Reflexis
for agile organisations

Reflexis Flow – Agile through Lua

Lua Workshop 2011 / Switzerland

Ashwin Hirschi

These slides accompanied my live demo of Reflexis Flow.
To give you an idea of what I demoed, we added a handful of extra slides as well as little annotations like these. Enjoy!

Agenda

Reflexis
for agile organisations

Agile web development with Lua

Topics

- Introduction
- Model-driven development
- Demonstrations
- More about Flow
- Powered by Lua
- Closing remarks

About us

Reflexis
for agile organisations

Who are we?

- a software company
- based in the Netherlands
- using Lua since 2002
- Workshop 2006: Reflexis Lite
- our focus: smart web applications

My Reflexis Lite talk in 2006 discussed our Lua-driven desktop runtime for Windows.

Roberto wanted me to point out that I also wrote the article “Traveling light, the Lua way”, that appeared in IEEE Software, September 2007. So, there ;-).

Our platform

Reflexis
for agile organisations

Initially, we created Flow to speed up the tedious bits of working on web applications, so we could spend more time on the enjoyable/worthwhile tasks.

Reflexis Flow

A model-driven platform for...
agile web application development.

Was created using Lua,
and can be extended using Lua.

Nothing too fancy here... But the "model-driven" part probably needs some explaining. So...

What's the problem?

- software should be easier to create
- software should be easier to change (later on)
- software should be less buggy
- software should be more secure (*especially* on the Web!)
- software should match requirements better

While far from complete, I guess the message is clear. If you don't recognise these points, you either are very lucky, or need to get out more... Possibly both.

Model-driven development

Reflexis
for agile organisations

A lack of productivity is often seen as the cause of issues with software. This naturally leads into the following...

Possible solutions?

- ask developers to stay late... (or type faster?)
- get more developers on-board
- outsource: find even more developers, elsewhere
- get smarter IDE's (to do the typing?)
- use a language closer to the problem domain

I reckon most of these options look familiar to many... The last option is the road less-travelled, and (therefore?) the one taken by the model-driven software folks :-).

Model-driven development

Reflexis
for agile organisations

What is it?

- building software by constructing models
- model = description of solution (or problem)…
- in a *suitable* domain language

Obviously, the trick lies in finding a suitable language. Unsurprisingly, this is also where various model-driven approaches start to diverge. As with more traditional programming languages, everyone has their own ideas of what makes most sense in a given context.

Demonstration

A little demonstration

Demo #1

- calculating Body Mass Index (BMI)
- purpose: showing Flow's basic model structure
- preparations: *none*

