Programming Languages and the Power Grid

Sebastian Egner, Head of Application Development,
Entelios AG, Berlin.

GOTO Aarhus, 30. September – 2. October 2013

Programming Languages and the Power Grid: Outline

1. **The Power Grid**
   - Design of a national power grid
   - Why and how to balance the grid
   - Two things to keep in mind on national scale

2. **Case Study**
   - Entelios AG
   - The right language for the job
   - Technology roadmap
   - Experiences

3. **Unfair Generalizations**
   - Two notable pitfalls of OO designs in practice
   - The "2-out-of-3" rule of dealing with project risk

4. **Programming Languages and the Power Grid**
   - Chains of availability

Programming Languages and the Power Grid: Outline

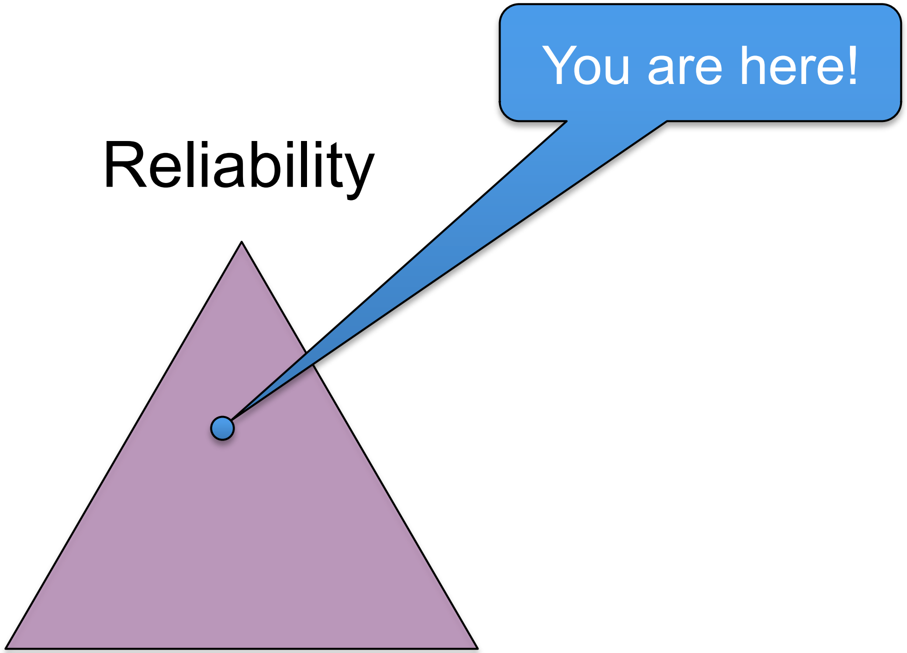
1. **The Power Grid**
   - Design of a national power grid
   - Why and how to balance the grid
   - Two things to keep in mind on national scale
2. **Case Study**
   - Entelios AG
   - The right language for the job
   - Technology roadmap
   - Experiences
3. **Unfair Generalizations**
   - Two notable pitfalls of OO designs in practice
   - The “2-out-of-3” rule of dealing with project risk
4. **Programming Languages and the Power Grid**
   - Chains of availability

Caution!
Font size will vary!

The Power Grid

“Design is not about
the actual choices you make.
It is about the alternatives
you have considered.”

