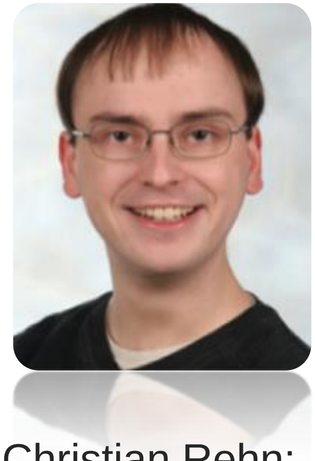
Design Types

Argumentativ zur besten Designentscheidung

<for developers only…maybe>



Christian Rehn:

- Software Engineer
- www.principles-wiki.net
- www.design-types.net

Matthias Wittum:

- Head of Source Center
- www.design-types.net

Roma
Ciampino
Villa Doria Pamphilj
Parco della Caffarella
Parco Naturale Regionale Appia Antica
Riserva Naturale Valle dei Casali
Stazione Tiburtina
Tiburtina
Via Monti Tiburtini; Via Lanciani
Via di Portonaccio
Giardinetti
Villaggio Breda
Villa Verde
La Romanina
Via Antolisei
Romanina
Giardini Tor di Mezzavia
Ponte Linari
Vermicino
Casal Morena
Morena
Centroni
Valle Copella
Aeroporto di Roma-Ciampino
Grottaferrata
Santa Maria
Castel di Leva
Fonte Laurentina
Mostacciano
Spinaceto
Vitinia
Tor de' Cenci
Tor de' Cenci - Spinaceto sud
Magliana Vecchia; Parco de' Medici
Fiumicino - EUR)
Via Portuense - Fiumicino
Viale Carlo Levi
GRA - Civitavecchia
Laurentina
Uscita 24 (Ardeatino)
Uscita 23 (Appia)
Gregna Sant'Andrea
Boccea)
della Stazione Aurelia
A90
A91
A1
SR148
SP3e
SP2c
SP77b
© OpenStreetMap-Mitwirkende

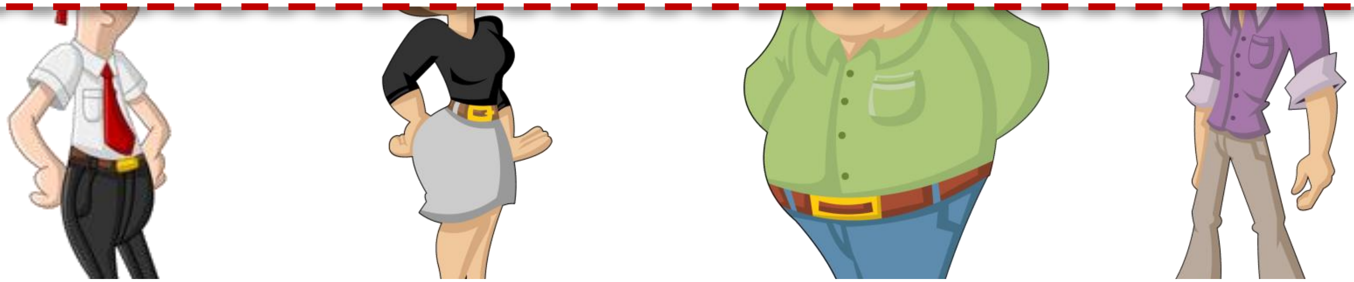
Es gibt viele Wege
um ans Ziel zu kommen…

…direkt
…schnell
…mit Abstechern
…explorativ

Szenario

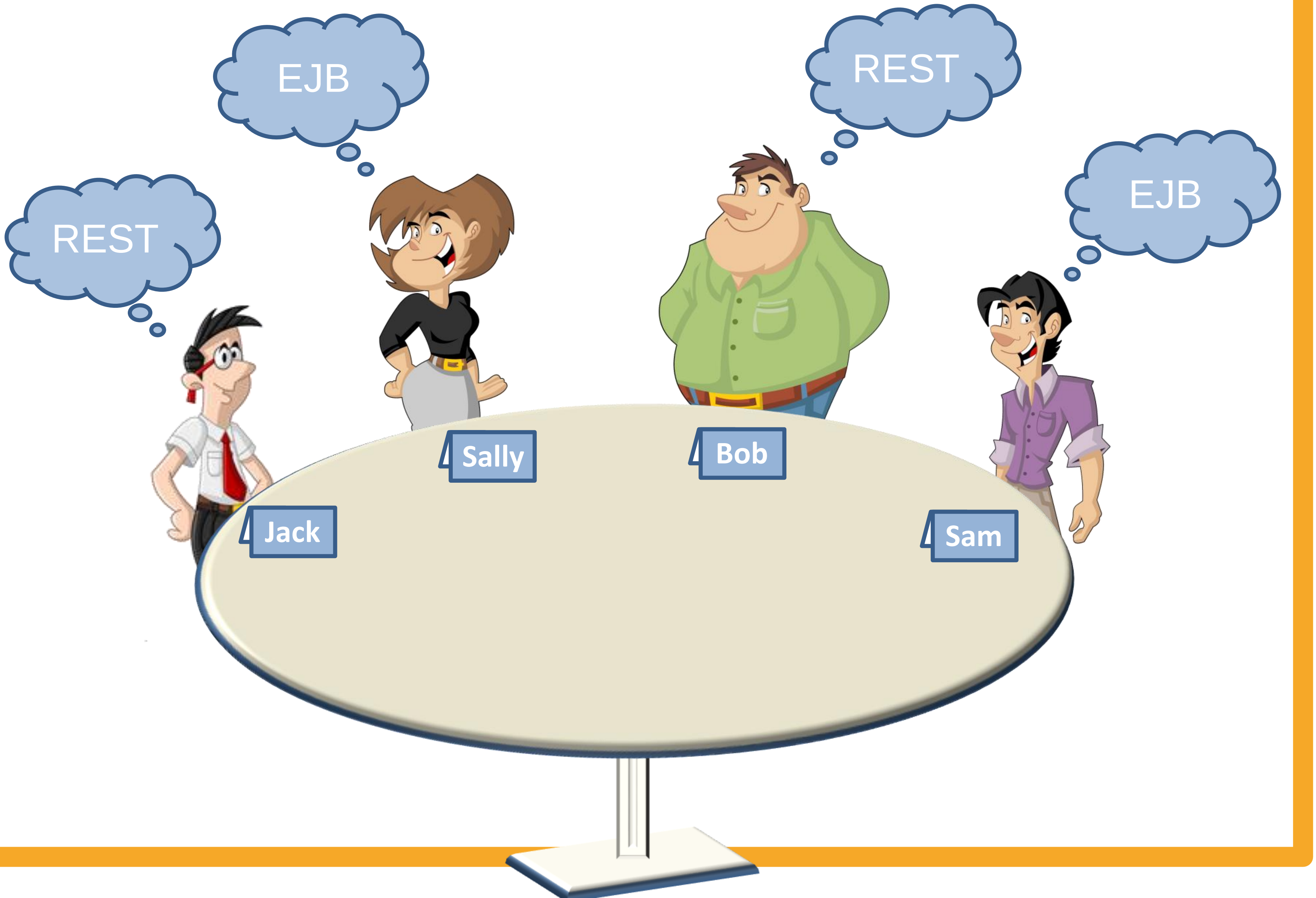
vs.

Ein völlig fiktives Szenario

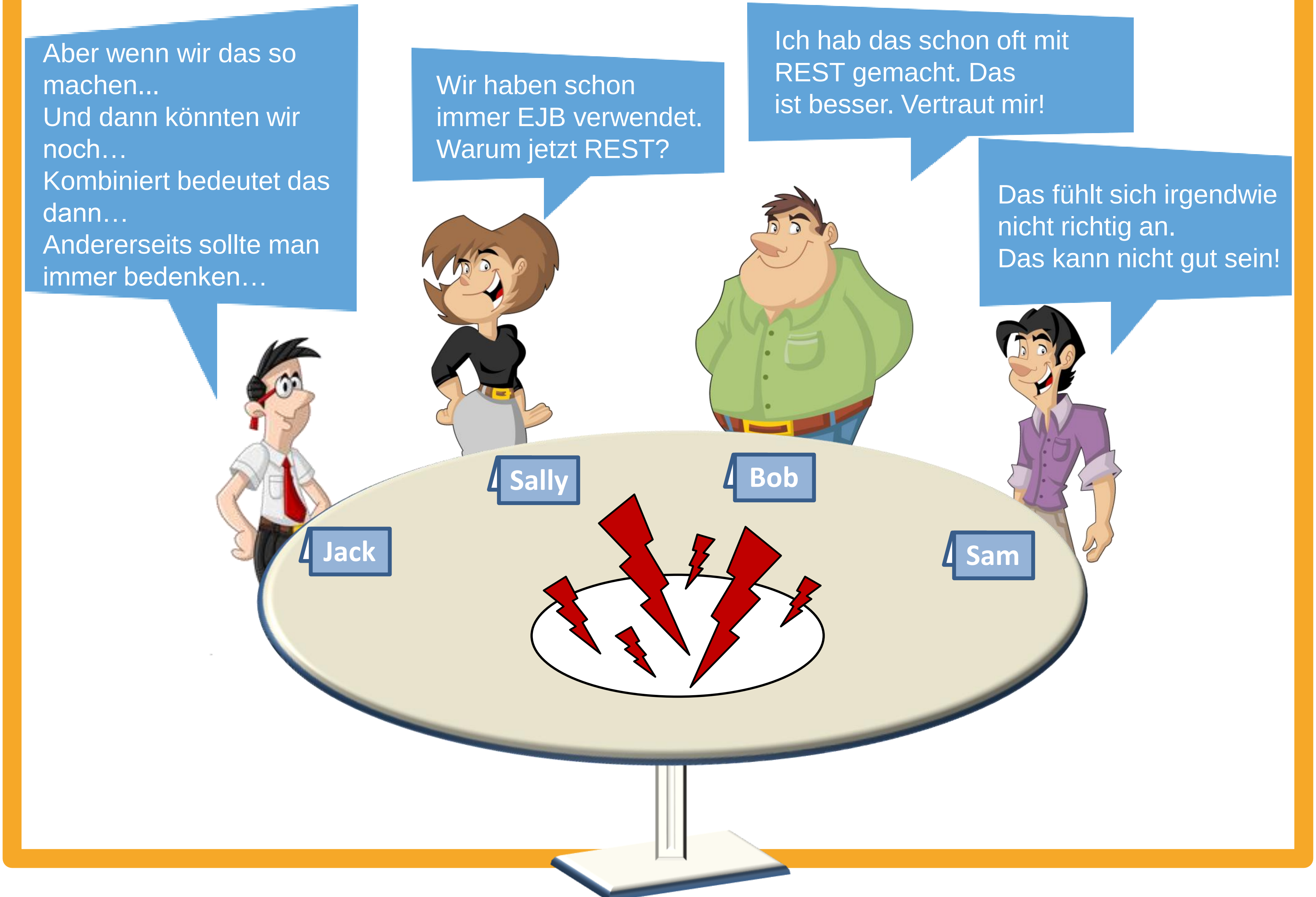


Eine Gruppe Entwickler diskutiert, ob eine neue Schnittstelle per EJB oder REST angeboten werden soll.