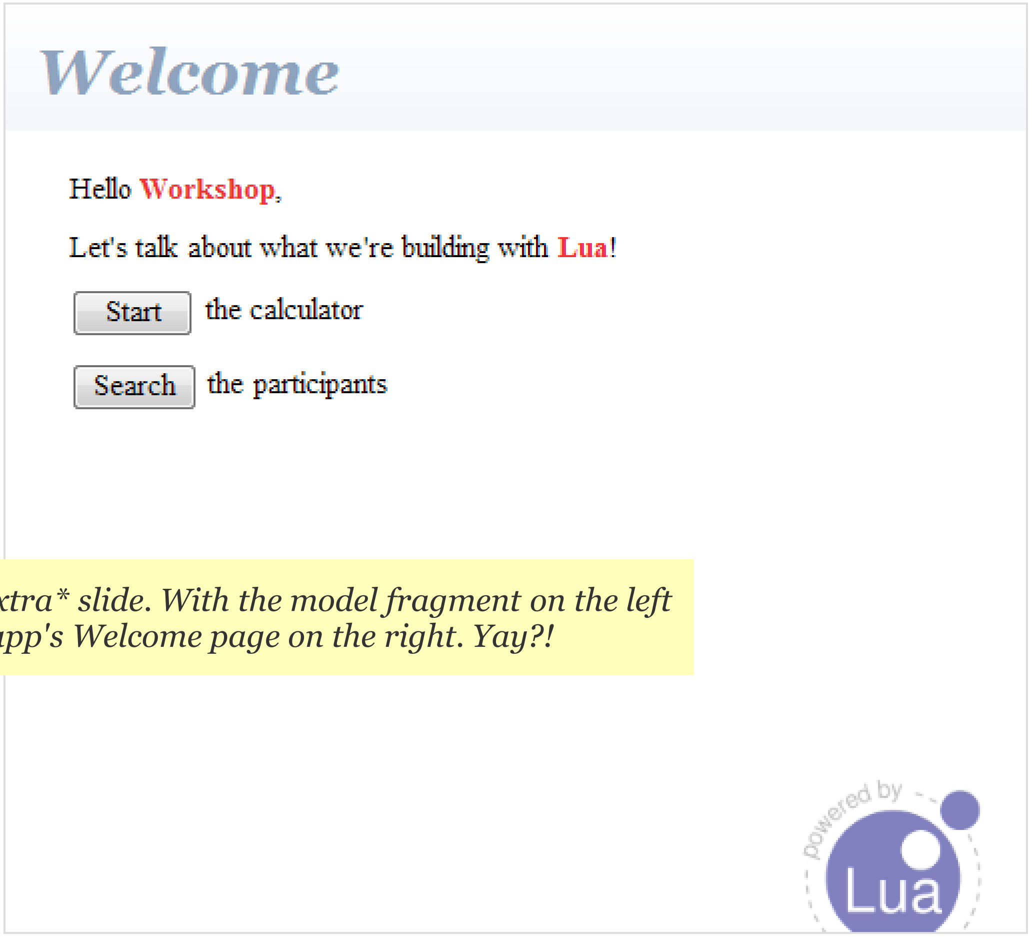
Discussing various model-driven approaches would take too long (and would probably bore people's socks off). Instead, I demonstrated how you can create a small web application with our Flow platform.

And, yes, a BMI calculator is silly. But it's also a good example of accepting some input, doing something and presenting the results.

Demonstration

Reflexis
for agile organisations

```
step "Begin" {
    kind = "user",
    header = "Welcome",
    page = [[
Hello <b>Workshop</b>,
<p>
Let's talk about what we're building with <b>Lua</b>!
<p>
$[start] the calculator
<p>
$[search] the participants
    ]],
    parts = {
        start = {
            kind = "ui/button",
            caption = "Start",
            target = "bmi",
        },
        search = {
            kind = "ui/button",
            caption = "Search",
            target = "search",
        },
    },
    gates = {
        bmi = "AskData",
        search = "ShowParticipants",
    },
}
```