Designing a Power Grid: *Where do you want to be?*

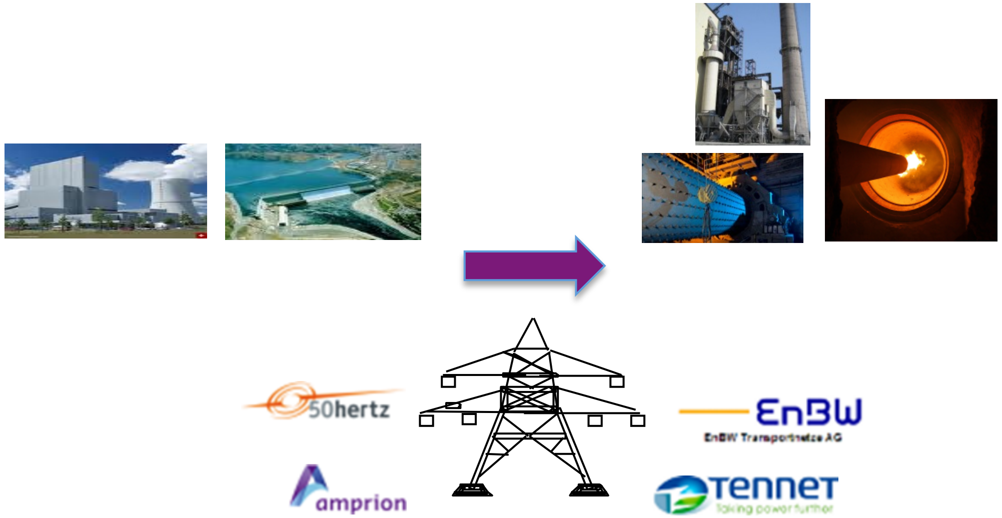


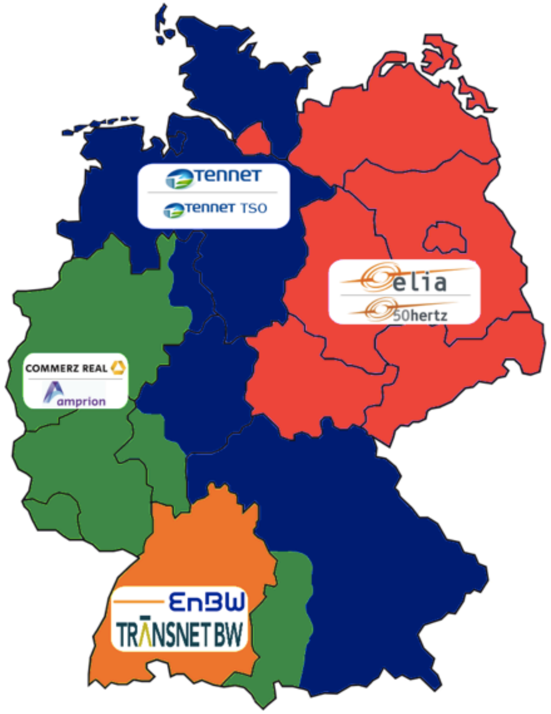
Reliability

Costs

Other:
- Environment
- Sustainability
- Major Risks

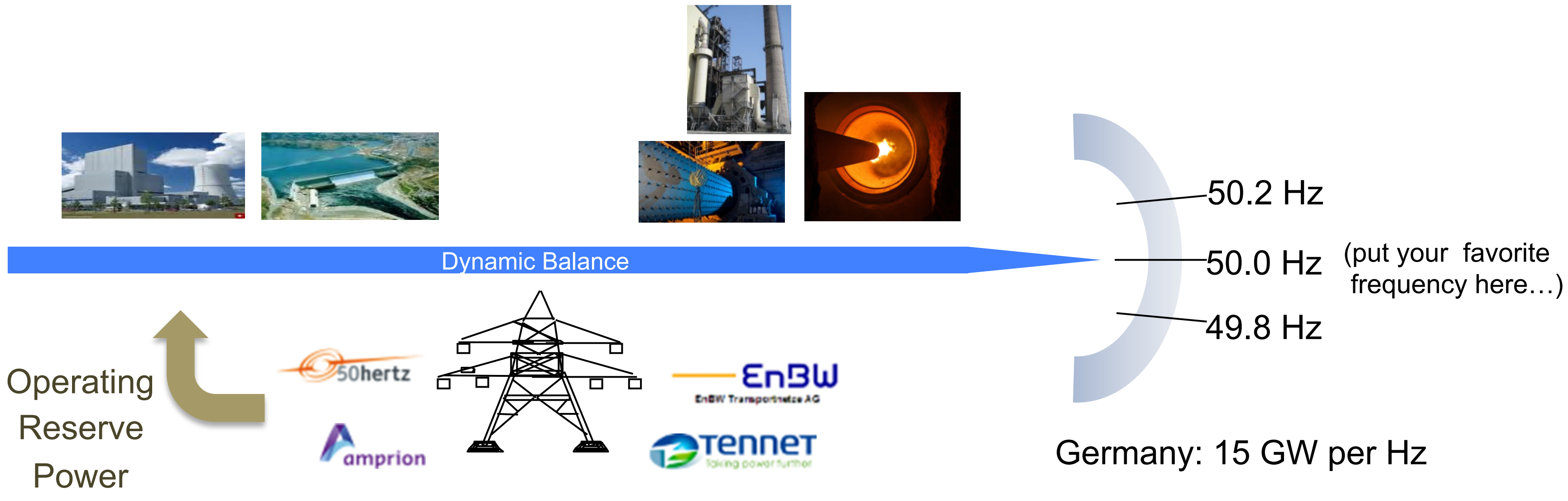
Balancing the *Power Grid*





Germany: 4 TSOs

Balancing the *Power Grid*



Dynamic Balance

50.2 Hz

50.0 Hz (put your favorite frequency here…)

49.8 Hz

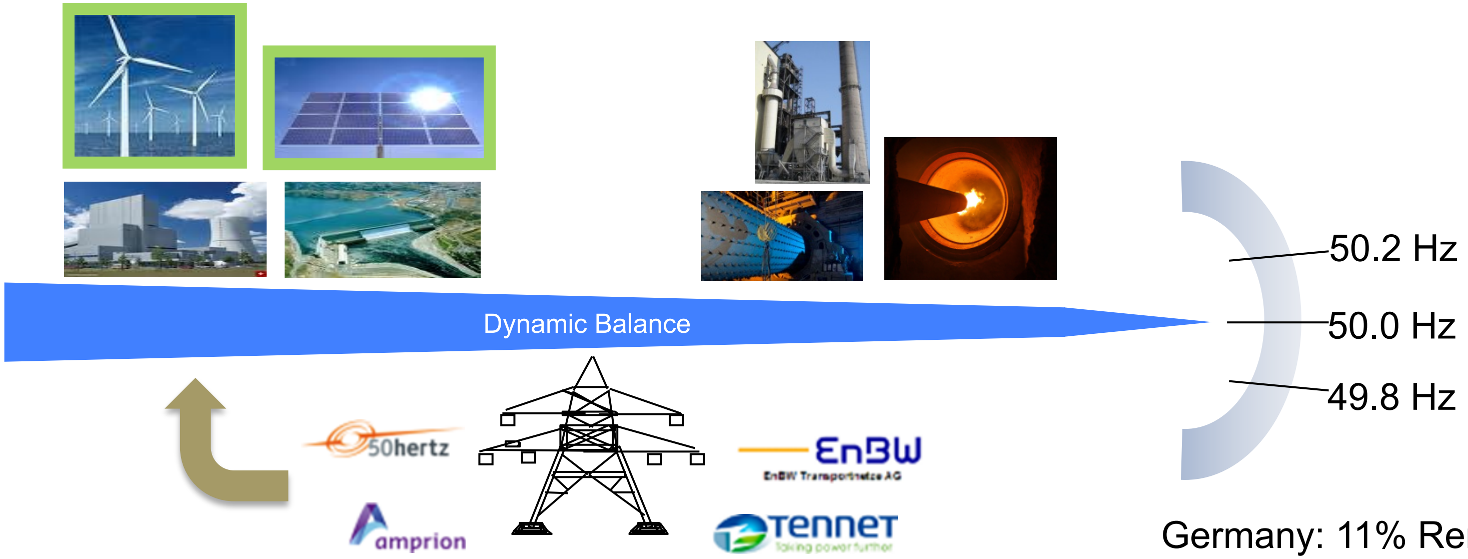
Operating Reserve Power

Germany: 15 GW per Hz

Industry Principle: **Generation** follows **Consumption**

Three level controller for reserve power (simplified):
- Frequency reserve (PRL), 20..200 mHz
- Secondary reserve (SRL), > 200 mHz, automatic
- Replacement reserve (MRL), > 15 min, manual

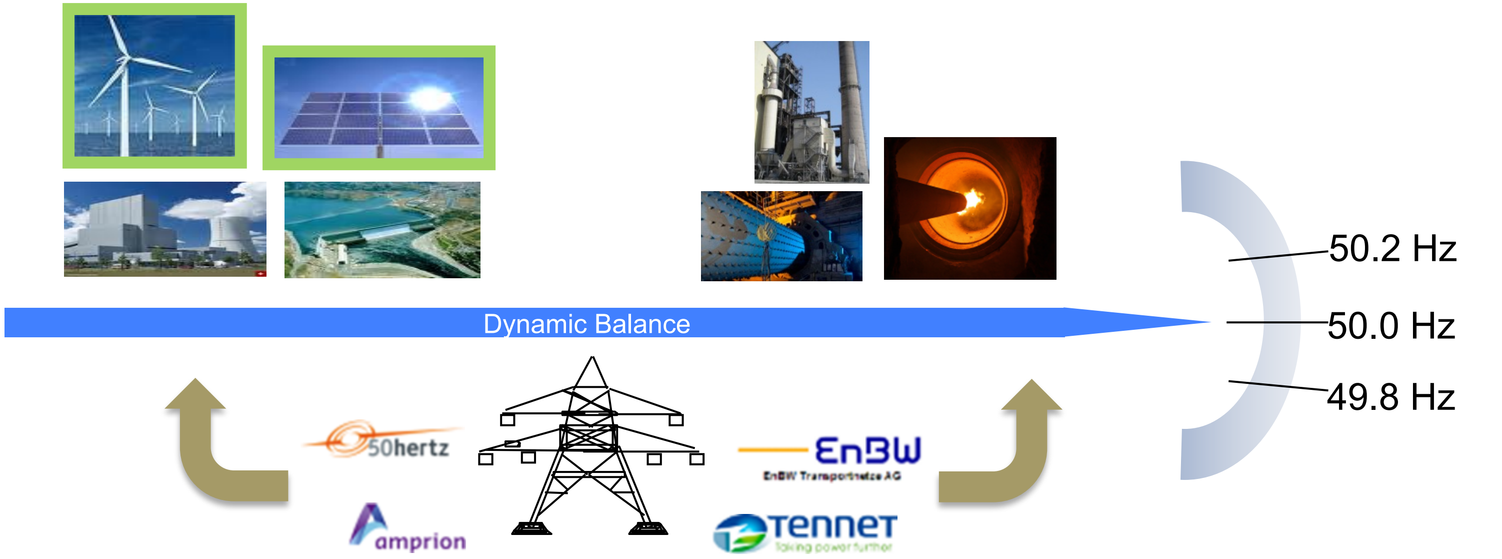
Balancing the *Power Grid*



Germany: 11% Renewables.
20 GW Peak (of 80 GW total)
on 18. August 2012.

Generation follows **Consumption,**
except for **Wind and Solar.**

Balancing the *Power Grid*



Generation follows **Consumption,**

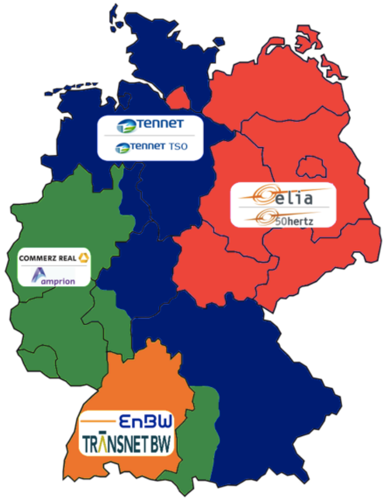
except for **Wind and Solar,**

and **Demand-Response** Management.

Demand-Response

- USA: Mature, IPO of EnerNOC, Inc., in 2007
- Load management *within* large consumers common, e.g. Xstrata Zink GmbH
- Extremely complex body of national regulations
- Europe: Early VC-funded companies (Entelios AG)

The Power of the Power Grid: *Mind the Order of Magnitude!*



100 mW

personal,
mobile phone

100 W

residential,
refridgerator

100 kW

industrial,
climate control

100 MW

industrial,
arc furnace

100 GW

national,
power grid
(e.g. Germany)

Entelios AG

The Batteries of the Power Grid: *Sometimes Not What You Expect*

You say: derinding buffer of a paper mill (Stora Enso, Eilenburg, Saxony), …

… I say: battery with 200 MWh capacity.

Case Study

Entelios AG

- Founded in 2010 by Oliver Stahl, Stephan Lindner and Thomas (Tom) Schulz
- VC-Funded (Series A completed in 2011 with a Dutch lead investor)
- Based in Germany (Munich, Berlin), employee range 20-50 + network of partners
- Runs its own Network Operations Center (NOC), with its own Balancing Area.
- Prequalified for providing Operating Reserve to German TSOs.

Services

Production of electrical energy by intelligent management of industrial consumers.

Exploiting dormant load flexibility, in particular in-production buffers.

Software-as-a-Service for Demand Response “(Virtual) Batteries Included”.

Providing a Commercially Viable Demand-Response Service

1. **Knowing the rules of the game:**

   Law, body of other regulations and actual practice.

2. **The actual business model:**

   “We sell A to B, who buy it because of C.”

   Exercise: Find A, B and C. (Note: Answers are graded in EUR +/−.)

3. **Finding industrial participants:**

   Why do they join? (Suppliers, found by sales process.)

4. **Technology:**

   Effective, reliable, usable, … and ever changing.

Providing a Commercially Viable Demand-Response Service

1. **Knowing the rules of the game:**

   Law, body of other regulations and actual practice.

2. **The actual business model:**

   “We sell A to B, who buy it because of C.”

   Exercise: Find A, B and C. (Note: Answers are graded in EUR +/−.)