Technische Diskussion: „die historische Variante“



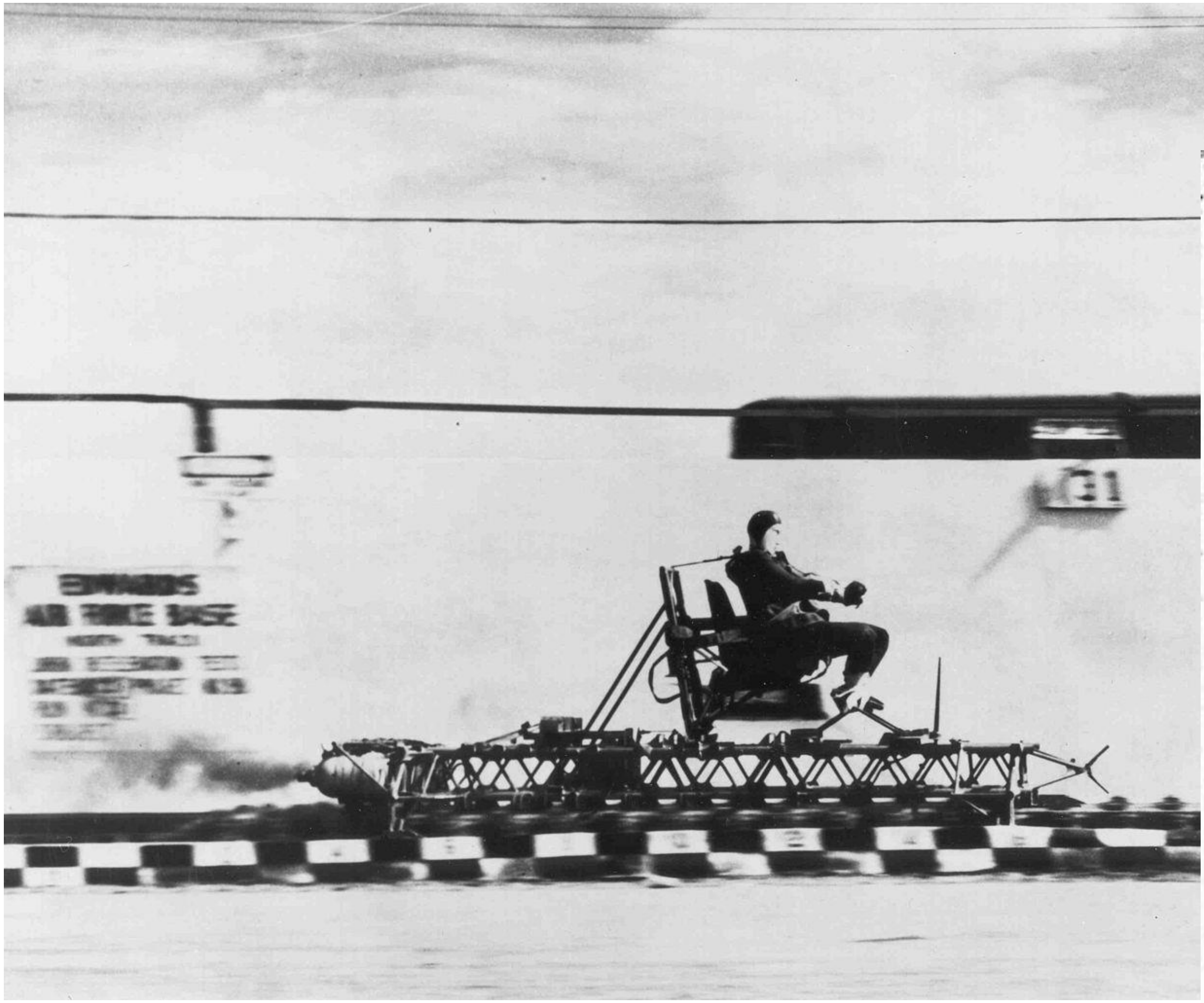
Technische Diskussion: „die historische Variante“


Aber wenn wir das so machen... Und dann könnten wir noch… Kombiniert bedeutet das dann… Andererseits sollte man immer bedenken…
Wir haben schon immer EJB verwendet. Warum jetzt REST?
Ich hab das schon oft mit REST gemacht. Das ist besser. Vertraut mir!
Das fühlt sich irgendwie nicht richtig an. Das kann nicht gut sein!
Sally
Bob
Jack
Sam

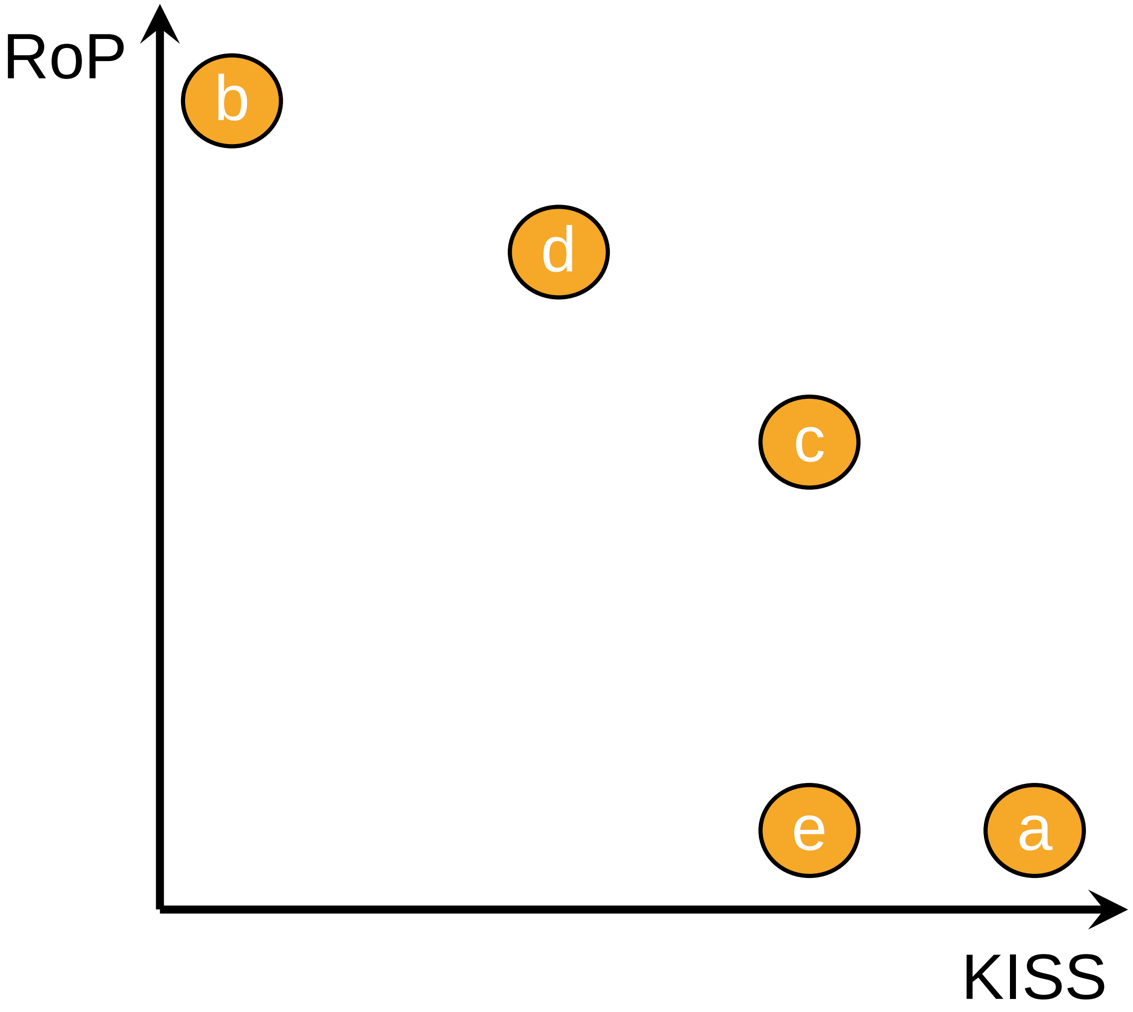
Überreden oder überzeugen?

Wir sind nicht die ersten...

John Paul Stapp und Edward A. Murphy



Konträre Prinzipien



IH/E

SDP

SoC

ML

KISS

ISP

SCP

DRY

SRP

LoD

OCP

FF

RoP

DIP

ECV

ZOI

MIMC

LC

UP

LLA

PSU

HC

EUHM

MP

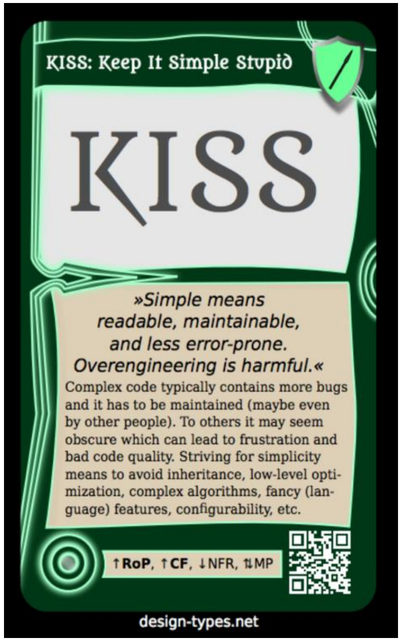
TdA/IE

PLS

IAP

ADP

Design Cards – Argumentkarten



KISS: Keep It Simple Stupid

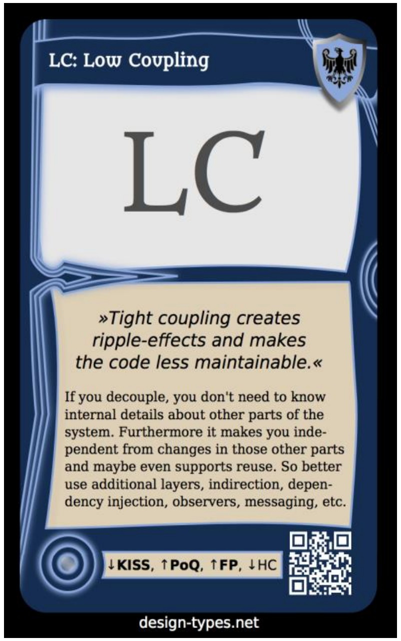
KISS

»Simple means readable, maintainable, and less error-prone. Overengineering is harmful.«

Complex code typically contains more bugs and it has to be maintained (maybe even by other people). To others it may seem obscure which can lead to frustration and bad code quality. Striving for simplicity means to avoid inheritance, low-level optimization, complex algorithms, fancy (language) features, configurability, etc.