Welcome

Hello **Workshop**,

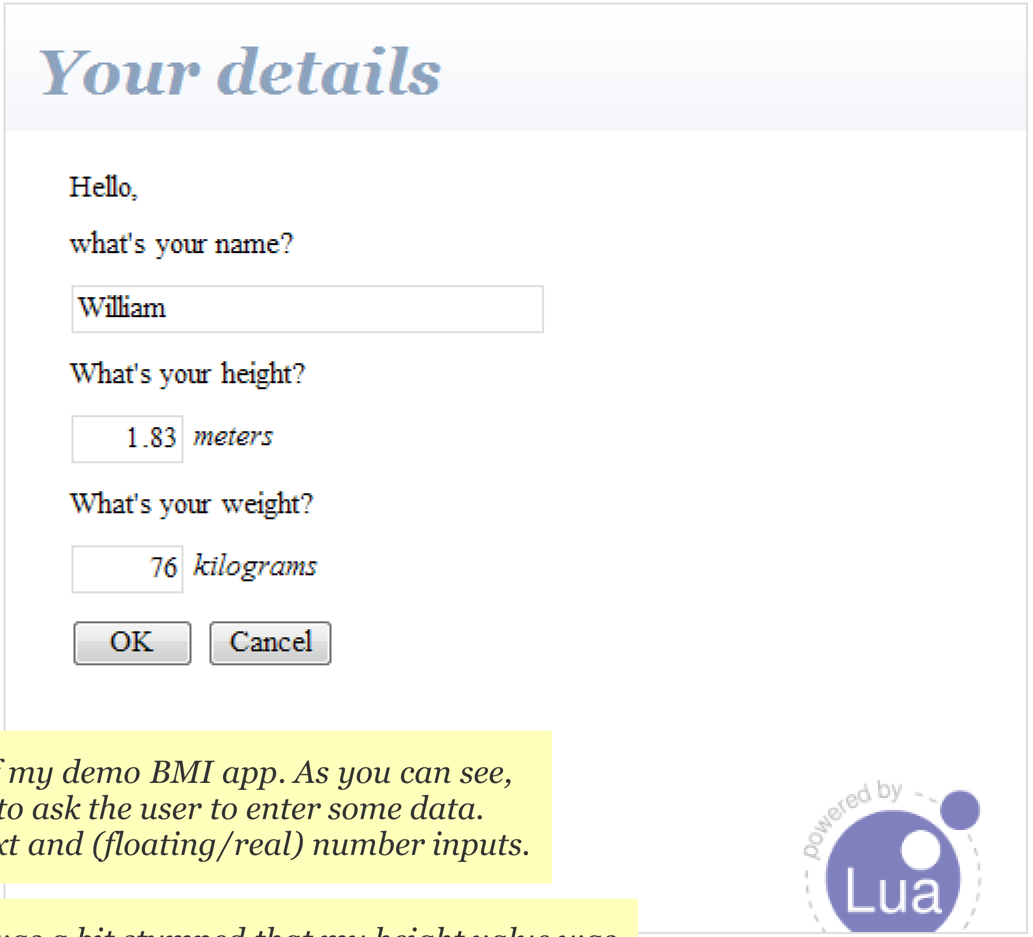
Let's talk about what we're building with **Lua**!

*Here's the first *extra* slide. With the model fragment on the left I created the webapp's Welcome page on the right. Yay?!*

Demonstration

Reflexis
for agile organisations

```
step "AskData" {
    kind = "user",
    header = "Your details",
    page = [[
Hello,
<p>what's your name?
<p>$[name]
<p>What's your height?
<p>$[height] <i>meters</i>
<p>What's your weight?
<p>$[weight] <i>kilograms</i>
<p>$[ok] $[cancel]
    ]],
    parts = {
        name = {
            kind = "ui/text",
            result = "person.name",
            explanation = "Please enter your name...",
        },
        height = {
            kind = "ui/number",
            result = "person.height",
            mode = "real",
        },
        weight = {
            kind = "ui/number",
            result = "person.weight",
            mode = "real",
        },
    },
    gates = {
        okay = "Calculate",
        cancel = "Begin",
    },
}
```



Your details

Hello,

what's your name?

William

What's your height?

1.83 *meters*

What's your weight?

76 *kilograms*

Here's the start of my demo BMI app. As you can see, it's a simple form to ask the user to enter some data. Note the mix of text and (floating/real) number inputs.

During the live demo I was a bit stumped that my height value was rejected. I had forgotten that by default the number part accepts only integer values. Adding the mode params fixed that.

Demonstration

```
step "Calculate" {
    kind = "server",
    page = [[ $[calc] ]],
    parts = {
        calc = {
            kind = "std/calculation",
            expression = "person.weight / person.height / person.height",
            result = "person.bmi",
        },
    },
    gates = {
        _default = "ShowBMI",
    },
}
```

This “server” step calculates the BMI and moves on.

The step does not produce an HTML page, but its page property still indicates which parts should run.