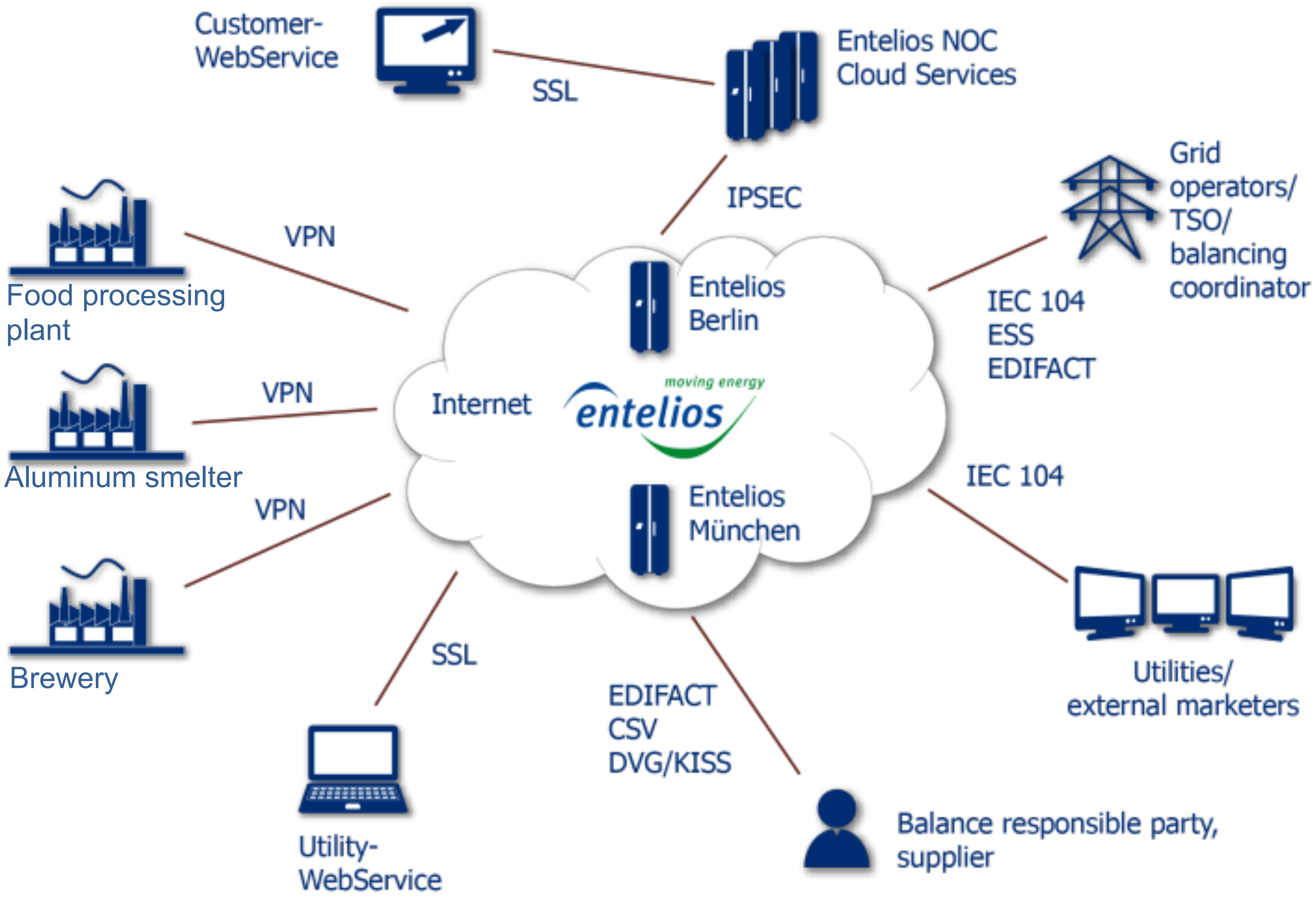
3. **Finding industrial participants:**

   Why do they join? (Suppliers, found by sales process.)

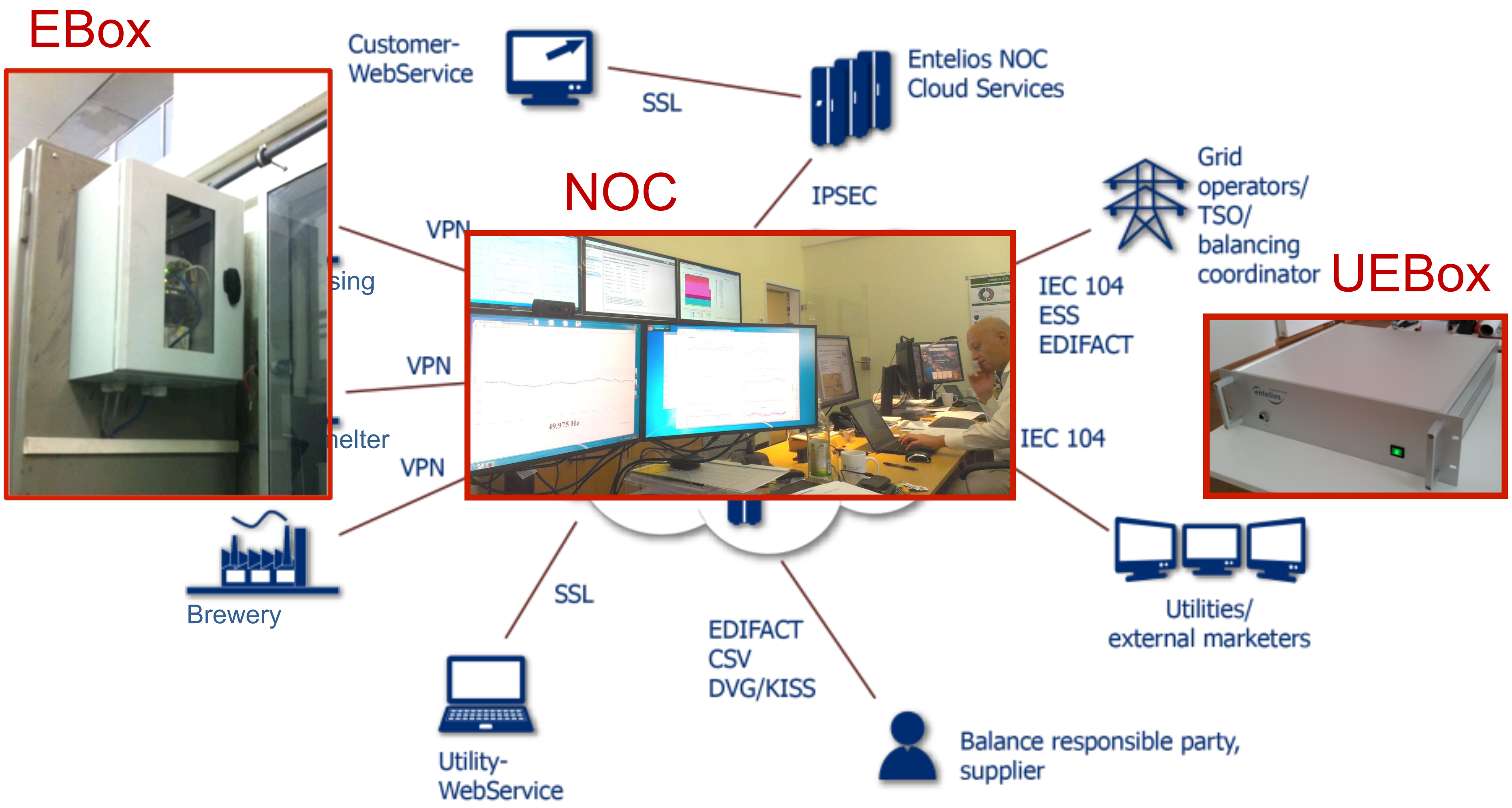
4. **Technology:**

   Effective, reliable, usable, … and ever changing.

Entelios AG in Context



Entelios AG in Context



The right language for the job… *So what is the job?*

Key Functionality

- Back-office system: 24/7, soft-realtime signal **acquisition / control** signals from / to industrial participants and grid operators. Sample rate: 2/min – 20/min
- Front-office system: Soft-**realtime GUI** for interactive planning and execution of curtailment events (load reduction) under time constraints. Task rate: 0 – 1/min
- Remote connection (**M2M**) to industrial participants via Internet, UMTS, GSM
- **Fieldbus-Interface** to the PLCs of the SCADA system of the industrial participants
- Interface to the operations centers of the grid operators (**IEC-104**, MOLS, …)
- Unsupervised Recovery from transient failure: UPS, **auto restart** at various levels

Additional Functionality (and there is a lot more…)

- Monitoring GUI, background screens
- Archiving of essentially all communications with external parties
- Export of time series data for periodic and *ad-hoc* analysis
- Periodic transfer of data to Energy Data Mgt. / Workflow / Trading Systems
- Various reports to participants and TSOs (for prequalification and quality control)