↑**RoP**, ↑**CF**, ↓NFR, ⇅MP

design-types.net



LC: Low Coupling

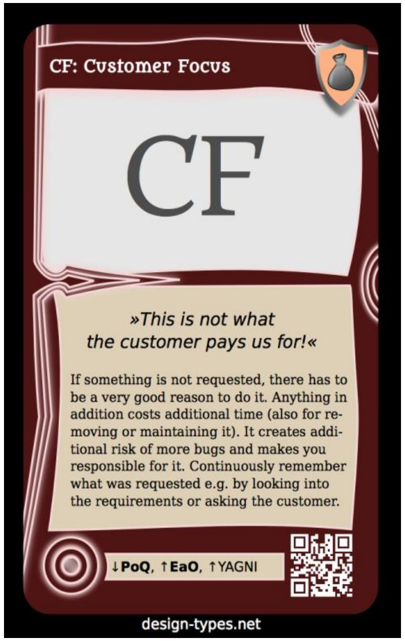
LC

»Tight coupling creates ripple-effects and makes the code less maintainable.«

If you decouple, you don't need to know internal details about other parts of the system. Furthermore it makes you independent from changes in those other parts and maybe even supports reuse. So better use additional layers, indirection, dependency injection, observers, messaging, etc.

↓**KISS**, ↑**PoQ**, ↑**FP**, ↓HC

design-types.net



CF: Customer Focus

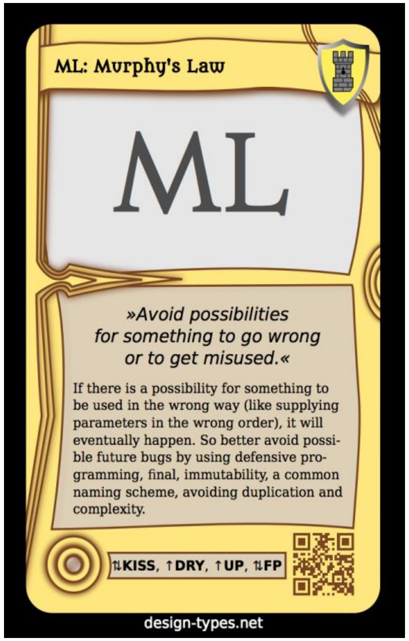
CF

»This is not what the customer pays us for!«

If something is not requested, there has to be a very good reason to do it. Anything in addition costs additional time (also for removing or maintaining it). It creates additional risk of more bugs and makes you responsible for it. Continuously remember what was requested e.g. by looking into the requirements or asking the customer.

↓**PoQ**, ↑**EaO**, ↑YAGNI

design-types.net



ML: Murphy's Law

ML

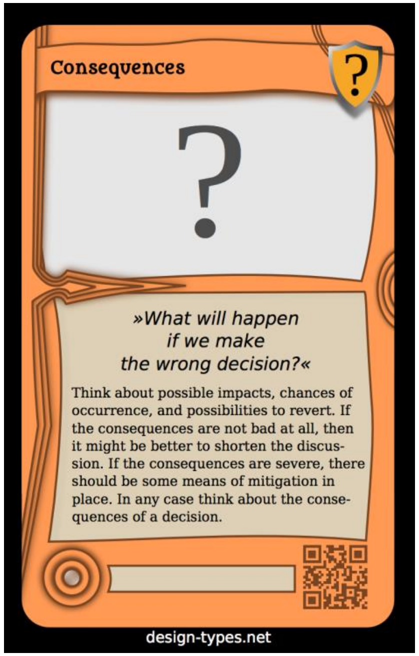
»Avoid possibilities for something to go wrong or to get misused.«

If there is a possibility for something to be used in the wrong way (like supplying parameters in the wrong order), it will eventually happen. So better avoid possible future bugs by using defensive programming, final, immutability, a common naming scheme, avoiding duplication and complexity.

⇅**KISS**, ↑**DRY**, ↑**UP**, ⇅**FP**

design-types.net

Design Cards – Moderationskarten



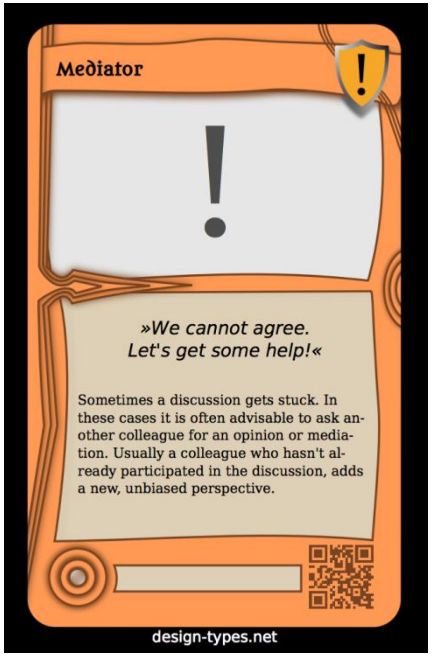
Consequences

?

»What will happen if we make the wrong decision?«

Think about possible impacts, chances of occurrence, and possibilities to revert. If the consequences are not bad at all, then it might be better to shorten the discussion. If the consequences are severe, there should be some means of mitigation in place. In any case think about the consequences of a decision.

design-types.net



Mediator

!

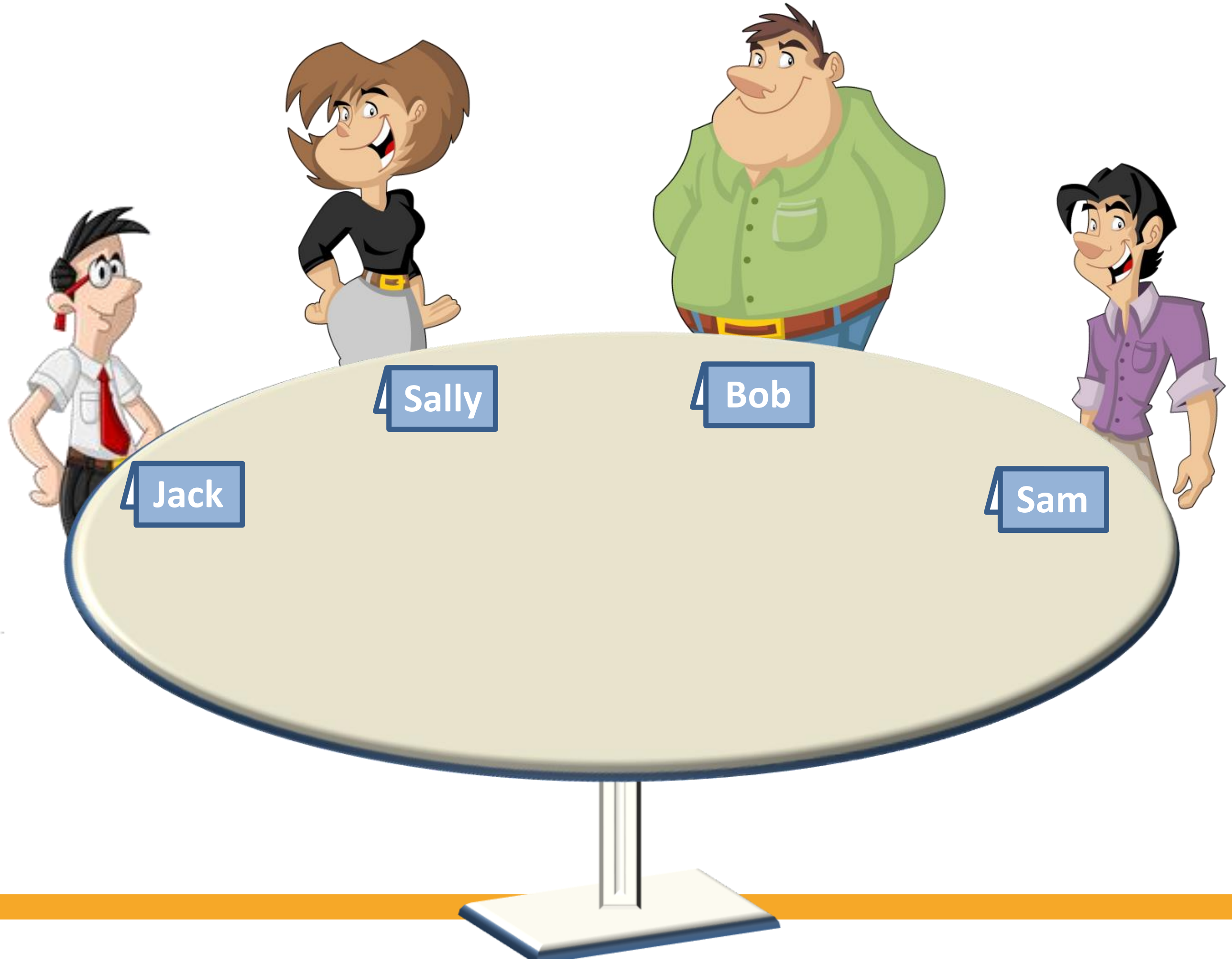
»We cannot agree. Let's get some help!«

Sometimes a discussion gets stuck. In these cases it is often advisable to ask another colleague for an opinion or mediation. Usually a colleague who hasn't already participated in the discussion, adds a new, unbiased perspective.