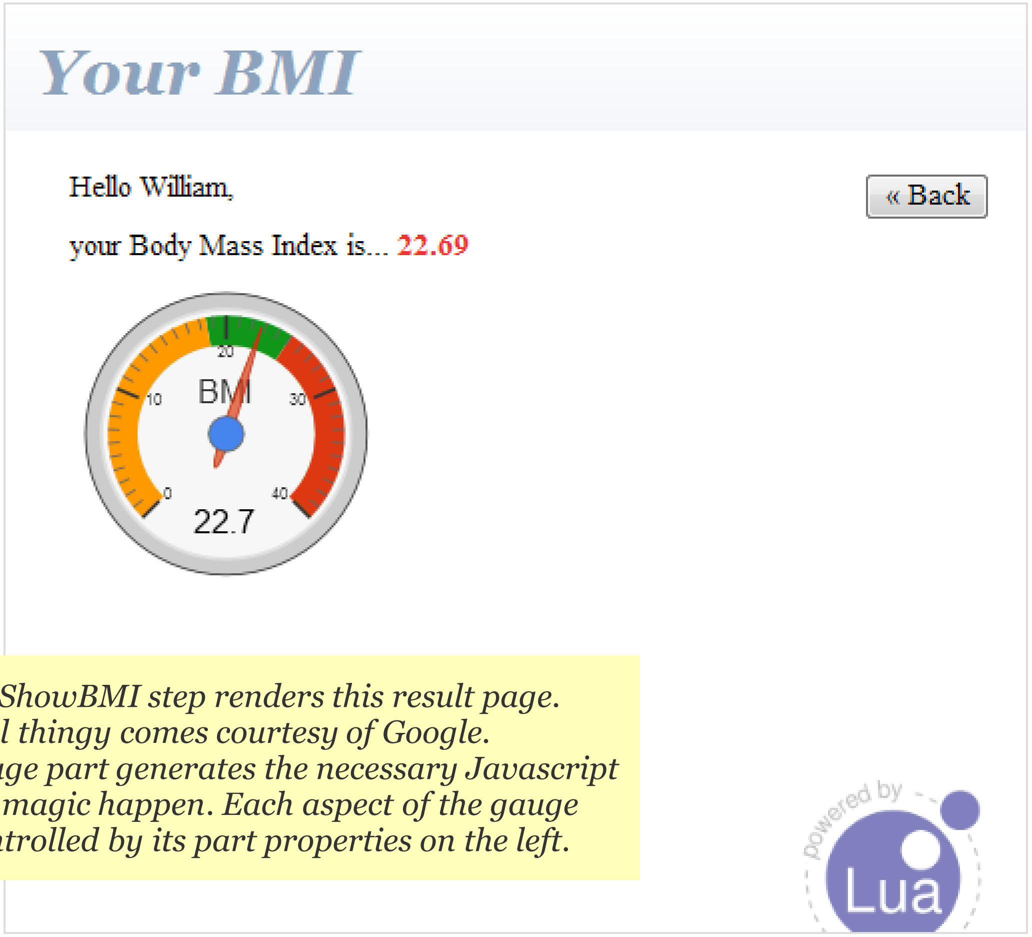
By now it should be clear that gates connect the steps of our flow. You can check this by examining the gates of the previous 2 steps.

Only one more step to add, to finish our first demo…

Demonstration

Reflexis
for agile organisations

```
step "ShowBMI" {
    kind = "user",
    header = "Your BMI",
    page =[[ $[back]
Hello $[name],
<p>your Body Mass Index is... <b>$[bmi]</b>
$[gauge]
    ]],
    parts = {
        name = {
            kind = "ui/display",
            expression = "person.name",
        },
        bmi = {
            kind = "ui/display",
            expression = "person.bmi",
            handler = "real:dec=2",
        },
        gauge = {
            kind = "ggl/gauge",
            path = "person.bmi",
            label = "BMI",
            max_value = 40,
            major_ticks = { 0, 10, 20, 30, 40 },
            minor_ticks = 10,
            yellow = { 0, 18.5 },
            green = { 18.5, 25 },
            red = { 25, 40 },
        },
    },
    gates = {
        back = "Begin",
    },
}
```


Your BMI

Hello William,

your Body Mass Index is... **22.69**

Finally, the ShowBMI step renders this result page.
The nice dial thingy comes courtesy of Google.
The ggl/gauge part generates the necessary Javascript to make the magic happen. Each aspect of the gauge above is controlled by its part properties on the left.