The right language for the job… *Ways to do a job*

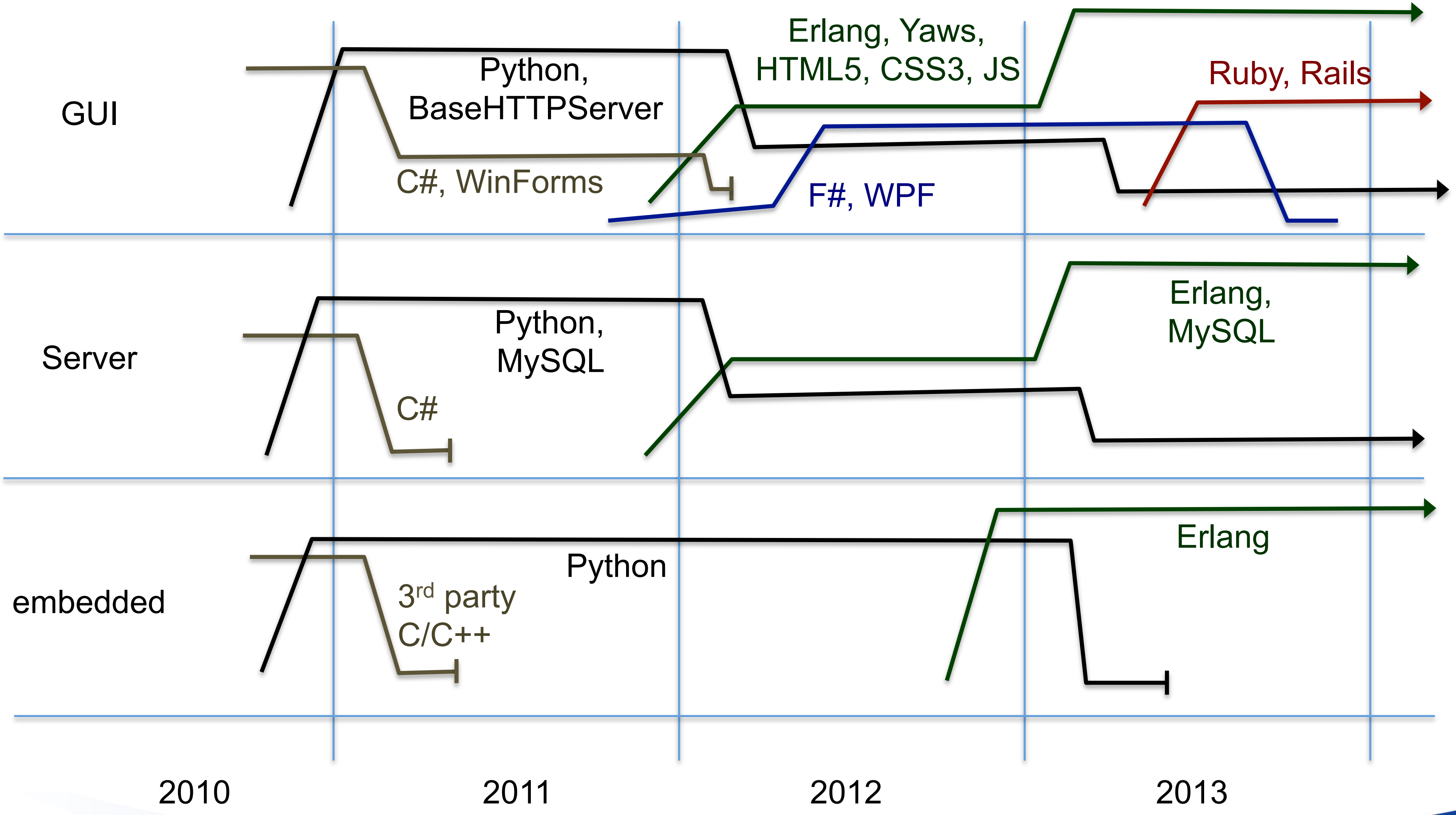
Snippets of how we do things:

- Cross-platform development from Day 1: Win 2008 Srv, Win 7 {32,64}-bit, MacOS X, {Deb,Ubu,SuSE}-Linux, embedded Linux.
- For new hires: “You can BYO anything you know how to use, or you get a Windows Notebook from us. Your choice.” So far: 100% Windows Notebooks, two of them actually used to work in Windows.
- Productivity = Hours * Effectiveness. (The second factor is the important one.)

Some principles:

- A successful system allows the user to do what she wants.
- Each tool is suitable for some task, but for other tasks there might be better tools.
- Choose which tools *not* to use. (Features bundled with your favourite toolkit…)
- The hardest task of software engineerign: getting rid of something.

Bits of Our Technology Roadmap (on the Rearview Mirror)



Green Field: *Initial Pragmatic Choices*

Embedded System and Server-Side Core:

- 1st choice of embedded platform turned out to be unlucky. (Their 3rd level support couldn't / wouldn't fix their own product…) → Supplier eventually dropped.
- 2nd choice was a lucky one. Devices optionally with an embedded Linux, incl. a Python 2.6 → **Embedded Python!** (Performance rel. to C not an issue for us.)
- Natural choice: Use **Python server-side**, too! → 99% overlap of embedded and server-side code, it's just "`--embedded`" to disable database access etc.
- Considerable part written in functional style, but of course not replacing `for` by home grown "`foreach`" calling a `lambda`.

Client-side GUI:

- Initial boundary condition: "Must run in .NET on Windows."
- Original concept required high amount of GUI interaction. → Rich client
- Choice of GUI toolkit (2010): WinForms (mature, aged L&F) vs. WPF (modern L&F)
- → **F# with WPF**, using Functional Reactive Programming for time series.

Requirements have Changed: *Adapting the Early Choices*

Redesign Server-Side Core in 2012:

- Increased scalability requirements along various dimensions: sample rate, redundancy, customers, industrial participants
- (Thread-)Concurrency in Python: It can be made to work, but that is tiring…
- Severely short on system tests. (Reasonable coverage in unit tests.)
- → **Erlang/OTP**: for concurrency and testability (and excellent previous experience)
- → Python stays for some functions (ad-hoc data analysis, forecasting, …)

Redesign Client-side GUI:

- Requirements have changed considerably:
  - Much less interaction required than original envisioned.
  - Also used for non-interactive monitoring.
- Only component to have repeatedly relapsed below roll-out Q-level:
  - Interaction performance (largely due to WPF's approach to widgets)
  - Memory leaks (widget resources, async + lazy + side-effects)
- → **Web-GUI in Erlang**, less interactive signal plots. Phasing-out F# / WPF.

And Now Focus has Changed, too: *It's Not Early Days Anymore*

Redesigned Embedded Platform in 2013:

- Motivation: Multi-controller access and redundancy, faster data acquisition, automatic catching-up after network outage.
- → **Erlang/OTP on the embedded platform**)
- → Porting effort for platform, submitting a few patches upstream.

Unifying Look-and-Feel of the GUI in 2013:

- Focus changed from functionality (=> each compontent brings its own UI style) to an integrated look-and-feel with brand recognition.
- Important for marketing the software as a "solution".
- Closer integration with the business-side software (workflow, ERP, accounting etc.)

Random Bits of Experience…

…with Python:

- Has served us well, in particular on the embedded platform.
- No “unsolvable” issues, rich library, program straight-forward to extend.
- Relatively large step from prototype (script) to production code.
- Major thread-headache for realtime system, especially controlled shutdown and restart.

…with F# / WPF:

- Has worked for us, and we do use it in production. Good fit with original concept.
- The only part of the software the relapsed several times below roll-out quality level.
- In practice, we find it hard to modify or correct other people’s F# / WPF code.
- One F# issue reported back to Microsoft (initializer). (Turned out version 2.0.0.0 ≠ 2.0.0.0.)

…with Erlang/OTP:

- Everybody working on the project and beyond is happy with it. *(Read this again, if you want.)*
- Relatively slow project start: building, testing, establish common coding style, etc.
- Three issues reported back to Erlang/OTP team (ARM middle endian; dialyzer bug; _/utf8).

Random Bits of Experience…

…with MySQL:

- The only technology that was with us from the start, and still is today.
- Nearly exclusively used in “archive mode”.
- SQL: data must be rectangular. Lucky for us, our (time series) data is!
- Had to hack our own MySQL client in Erlang: *not* easy, one size *does not* fit all
- Insulated by about 30 min. worth of buffering from the soft real-time system.
- Amazing issues (v5.1): float in – another float out; character encoding broken.
- Nothing that we couldn’t work around.

…with HTML5 / CSS3 / JS:

- Surprise: Browser compatibility less of an issue than expected.
- We keep it even simpler: CSS is hard to test, JS is browser-side (for us)
- Wrote our own CSS parser (in Erlang) for detecting dead (unreachable) CSS code.