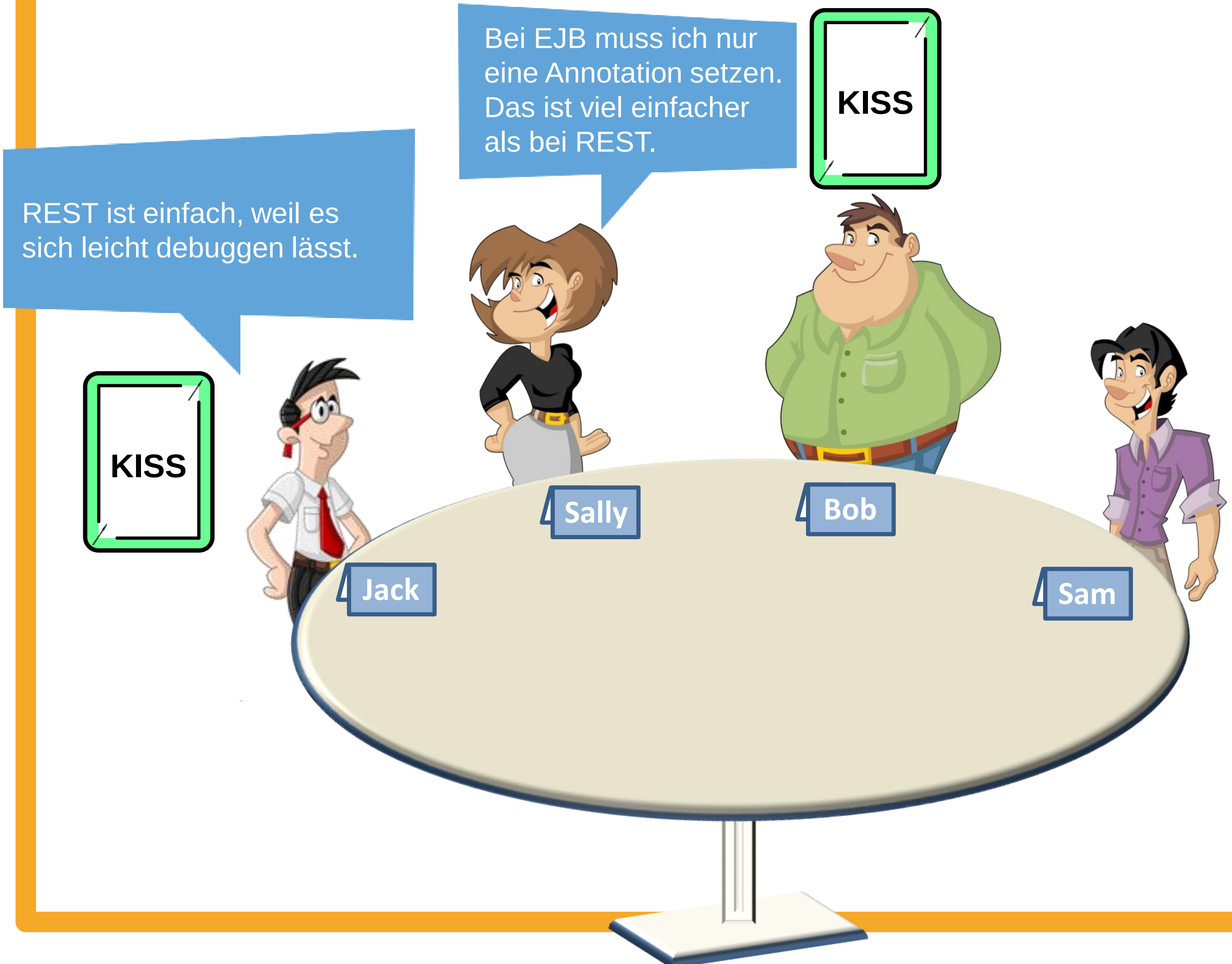
design-types.net

Technische Diskussion: „kartengestützte Argumentation“

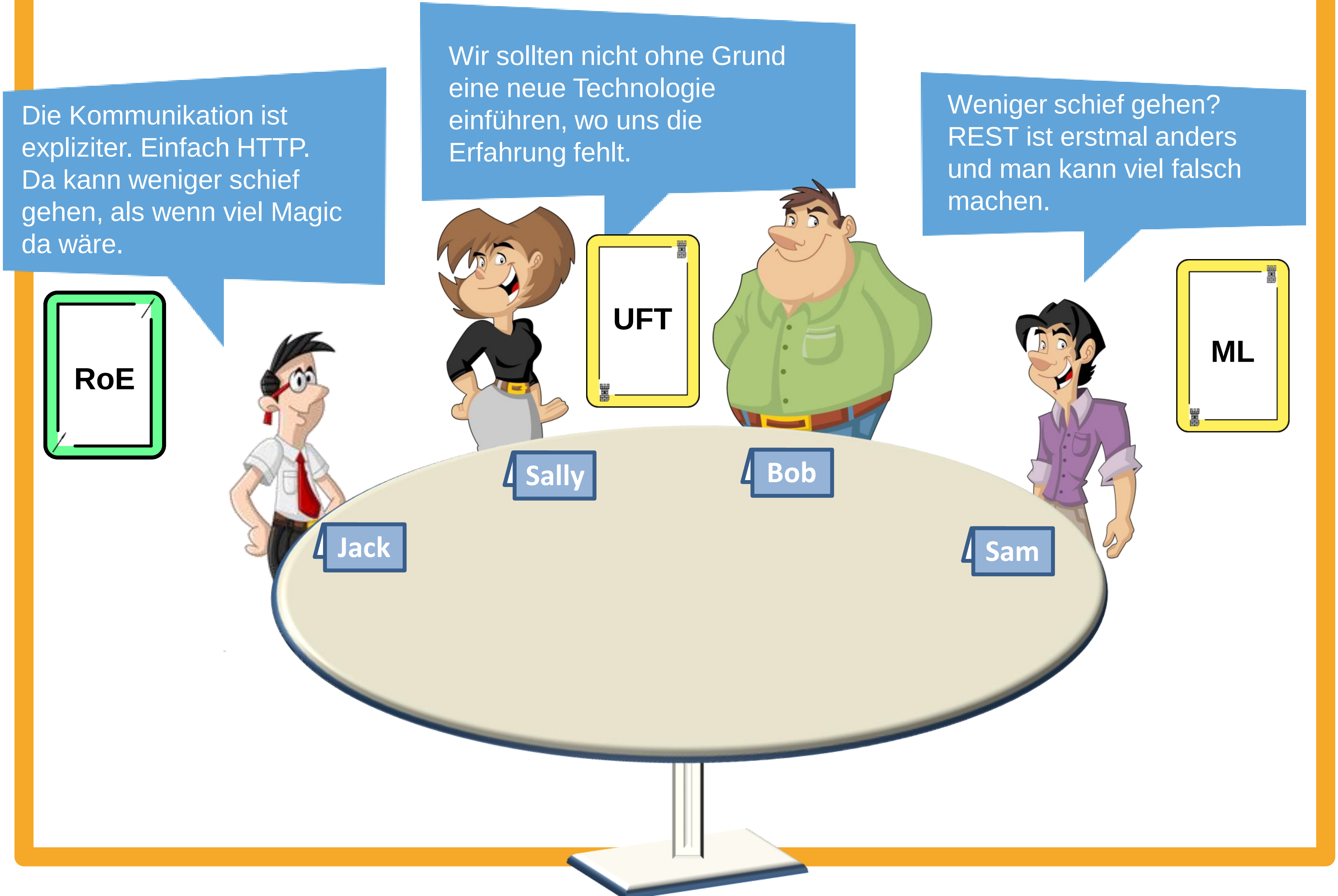
Ziel: Einsatz von nachvollziehbaren Argumenten



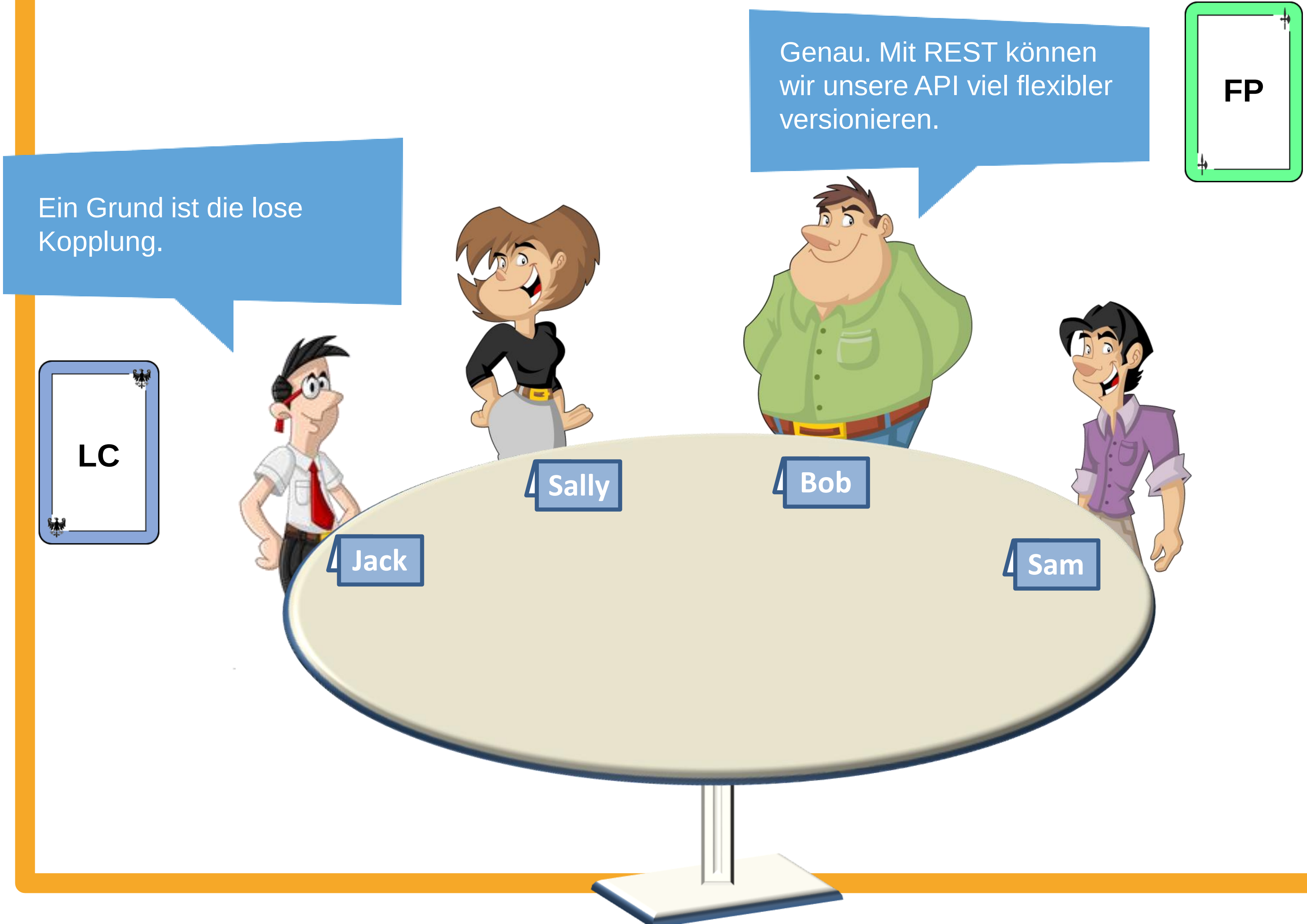
Technische Diskussion: „kartengestützte Argumentation“