Demonstration

A little demonstration

Observations

- we created it quickly
- we tested it while creating it
- we did not write any code
- the flow definition/model is compact
- each submit refreshes the entire page

At the risk of belabouring the point, this first demo shows that with Flow you can create a working web application within minutes and without writing any code.

Demonstration

Another demonstration

Demo #2

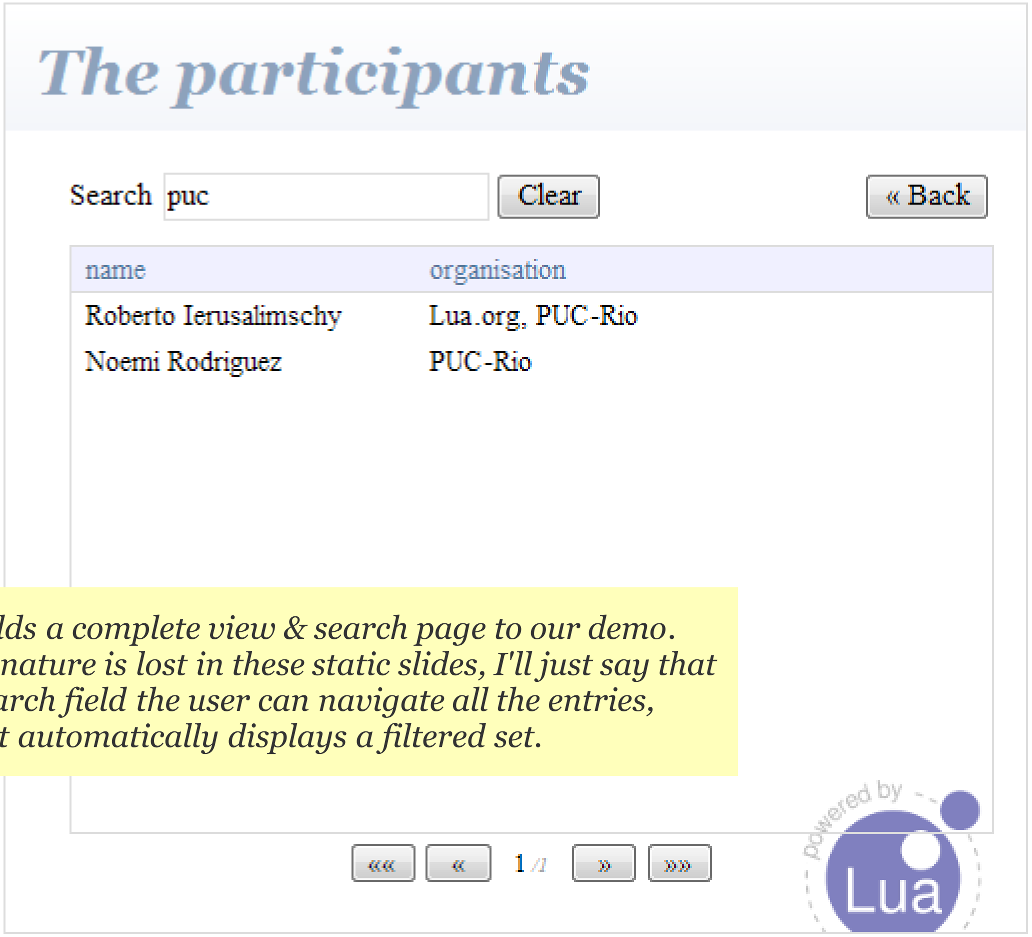
- searching the list of participants
- purpose: showing *asynchronous* operation
- preparations: a database table with the participants