Observations on Erlang/OTP

- Relatively small step from prototype and production code.
- Easy to understand other people’s code. (The questions “How do I define a gen_server in monadic style?” and “When do they get around to object-oriented Erlang?” disappear quickly.)
- Often you refactor in Erlang and your code becomes 2x smaller, and that alone feels like you did something right. (Java: You refactor, it is clearly the right thing to do, and you constantly ask yourself is the result worth all the cruft.)
- Production code often stays stable for years. (This means modularization is effective.)
- Make well-tested building blocks can be recombined into different systems.
- Final production code much smaller (say 5x c.t. Java), once it is finished. Not necessarily faster to develop, though.
- Difficult: Shutting down processes properly without undue error propagation. (Eventually, I wrote a small combinatorial program to generate and study all possible ways a gen_server example can exit, and what happens then.)
- Common_test: Very useful, but noisy…
- QuickCheck: Complements hand-crafted tests perfectly. Hand crafted: rifle. QC: shot gun.
- Great: interactively debugging a live system.
- Great: resilience (Example: system was limping on for hours, did not loose any data)
- Great: hot code-update (we do the easy cases, only)

What We Have Added to Erlang/OTP

Our own build mechanism “ebt” (= Entelios/Erlang Build Tool), including:

- build the system (on Linux, Windows and MacOSX)
- build the embedded system (on ARM-based Linux, on server as cross-compile)
- run the tests (Common_test). Variant: run only the fastest tests until 5 min. are up
- run the tests with cover analysis (Cover)
- pragma to silence Dialyzer (static code analysis): … `% dialyzer: -warn_failing_call`
- internationalization (“i18n”): crawls the code for certain function calls, then runs *GNU gettext*
- check basic coding standards (no tabs etc.): crawl .erl, .hrl, .yaws, .css, .js, etc.
- compile Mercurial version into the code: *every* build knows its version!
- run Leex/Yecc (parser generators)

General libraries within our Erlang code base:

- strings (UTF-8 as binary), timestamps (ms precision), option lists (= uptight proplists)
- Tracing (application-defined, not by structure of process tree)
- Running Gnuplot, GLPK and Python (on Linux, Windows and MacOSX)
- Password file access
- validation of HTML5, CSS3

What We Are NOT Using from Erlang/OTP

Meta-programming and ways to obscure function calls at the call site:

- parse_transformations: consider using Erlang, repeat
- (define own) behaviour: we did and we rolled it back for reducing code redundancy
- `-import`: when fingers get sore, `-define` an abbreviation

“Let it crash!” and error discipline in general:

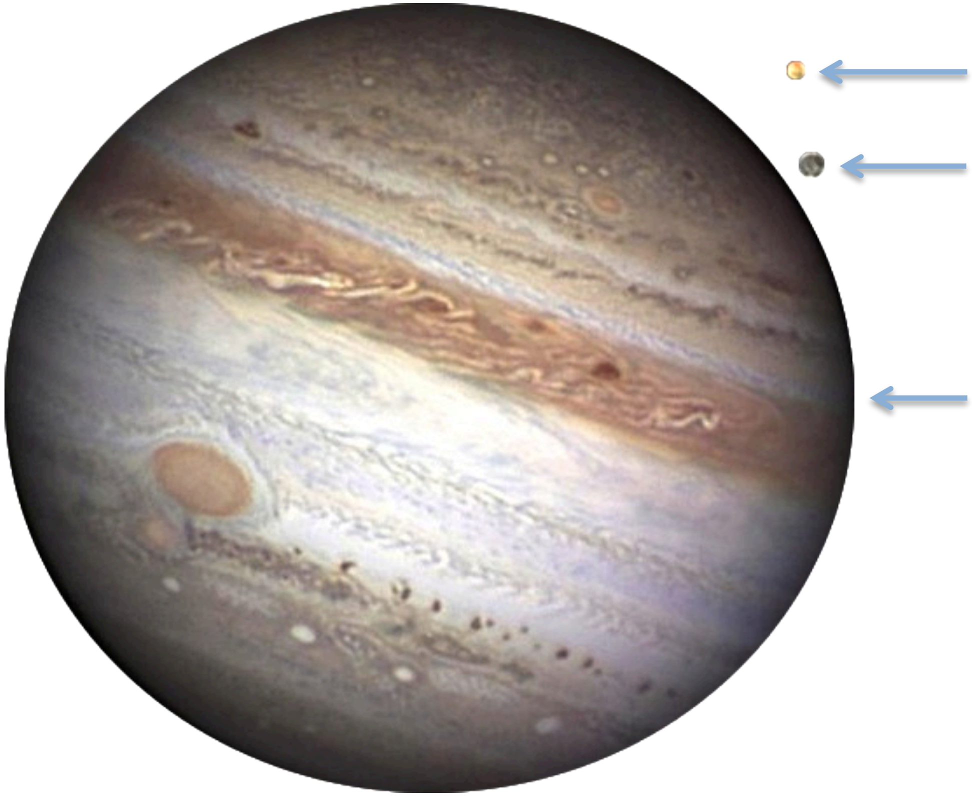
- In a test: yes
- In the webserver: no.
- In a library: probably not. (It might end up part of the webserver, and it usually does.)
- We like `{ok,Value} | {error,Reason::atom(),Details::proplist()}` a lot.
- There is a difference between a programming error (crash is good) and bad input.
- `check_MyType(Arg)` functions returning `ok|{error,_,_}` do an in-depth check of a data structure (incl. dynamic invariants); used as assertion (`ok = check(…)`) or in a `case`.

Type annotations, documentation and helping with static type analysis:

- `-compile(export_all)`: just `-export`
- `-spec`: nice feature, we avoid it. Found in places where proper documentation was due.

Unfair Generalizations

When OO in the wild fails (1)… “Jupiter Design”

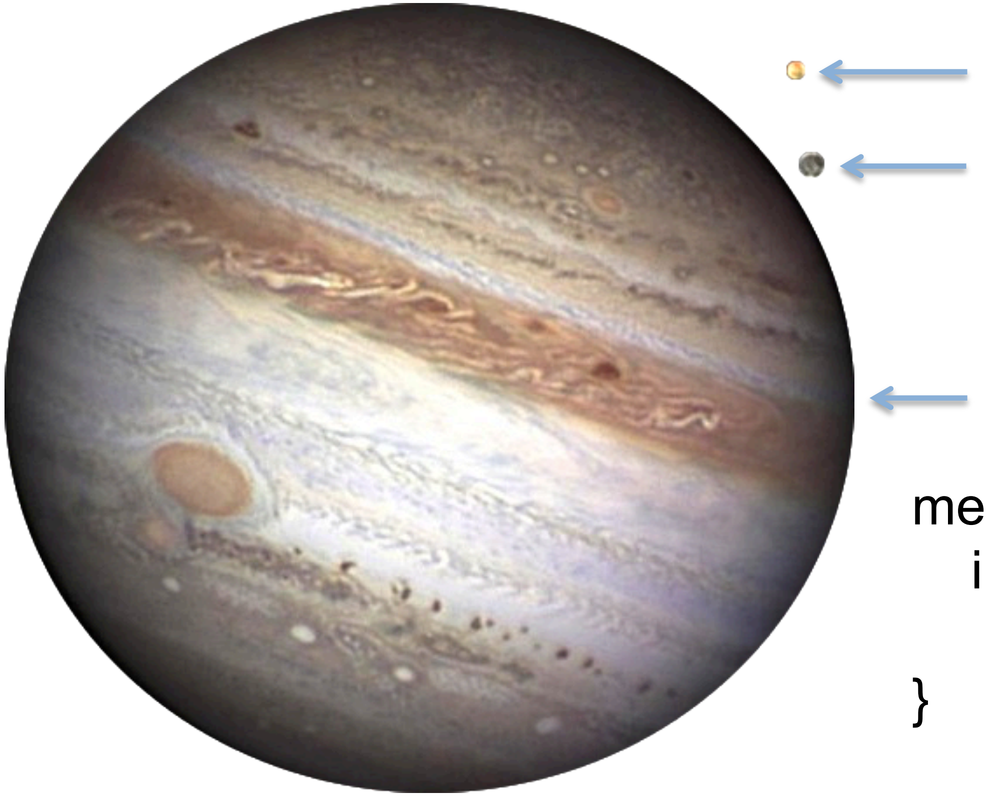


class Point

class Rect

class EverythingElseAndTheGUI_too

When OO in the wild fails (1)… “Jupiter Design”



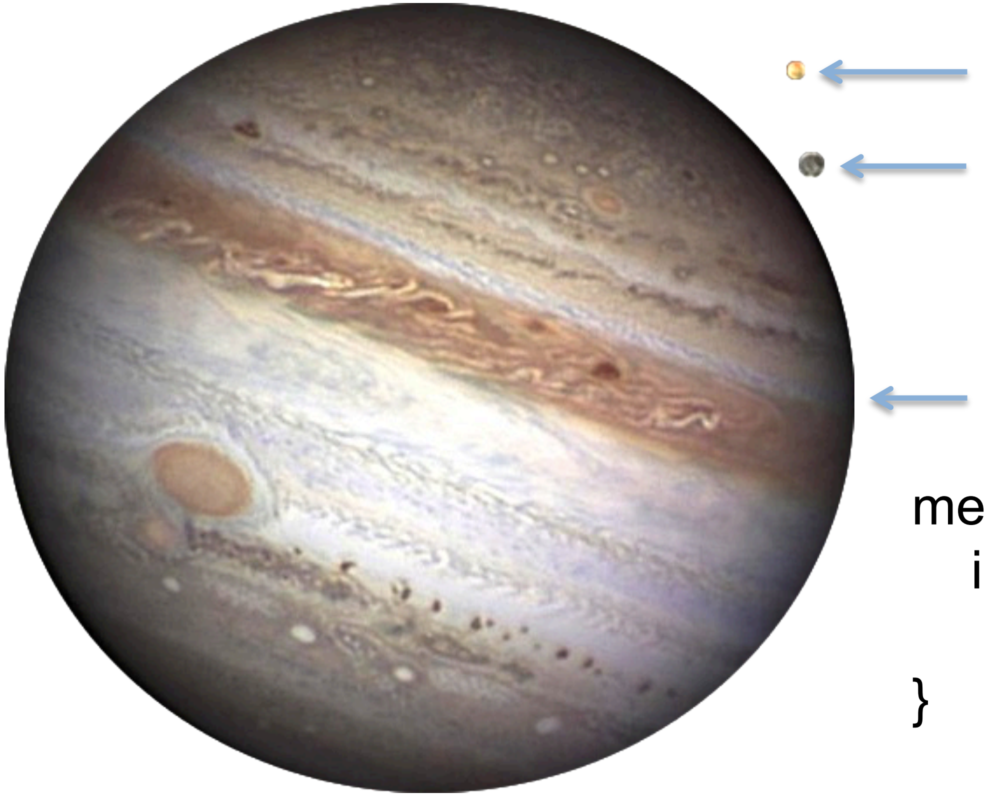
class Point

class Rect

class EverythingElseAndTheGUI_too

```
method innocuous_looking(void) {
    indirectly_access(potentially, any,
                                  instance, variable);
}
```