REST ist einfach, weil es sich leicht debuggen lässt.
Bei EJB muss ich nur eine Annotation setzen. Das ist viel einfacher als bei REST.
KISS
KISS
Jack
Sally
Bob
Sam

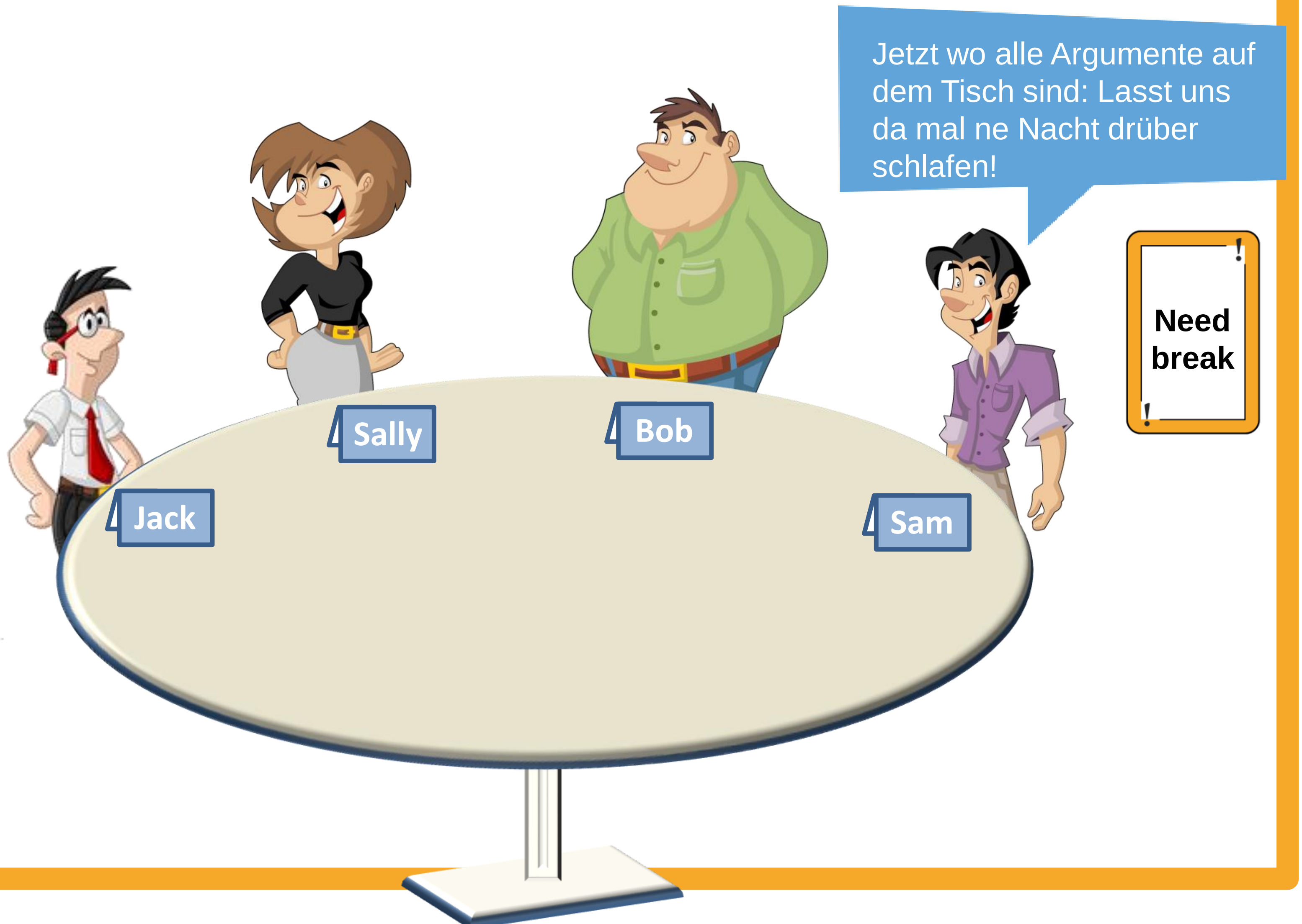
Technische Diskussion: „kartengestützte Argumentation“


Die Kommunikation ist expliziter. Einfach HTTP. Da kann weniger schief gehen, als wenn viel Magic da wäre.
RoE
Wir sollten nicht ohne Grund eine neue Technologie einführen, wo uns die Erfahrung fehlt.
UFT
Weniger schief gehen? REST ist erstmal anders und man kann viel falsch machen.
ML
Sally
Bob
Jack
Sam

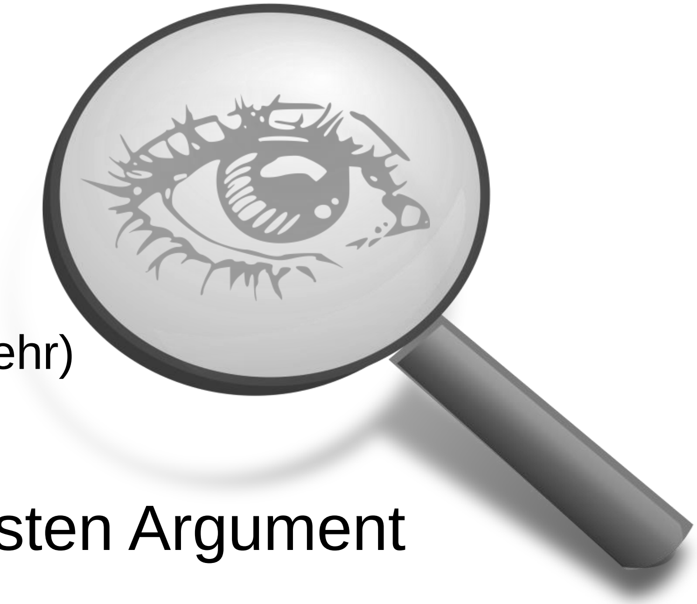
Technische Diskussion: „kartengestützte Argumentation“


Genau. Mit REST können wir unsere API viel flexibler versionieren.
FP
Ein Grund ist die lose Kopplung.
LC
Sally
Bob
Jack
Sam

Technische Diskussion: „kartengestützte Argumentation“


Jetzt wo alle Argumente auf dem Tisch sind: Lasst uns da mal ne Nacht drüber schlafen!
Need break
Sally
Bob
Jack
Sam

Erste Beobachtung

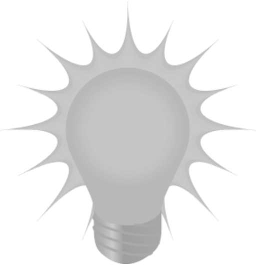


- Die Entwickler argumentieren klarer und nachvollziehbarer
  (Jack überrennt seine Kollegen nicht mehr)

- Ein Argument leitet zum nächsten Argument oder Gegenargument über

Einsatzgebiete

- Konzeption



- Pair Programming

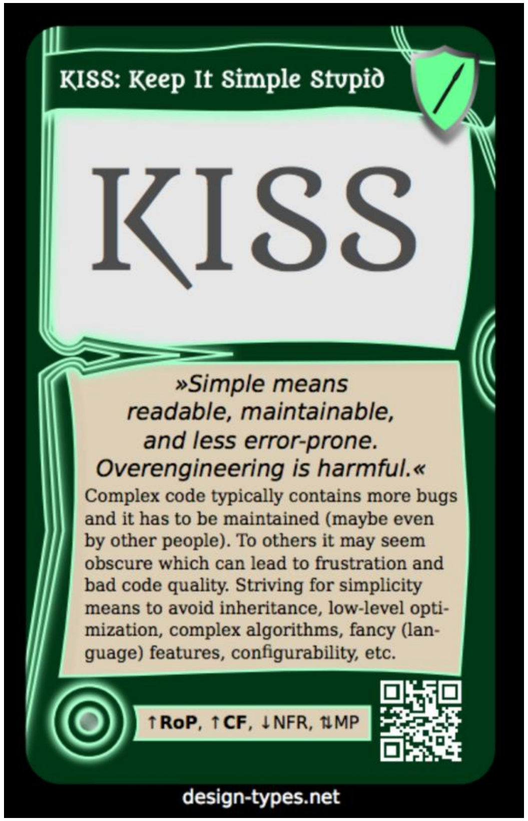
- Code Reviews

```
#include "utilities.h"

Game game;

int main(int argc, char **argv) {

game = Game();

game.initialize();
}
```

- Gamification

Die Karten im Detail

Simple
design-types.net



KISS: Keep It Simple Stupid

KISS

»Simple means readable, maintainable, and less error-prone. Overengineering is harmful.«

Complex code typically contains more bugs and it has to be maintained (maybe even by other people). To others it may seem obscure which can lead to frustration and bad code quality. Striving for simplicity means to avoid inheritance, low-level optimization, complex algorithms, fancy (language) features, configurability, etc.

↑**RoP**, ↑**CF**, ↓NFR, ⇅MP

design-types.net

Wir haben noch gar keinen Konsens gefunden!

Die Dimensionen unserer Entwickler-Typologie

Simple	vs.	Powerful
Abstract	vs.	Concrete
Pragmatic	vs.	Idealistic
Robust	vs.	Technologic

Die konkreten Ausprägungen

S

Simple means:
- to keep code simple for better understandability
- to omit unnecessary things (lower risk; fewer bugs)
- to reduce complexity
- to prefer explicit solutions instead of implicit ones
- etc.

Powerful means:
- to build powerful and generalized solutions
- to have flexibility/extensibility by foresighted design
- to have configurable solutions
- to master complexity
- etc.

C

Abstract means:
- to think in concepts and abstractions
- to focus on the big picture and component interactions
- to know all potential consequences of a change
- to build models of the real world
- etc.

Concrete means:
- to think in code or simultaneously transfer ideas into code immediately
- to optimize algortihms for better performance
- to understand systems by reading the code
- etc.

I

Pragmatic means:
- to fulfill requirements asap
- to focus on customer needs to guarantee a value
- to omit unnecessary things
- to bring others down to earth
- etc.