> *Now, as browsers have moved on since 2004, in this second demo we'll add an extra page to show some AJAX interaction.*

Demonstration

Reflexis
for agile organisations

```
step "ShowParticipants" {
    kind = "user",
    header = "The participants",
    page = [[
$[back]
Search $[search]
$[overview]
    ]],
    parts = {
        overview = {
            kind = "ui/data_grid",
            query = "select firstname, lastname, organisation
from participant",
            columns = { "name", "organisation" },
            item = "<td>#1# #2#</td><td>#3#</td>",
            filter = "search",
            search = "firstname lastname organisation",
        },
        search = {
            kind = "ui/typeahead",
            result = "search",
            associates = { "overview" },
            reset = "Clear",
        },
    },
    gates = {
        back = "Begin",
    },
}
```



The participants

Search puc Clear « Back

name	organisation
Roberto Ierusalimschy	Lua.org, PUC-Rio
Noemi Rodriguez	PUC-Rio

This single step adds a complete view & search page to our demo. Since its dynamic nature is lost in these static slides, I'll just say that by clearing the search field the user can navigate all the entries, while entering text automatically displays a filtered set.

Time to switch back to the original/regular slides...

Basic model structure

- flows are applications
- flows contain steps
- steps are pages
- gates connect steps
- steps contain parts
- parts are components

As this talk was presented during the Lua Workshop, I created the demo flow using a text definition (and Lua syntax). However, many simply build their apps using our browser-based modeller (and don't get stumped by property defaults and such... ;-).

About parts

- are essentially components for webapps
- can act both synchronously & asynchronously
- come in many shapes & sizes
- created by programmers in Lua, but...
- everyone can use them (non-programmers included!)

I actually wrote the gauge part specifically for this presentation to add something visual to the otherwise down-to-earth demo.

It took about an hour to create the basic version from scratch (though afterwards I played around for another 2 hours to add the final touches).

Now it takes less than a minute to add a gauge to your own flows.

Advantages of models

- you can discuss them
- you can analyse them
- you can visualise or transform them
- you can still comprehend them if they grow
- they help you reduce errors
- they help you promote best practices

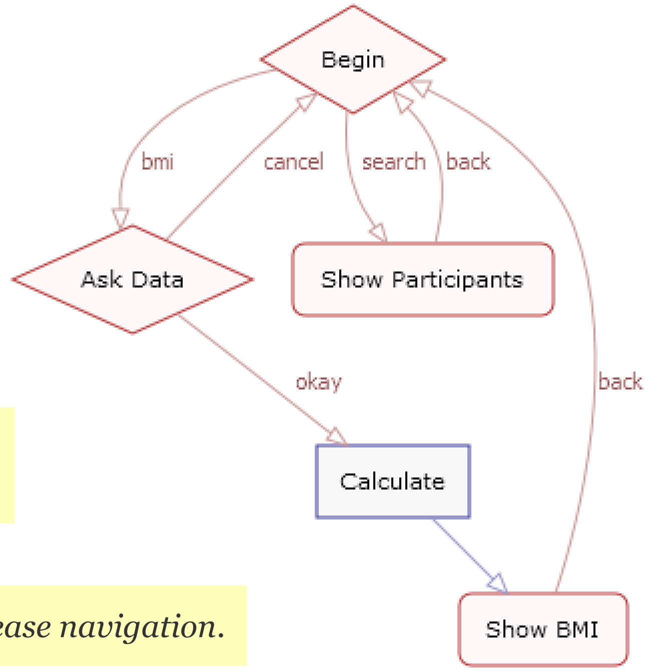
Note that flow definitions are basically simple data structures, making the models easy to read and process in various ways.

Reflexis Flow

The demo visualised

This flow graph was captured from our flow modeller.

Graphs like these function as “application maps”.



Our experience is that larger applications generally lead to “maps with many islands”.

Therefore, our modeller uses “local graphs” to ease navigation.