When OO in the wild fails (1)…   “Jupiter Design”



class Point

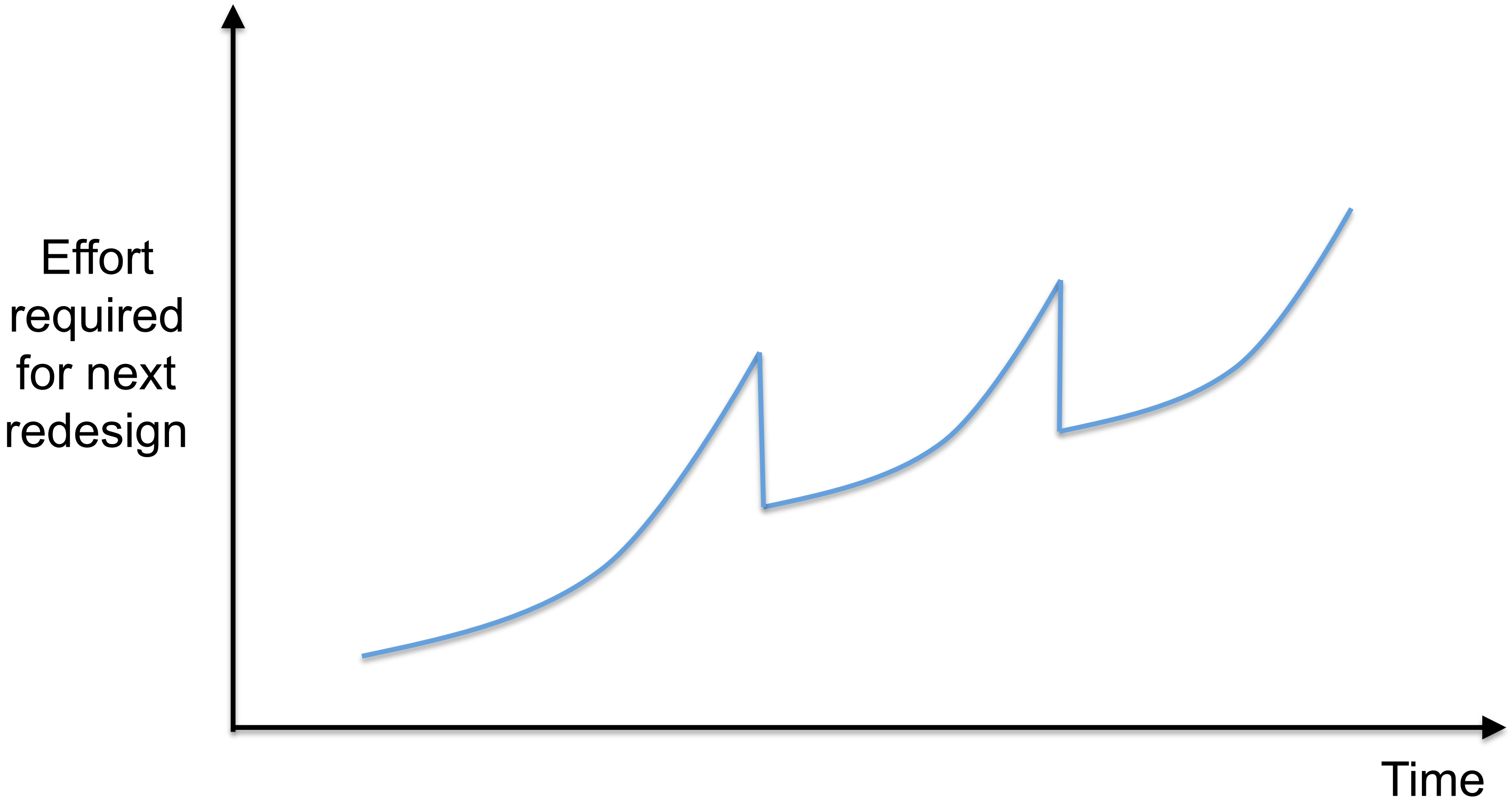
class Rect

class EverythingElseAndTheGUI_too

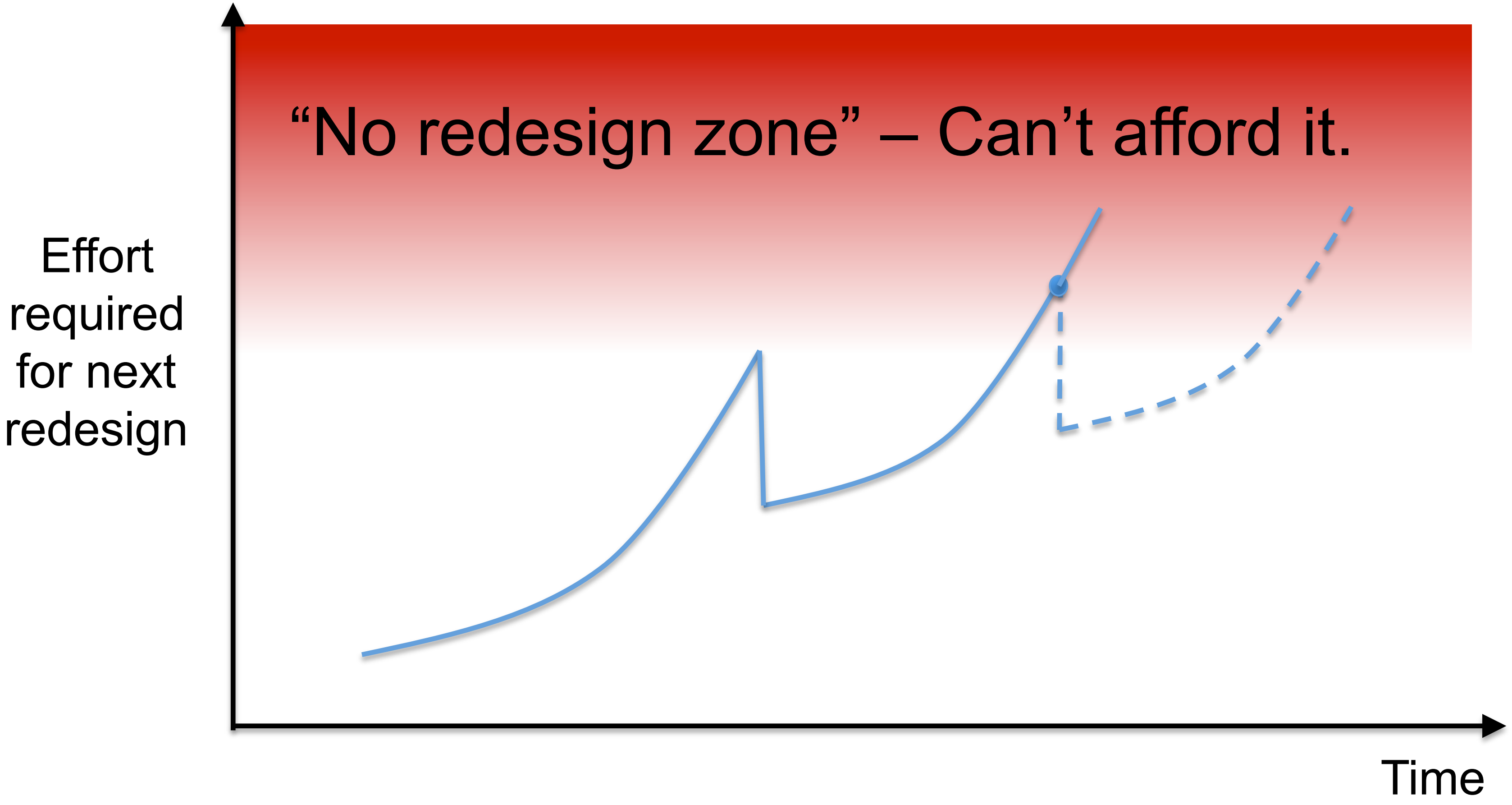
```
method innocuous_looking(void) {
    indirectly_access(potentially, any,
                                 instance, variable);
}
```

Cause of Failure: **Human** Error…
(“overuse of global variables”)