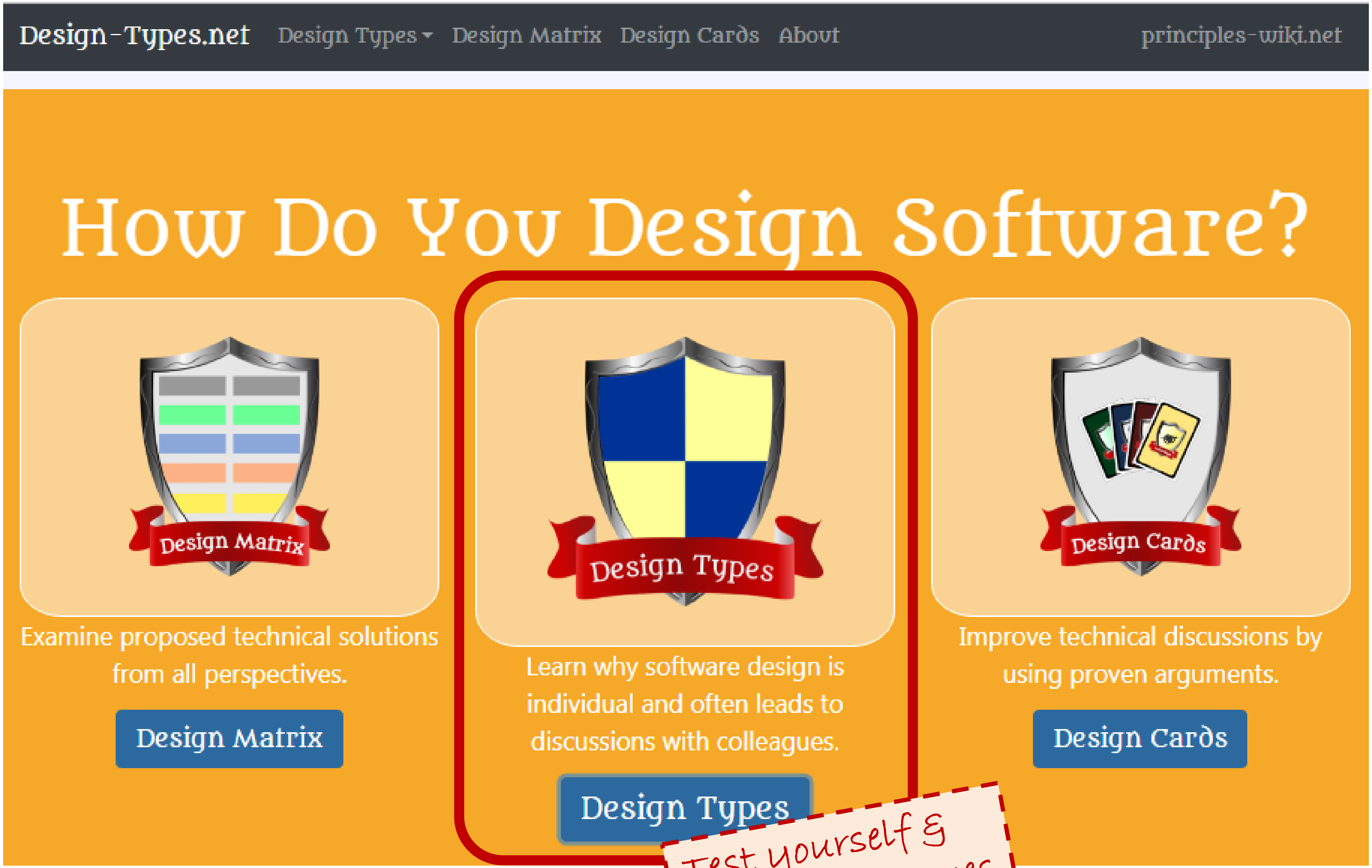
Idealistic means:
- to make things right – not only 80%
- to consider all aspects not only functional ones
- to know that everything has its right place
- to do not misuse existing concepts, APIs, etc.
- etc.

R

Robust means:
- to protect applications against risks and potential bugs
- to define and adhere to standards
- to avoid too much magic and complexity to reduce risks
- to use proven solutions which stood the test of time
- etc.

Technologic means:
- to use new, modern and more productive technologies and to get rid of legacy
- to evolve with technology to be more competitive
- to broaden your personal horizon
- etc.

Selbst ausprobieren: www.design-types.net



Design-Types.net Design Types ▾ Design Matrix Design Cards About principles-wiki.net

How Do You Design Software?

Examine proposed technical solutions from all perspectives.

Design Matrix

Learn why software design is individual and often leads to discussions with colleagues.

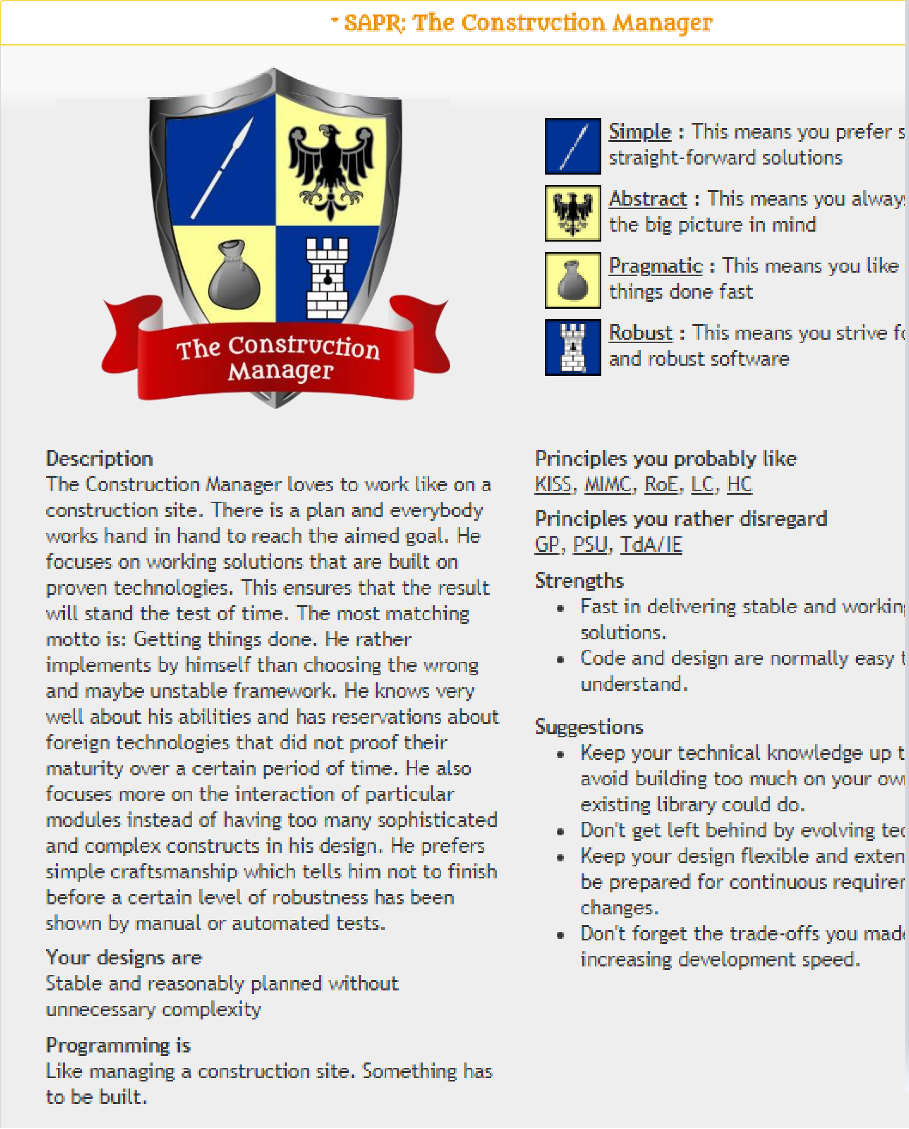
Design Types

Improve technical discussions by using proven arguments.

Design Cards

Test yourself &
Assess colleagues

Ein Beispiel-Ergebnis



SAPR: The Construction Manager

Simple : This means you prefer s straight-forward solutions

Abstract : This means you alway the big picture in mind

Pragmatic : This means you like things done fast

Robust : This means you strive fo and robust software

Description
The Construction Manager loves to work like on a construction site. There is a plan and everybody works hand in hand to reach the aimed goal. He focuses on working solutions that are built on proven technologies. This ensures that the result will stand the test of time. The most matching motto is: Getting things done. He rather implements by himself than choosing the wrong and maybe unstable framework. He knows very well about his abilities and has reservations about foreign technologies that did not proof their maturity over a certain period of time. He also focuses more on the interaction of particular modules instead of having too many sophisticated and complex constructs in his design. He prefers simple craftsmanship which tells him not to finish before a certain level of robustness has been shown by manual or automated tests.

Your designs are
Stable and reasonably planned without unnecessary complexity

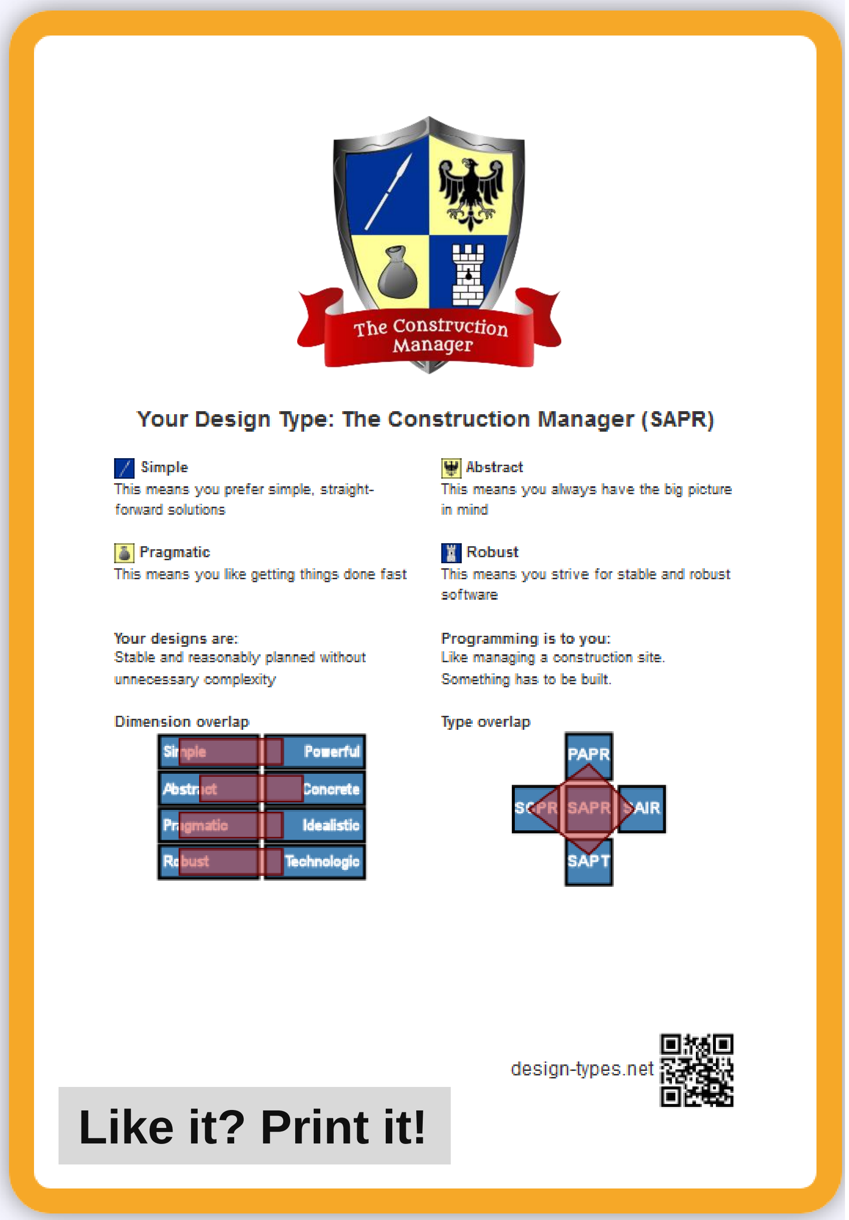
Programming is
Like managing a construction site. Something has to be built.