Finally, our flow modeller is a flow application itself, making it easy to adapt it to any special needs.

Powered by Lua

Reflexis
for agile organisations

Flow borrows much of its power from Lua

- Flow was developed entirely in Lua:
  - runtime section: approx. 2400 lines of code
  - modeltime section: approx. 1000 lines of code
  - tooling section: approx. 700 lines

I'm not counting part code and regular library stuff, obviously.

- Examples of Lua-based file formats:
  - flow definitions
  - session files
  - help files
  - configuration data

Powered by Lua

An important aspect of any model-driven system is how it deals with scenario's that it does not (yet) support. The handlers and snippets I discussed here are handy "fallback mechanisms".

Some cases

*A handler is a little piece of code that can tweak the behaviour of *existing* parts. The 'real' handler below is already shown on the ShowBMI step slide, btw.*

- handlers: adding custom validation or rendering
  - demo: add "real" handler to display BMI results
  - ui/text & ui/data_grid are handler-enabled parts
- std/snippet: a mechanism to easily fill functional gaps
  - demo: add "remark" snippet to the same results page
  - snippet code runs in either present or process phase

The snippet part enables you to insert some (Lua) code anywhere you want. The 'remark' snippet is shown in detail on the next slide...

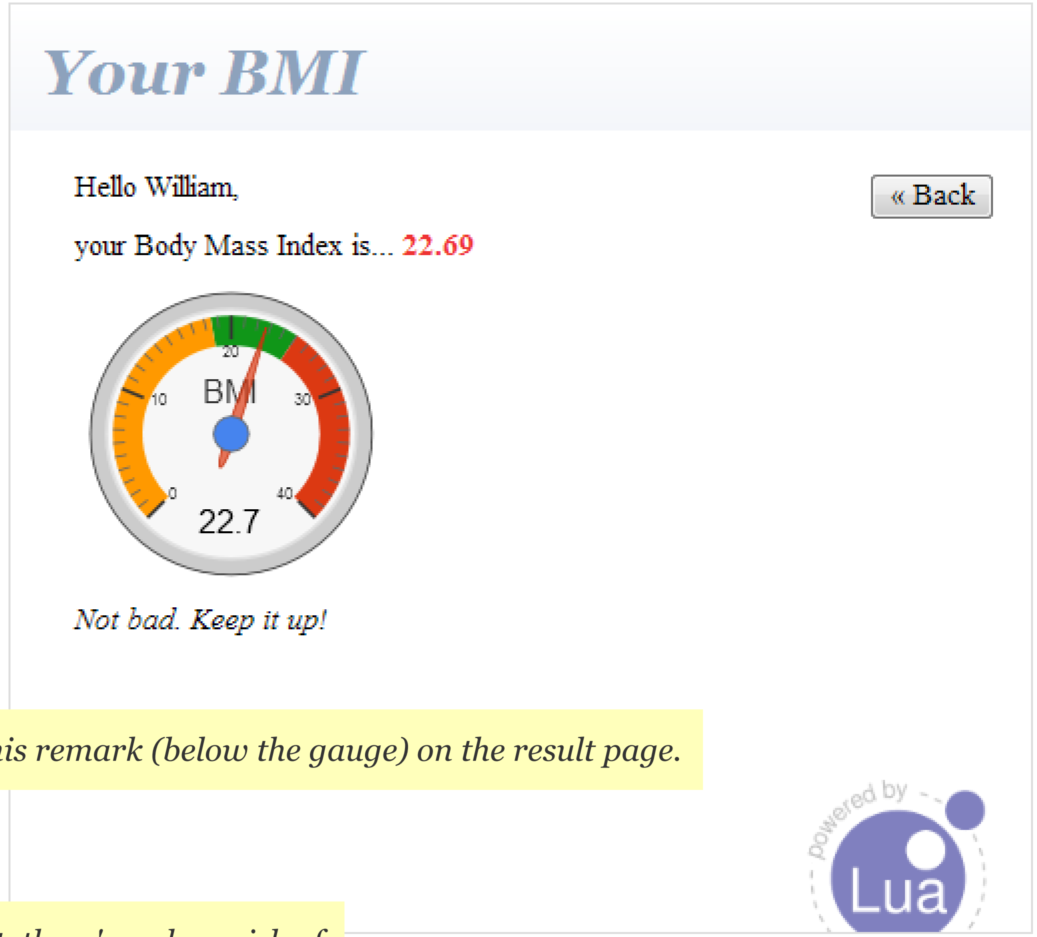
Demonstration

Reflexis
for agile organisations

Adding this (Lua code) snippet...

```
        remark = {
            kind = "std/snippet",
            code = [[
local bmi = tonumber(context.person.bmi)
if not bmi then return end

if bmi < 18.5 then
    print "Ummm... time for a snack?"
elseif bmi > 25 then
    print "No more cookies for you today!"
else
    print "Not bad. Keep it up!"
end
            ]],
            phase = "present",
        },
```



Your BMI

Hello William,

your Body Mass Index is... 22.69

Not bad. Keep it up!

...renders this remark (below the gauge) on the result page.

Though snippets are incredibly convenient, there's a clear risk of relying on them too much. We often urge people to create their own parts, since this raises overall quality & promotes re-use.

Powered by Lua

Time ran out during my original presentation, so I didn't get around to discussing these topics...

More cases

Having said that, the fact you can easily develop your own powerful parts is the best Flow feature by far!

- Extend Flow by simply writing new parts... in Lua
  - demo: show & tweak ui/button part code
  - parts are self-describing