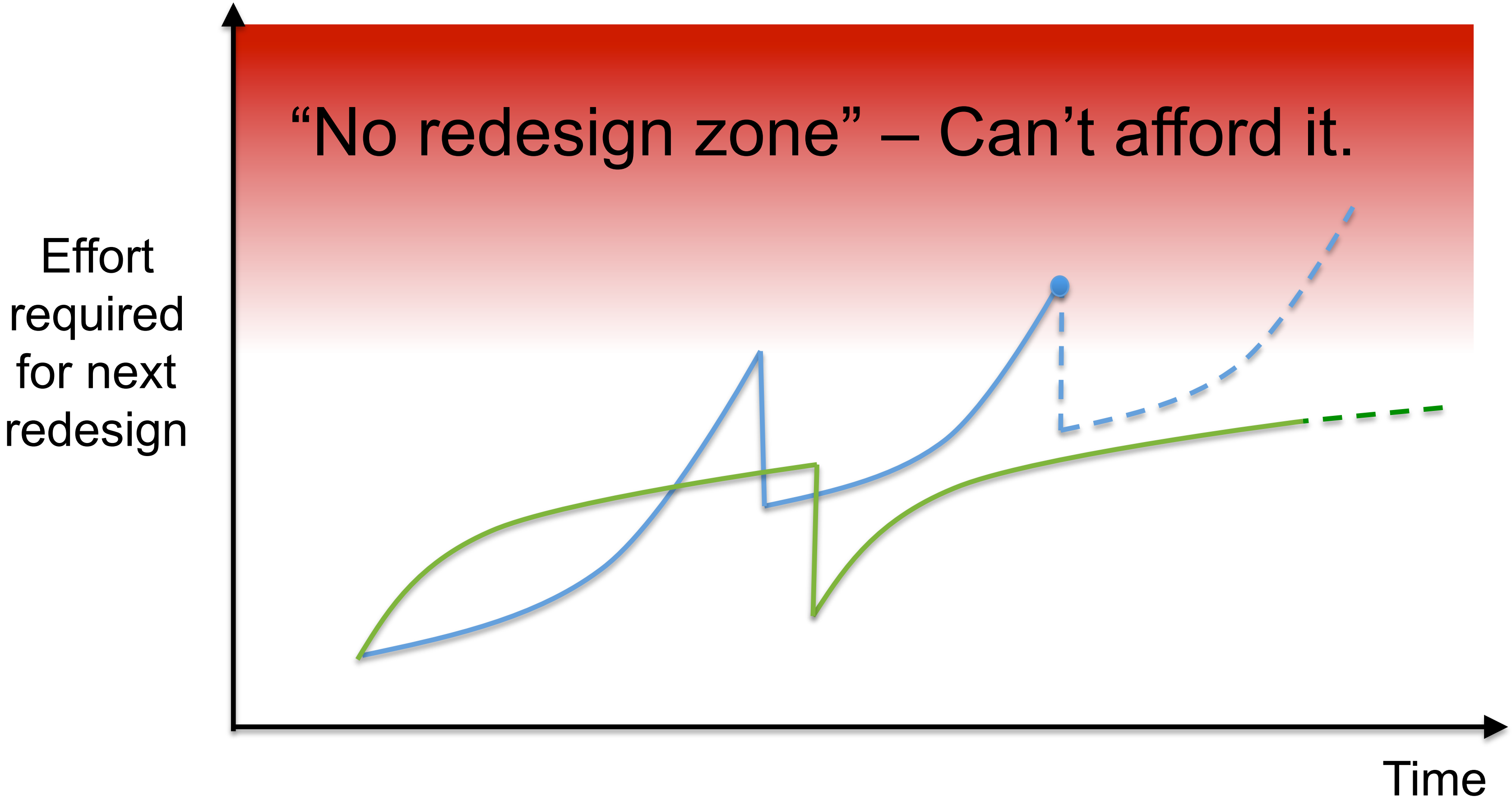
…but it's also related to the tools! *The Economics of Redesign*



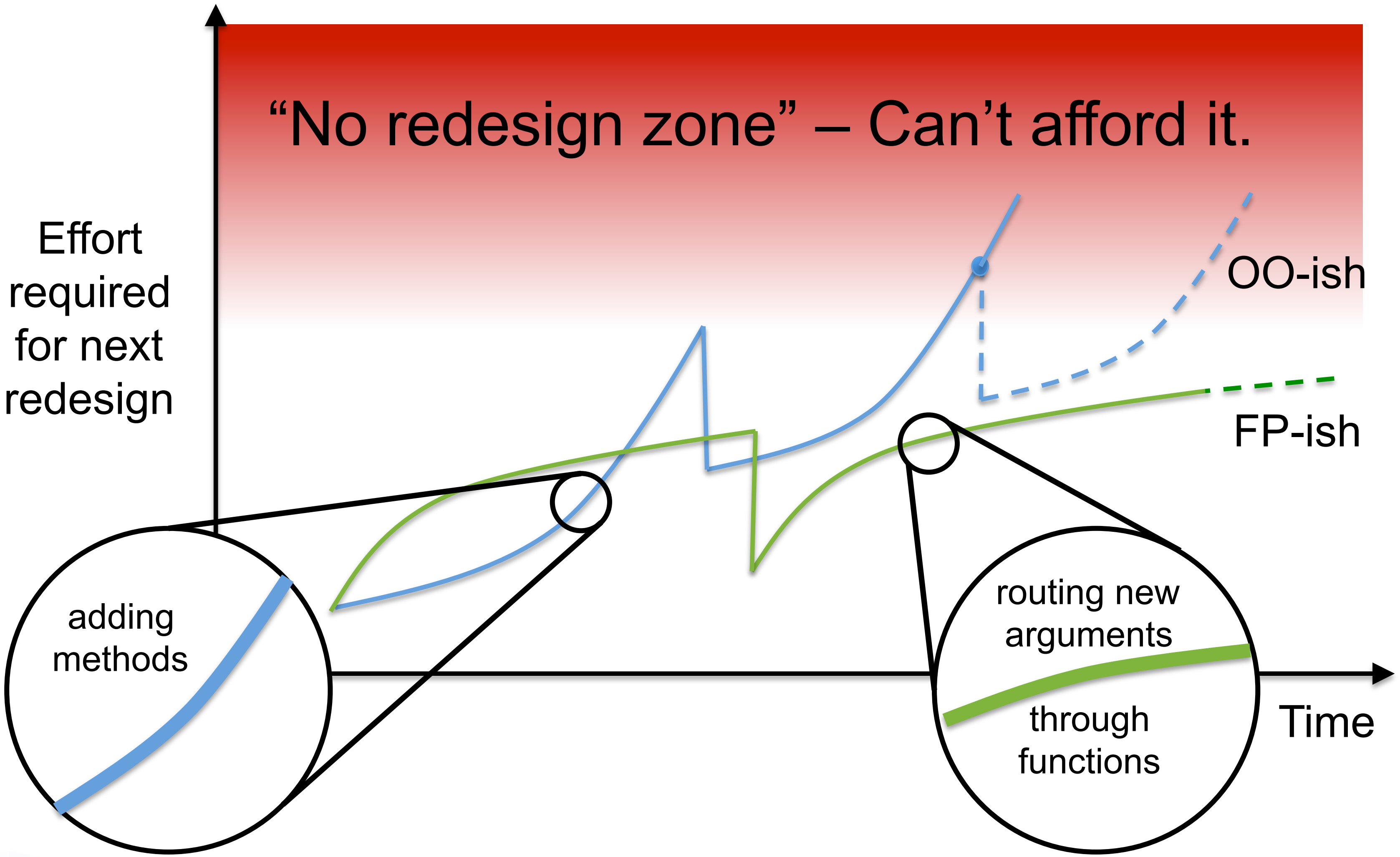
…but it's also related to the tools! *The Economics of Redesign*



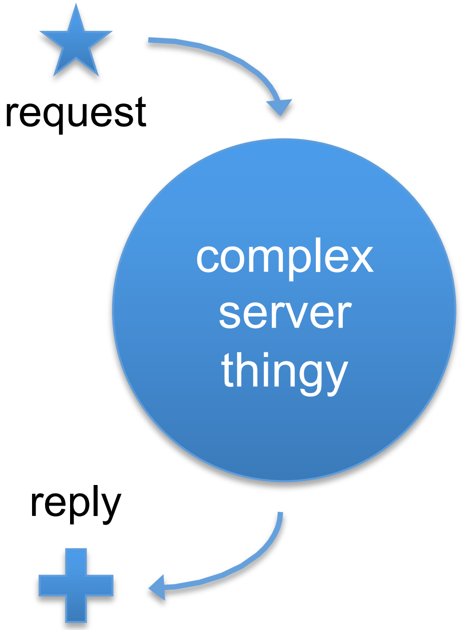
…but it's also related to the tools! *The Economics of Redesign*

