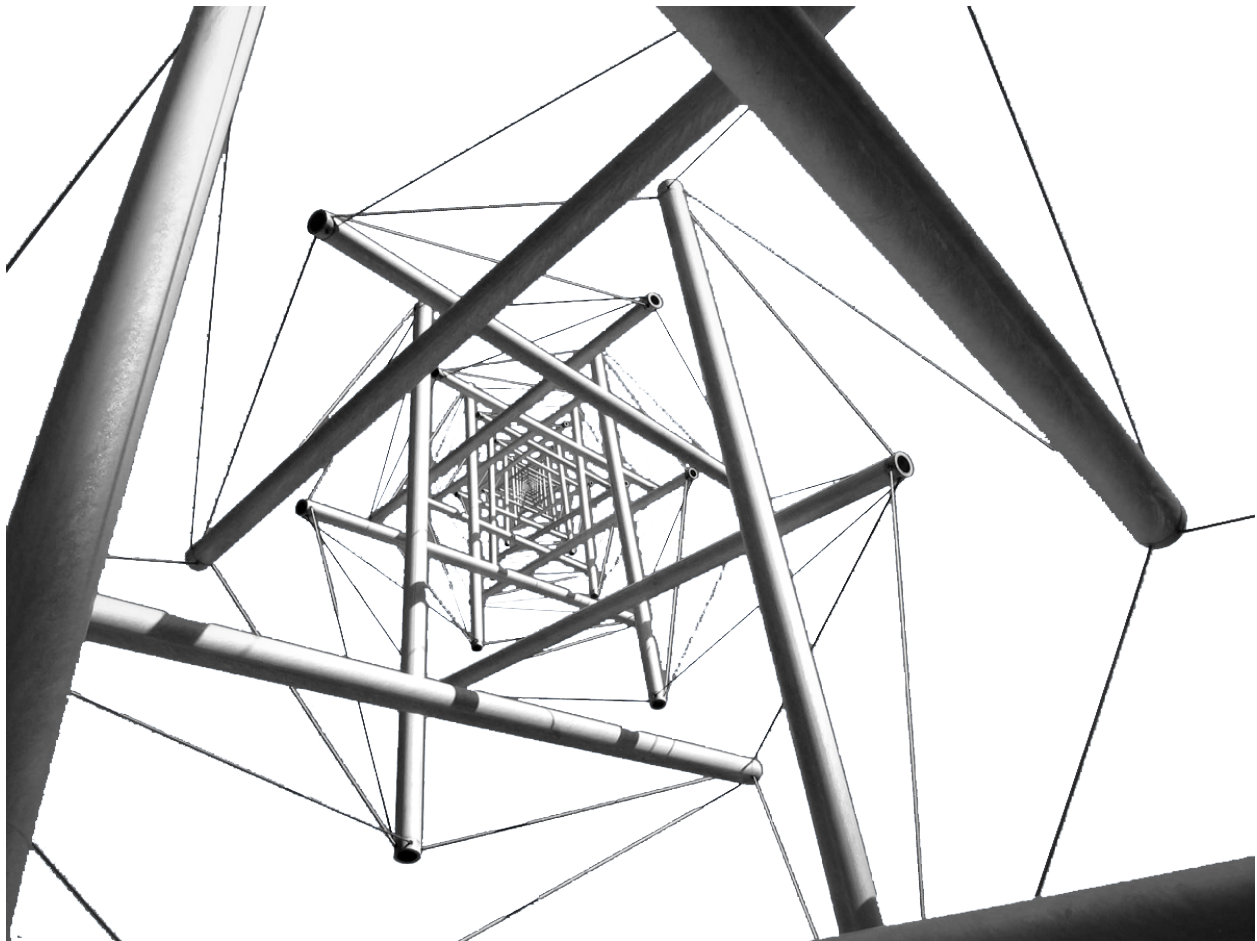
#### Practice

- *Pub/Sub* (2020/03)
- *ThreadPoolExecutor* (2020/03)
- *dPID: A Python ‘homework’ exercise* (2017/10)

### 1.5.2 More indexes

- *Categories*
- *Tags*
- *Authors*
- *Languages*
- *Locations*
- *(Yearly) Archives*
- *Drafts*

## TENSEGRITY, AS INSPIRATION



**Tensegrity** is a synthesis of the names 'tensional' and 'integrity'. It is based on "*teamwork*" of tension and compression forces. Although the image may look confusing, these structures are very simple. All you need are some poles, some cable, and good engineering. This results in a beautiful 'tensegrity-tower' where the poles almost float in the air; as shown [above](#)

It is also a well-known architectural principle for skyscrapers!

For me, it is also an inspiration for Software-Engineering: It should be based on teamwork: a synthesis of creative and verifying people. Together with a methodical way-of-working the amplify each other. Then, the sky becomes a limit, which is easy!



## INDEX

### C

`callback_function_type()` (in module `pubsub.AbstractType`), 43  
`callback_method_type()` (in module `pubsub.AbstractType`), 43

### D

`done()` (*FutureObject* method), 28  
`dPID` (class in `dpid`), 47

### F

`FutureObject` (built-in class), 28

### M

`measured()` (`dpid.dPID` method), 47  
Modern Engineering, 3

### P

`Publisher` (class in `pubsub.AbstractType`), 43

### R

`result()` (`dpid.dPID` method), 47  
`result()` (*FutureObject* method), 28

### S

`set_min_max()` (`dpid.dPID` method), 47  
`setpoint()` (`dpid.dPID` method), 47  
Sovereign Software, 3  
`submit()` (*TPE* method), 28  
`Subscriber` (class in `pubsub.AbstractType`), 43

### T

`Topic` (class in `pubsub`), 42  
*TPE* (built-in class), 28