This ensures the modeller recognises the parts you add.

- Lua to the rescue: dealing with SQLite
  - how to handle busy situations; wrapping the binding
  - SQL profiling; based on the same wrapper

Powered by Lua

Lua enables us to interpret models

No code generation

- Reflexis Flow executes everything on the fly
- all changes are reflected immediately
- no tool chain overhead whatsoever
- modeltime environment == runtime environment
- keeps everything (very) light-weight

*Code generation is another aspect that sets model-driven systems apart. Though I believe most systems *do* generate code (targetting e.g. the Java or .Net runtime), I'm very happy with our approach (for the reasons above).*

Closing

Reflexis
for agile organisations

The tip of the iceberg…

Not discussed today

- high-performance, extensible workflow engine
- translation mechanism to ease localisation
- integrated context-sensitive help
- easy deployment, using manifests & packages
- workers: for longer-running, asynchronous actions
- starters: session authorisation logic
- monitoring: RSS feeds to keep an eye on things
- all tooling in Lua (of course ;-)

So much to say, so little time… ;-)

Closing

Reflexis
for agile organisations

Bridging paradigms

Conclusion

- Reflexis Flow brings 3 worlds together:
  - straight-forward procedural programming
  - convenience of component-based assemblage
  - model-driven systems for higher-level designs

- When creating software, always consider alternatives to producing yet more code (*even* if it's in Lua!).

In a nutshell: writing more & more code is not always the best way to move ahead.
Lua can help you fuse techniques in creative ways and get more out of less.
It helped us create Reflexis Flow. And now I can't imagine ever having to do without.

Closing

Questions?

Contact us

- Arno Kusters (CEO)
- Ashwin Hirschi (CTO)
- still in Switzerland till Saturday (noon, probably)
- email: **questions** @t **reflexis.com**

A fortnight down the road, it's safe to say we're not in Switzerland anymore...
But the email address is still valid, and your questions are just as welcome.

Closing

Thank you, Roberto,

and Luiz & Waldemar,

for giving us the gift of Lua!

*The slide really says it all... Then again, some things are worth repeating. It's one thing to be smart. But it's quite something else to be smart *and* dedicated. I believe it's this combination that has made Lua into the Great Piece Of Software it is today. But, yeah, the slide says it all, really. So, THANK YOU!*

The End. I hope you enjoyed the slides. -- Ashwin.