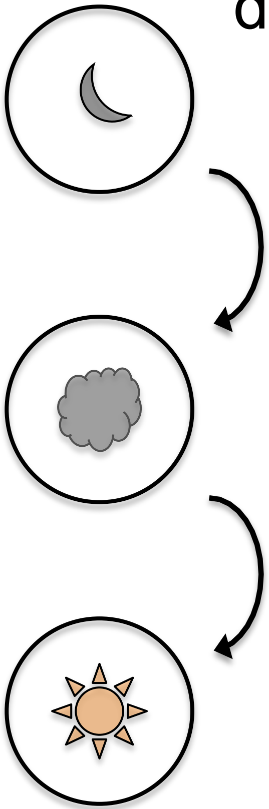


…but it's also related to the tools! *The Economics of Redesign*


"No redesign zone" – Can't afford it.
Effort required for next redesign
OO-ish
FP-ish
adding methods
routing new arguments through functions
Time

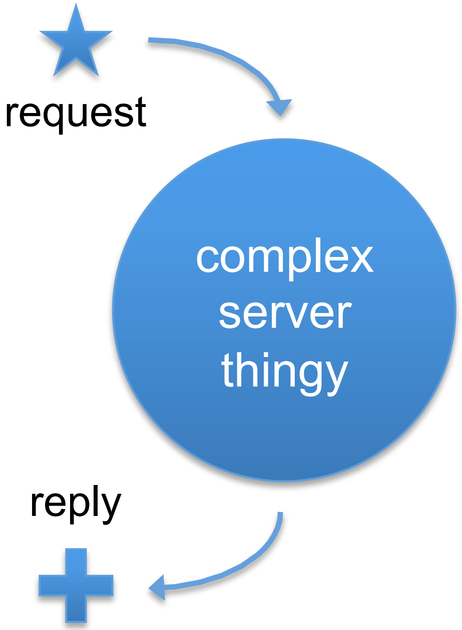
When OO in the wild fails (2)…   “State Limbo”

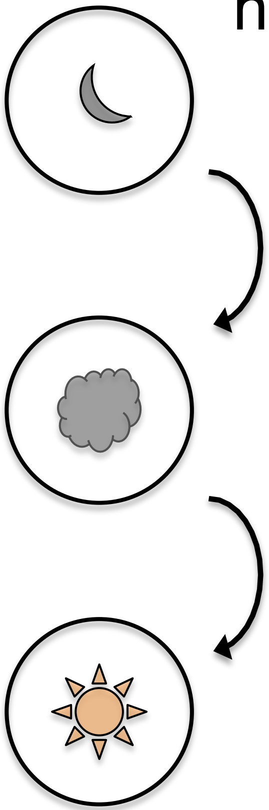




```
def handle_request(self):
        # self represents ‘moon’

        self.cloudy_moon_setter()

        # self represents ‘cloud’

        self.rise_and_shine()

        # self represents ‘sun’

        return ‘ok’
```

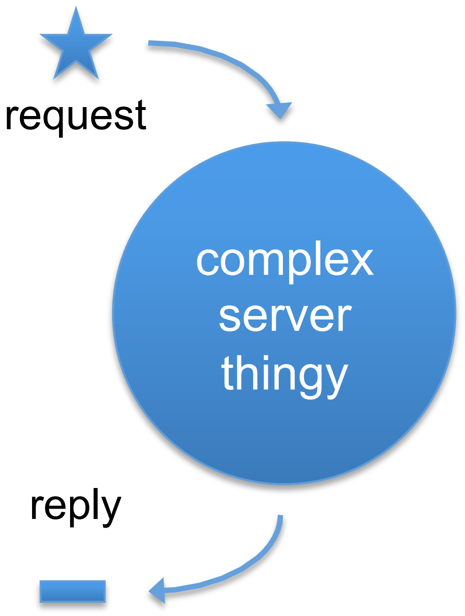
When OO in the wild fails (2)… “State Limbo”

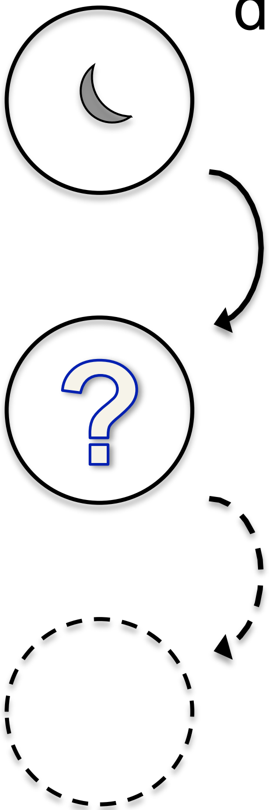




```
handle_call(Request, _, S1) ->
        % state S1 is ‘moon’

        S2 = cloudy_moon_set(S1)

        % made a new state S2 = ‘cloud’

        S3 = risen_and_shining(S2)

        % yet another state S3 = ‘sun’

        {reply, ok, S3}. % set next state
```

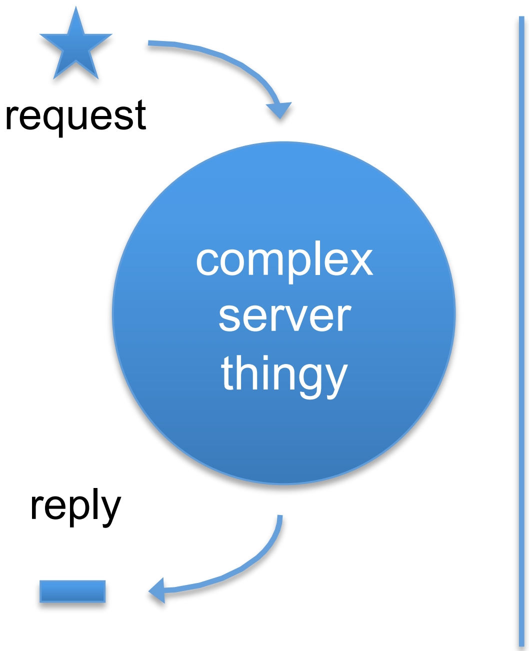
When OO in the wild fails (2)… “State Limbo”

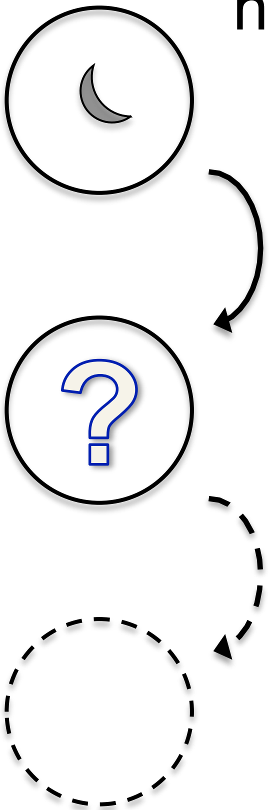


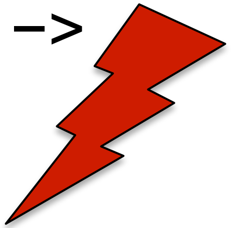


```
def handle_request(self):
        # self represents ‘moon’

        self.cloudy_moon_setter()

        # What is the state now?
        # Clean up OR press on?
        # But how?

        return ‘bummer’
```

When OO in the wild fails (2)… “State Limbo”

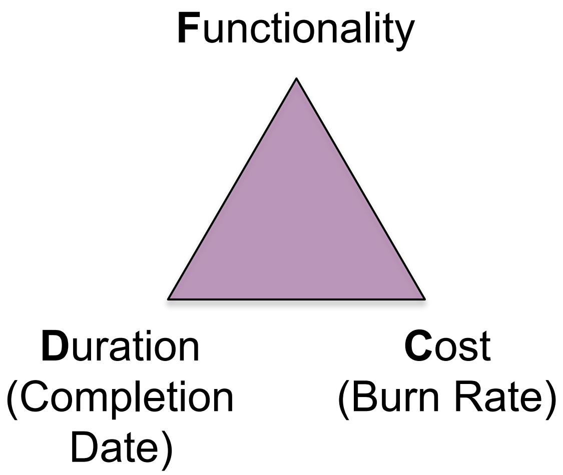






```
handle_call(Request, _, S1) ->
        % state S1 is ‘moon’

        S2 = cloudy_moon_set(S1)

        % The state is S1 + side-effects
        % from cloudy_moon_set/1.
        % Server state S1 is still around,
        % can be used to clean up.

        {reply, bummer, S1}.
```

The “2-out-of-3” Rule of Dealing of Project Risk



Fixed	*Risk*	*Remark*
FD	More resources	Traditional “waterfall” project
FC	Sliding deadlines	Traditional “institutional” project
DC	Feature starvation	“Agile” project
F	Slow & expensive	Confused “agile” with *carte blanche*
D	Unusable result	Endless financial renegotiation
C	Unusable result	Endless feature reprioritizing
FDC	Project locks up	“Ignore the risks” project, will blow up
-/-	Loss of focus	“Nothing gets ever done” project

Examples

FD: Module of deep space probe (dependability requirements, launch window)

FC: Next version of major operating system (functionality previews, limited resources for fixing bugs)

DC: Milestone of start-up company (expectations of partners, hours/day limited)

Programming Languages and the Power Grid

Summary

- "Power Grid" sounds more fixed and set than it actually is.
- Society's preferences for the power grid can and do change.
- Entelios AG is a young company helping stabilize the grid using the approach of *Demand Response*.
- *Functional Programming* concepts and tools have served us well in accomplishing this.
- Systems connected to the power grid could benefit by re-evaluating the basic assumptions.

Time for Questions