Principles you probably like
KISS, MIMC, RoE, LC, HC

Principles you rather disregard
GP, PSU, TdA/IE

Strengths
- Fast in delivering stable and workin solutions.
- Code and design are normally easy t understand.

Suggestions
- Keep your technical knowledge up t avoid building too much on your ow existing library could do.
- Don't get left behind by evolving tec
- Keep your design flexible and exten be prepared for continuous requirer changes.
- Don't forget the trade-offs you mad increasing development speed.



Your Design Type: The Construction Manager (SAPR)

Simple
This means you prefer simple, straight-forward solutions

Abstract
This means you always have the big picture in mind

Pragmatic
This means you like getting things done fast

Robust
This means you strive for stable and robust software

Your designs are:
Stable and reasonably planned without unnecessary complexity

Programming is to you:
Like managing a construction site. Something has to be built.

Dimension overlap

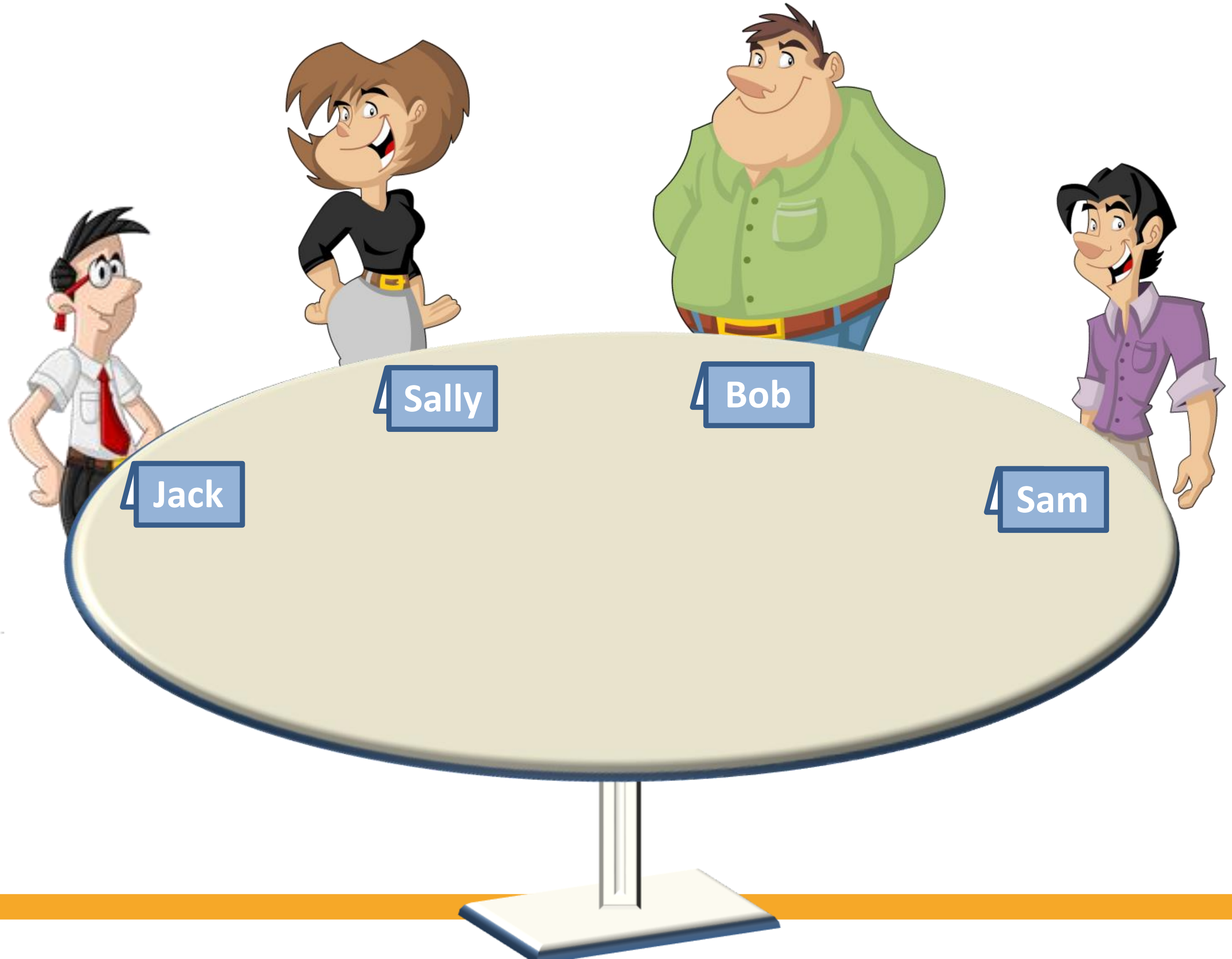
Type overlap

design-types.net

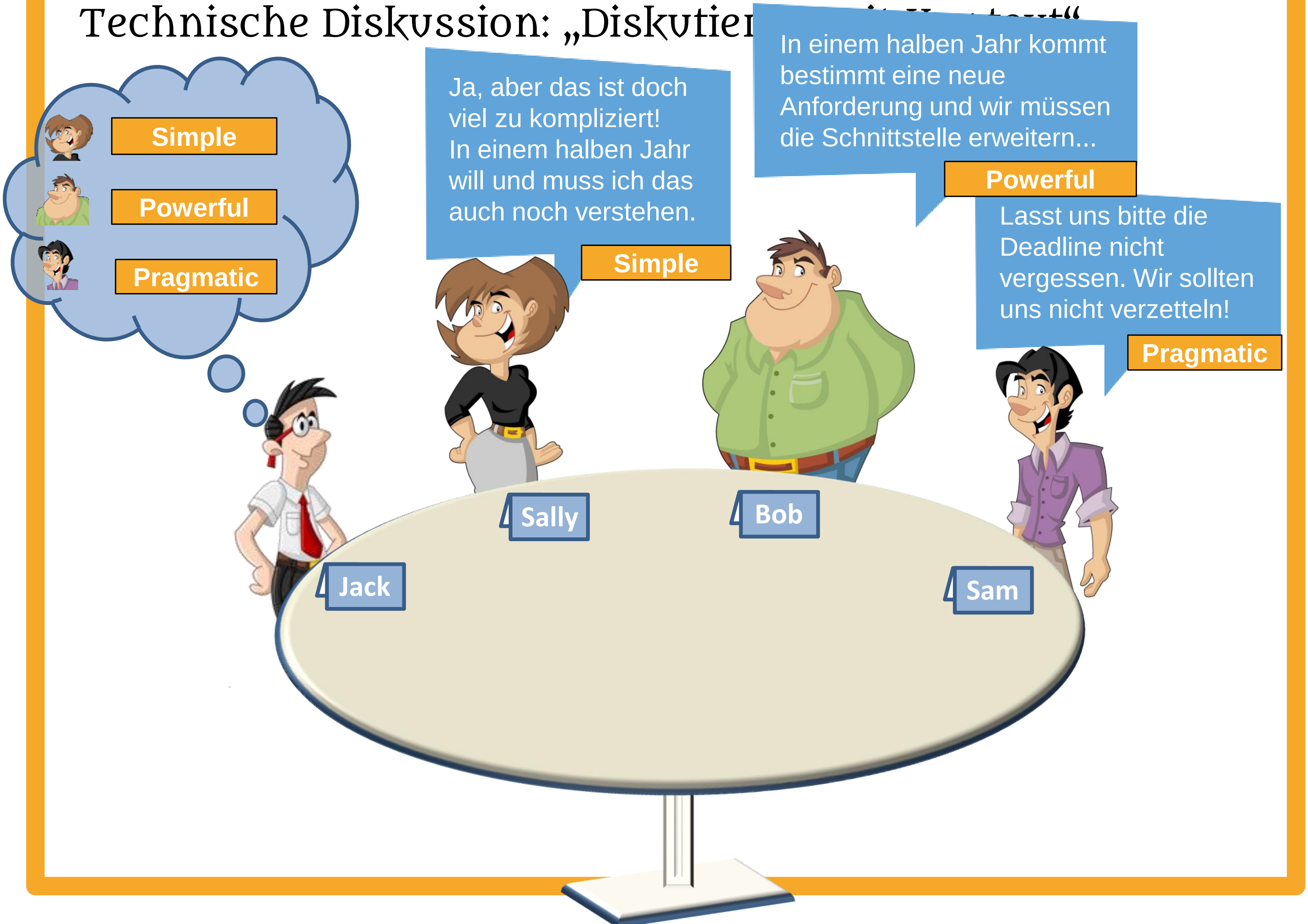
Like it? Print it!

Technische Diskussion: „Diskutieren mit Kontext“

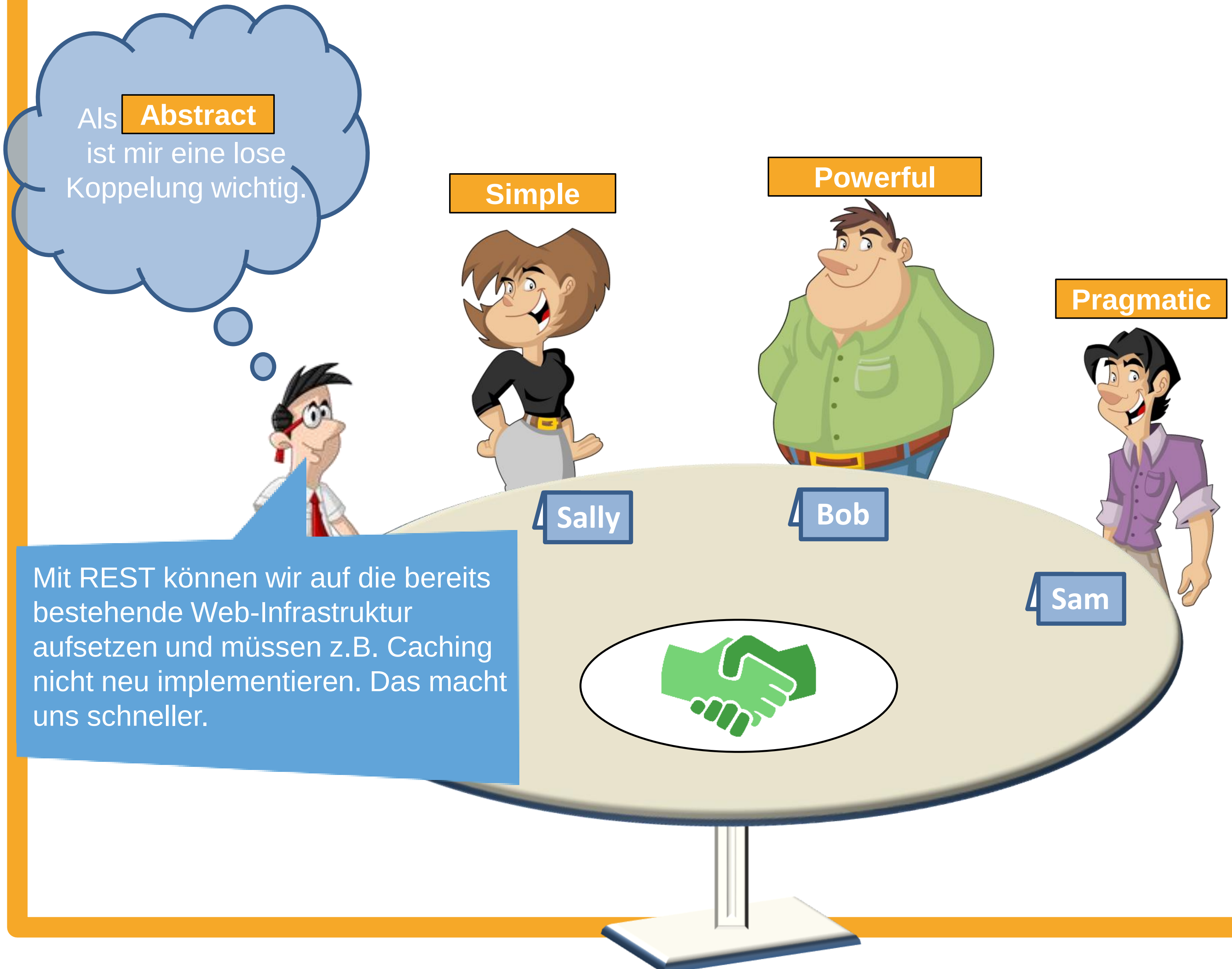
Der nächste Tag…



Technische Diskussion: „Diskutier


Simple
Powerful
Pragmatic
Ja, aber das ist doch viel zu kompliziert! In einem halben Jahr will und muss ich das auch noch verstehen.
Simple
In einem halben Jahr kommt bestimmt eine neue Anforderung und wir müssen die Schnittstelle erweitern...
Powerful
Lasst uns bitte die Deadline nicht vergessen. Wir sollten uns nicht verzetteln!
Pragmatic
Sally
Bob
Jack
Sam

Technische Diskussion: „Diskutieren mit Kontext“



Als Abstract ist mir eine lose Koppelung wichtig.

Mit REST können wir auf die bereits bestehende Web-Infrastruktur aufsetzen und müssen z.B. Caching nicht neu implementieren. Das macht uns schneller.

Was nehmen wir mit?

- Gute und nachvollziehbare Argumente
  - Design Cards

- Gegenseitiges Verständnis für unterschiedliche Positionen
  - Design Types

- Um Blockaden oder Patt-Situation auflösen zu können, benötigt man Exitstrategien
  - Moderationskarten

Gibt es noch mehr?

Design Matrix

Description of design challenge

Name:

Topic overview & Solution details:

If useful, link relevant documents

Result of design decision

Date:

Status: □ Approved □ Rejected

Decided by:

Stakeholder:

Summary:

Simple

- □ Is the solution easy to understand (even in the future)? Is there a solution that is easier?
- □ Does it avoid „clever“ magic and overly generic approaches?
- □
- □

notes

notes

Powerful

- □ Is the solution foresighted enough?
- □ Does it take non-functional requirements into account?
- □ Is the solution generic and reusable?

Ziel: Vorbereitete Design-Entscheidungen aus verschiedenen Perspektiven beleuchten.

Abstract

- □
- □ on its own?
- □ Are modules cohesive and is coupling low?

Concrete

- the same breath?
- □ Can the solution grow naturally over time? (e.g. allow further changes/refactorings)

Pragmatic

- □ Does the solution provide value early on?
- □ Does the solution really address the customer‘s goals/use cases?
- □ Does the solution really fit to the timeline?
- □ Can we use already existing Code (snippets, libraries, services)?

notes

notes

Idealistic

- □ Is this the right solution?
- □ Is it consistent with the rest of the system?
- □ Is ensured that there are no workarounds or bad decisions that will produce serious problems later?

Robust

- □ Is the solution hard to misuse?
- □ Are the chances for something to go wrong minimized?
- □ Are standards used and adhered to?
- □ Are used technologies/libraries stable?
- □ Do all involved people have the necessary knowledge?

notes

notes

Technologic

- □ Is there already an existing technology or library that helps us?
- □ Is the solution state-of-the-art?
- □ Is the solution a technologic progress?
- □ Can we get rid of legacy code?

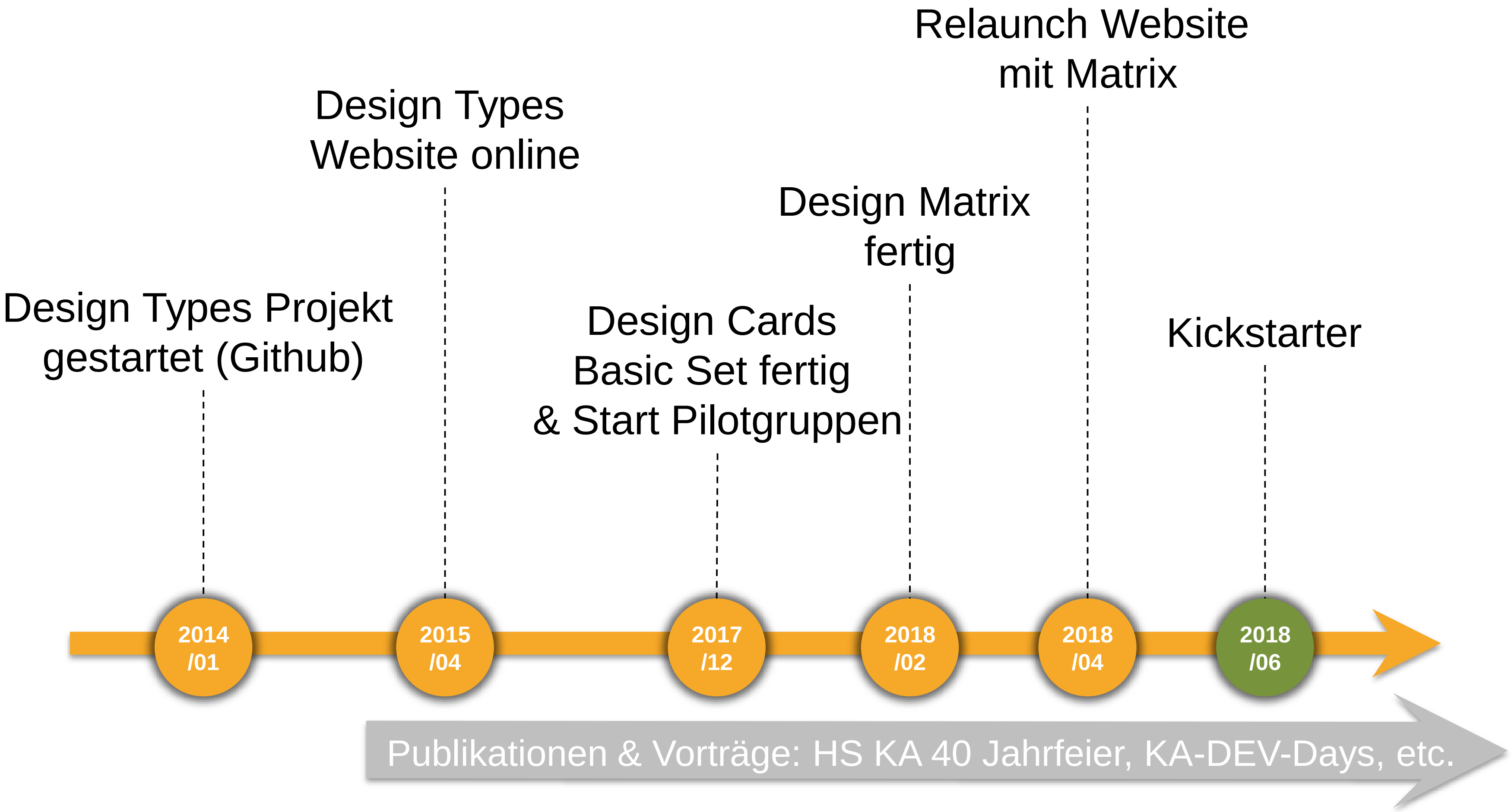
Stand der Dinge

- ■ Design Types
  - □ Fertig
  - □ > 2500 Teilnehmer bisher

- ■ Design Matrix
  - □ Fertig
  - □ Aktuell Feedback durch Pilotgruppen
  - □ Verfügbar als Download

- ■ Design Cards
  - □ 26/54 Karten fertig (Basic Set)
  - □ Online-Karten gerade im Entstehen
  - □ Aktuell Feedback durch Pilotgruppen

Timeline – was passiert(e)?



2014 /01 – Design Types Projekt gestartet (Github)

2015 /04 – Design Types Website online

2017 /12 – Design Cards Basic Set fertig & Start Pilotgruppen

2018 /02 – Design Matrix fertig

2018 /04 – Relaunch Website mit Matrix

2018 /06 – Kickstarter

Publikationen & Vorträge: HS KA 40 Jahrfeier, KA-DEV-Days, etc.

Thank you for your interest…
…the „Software Design Knights“

Matthias Wittum

Christian Rehn

Any Questions?

Contact

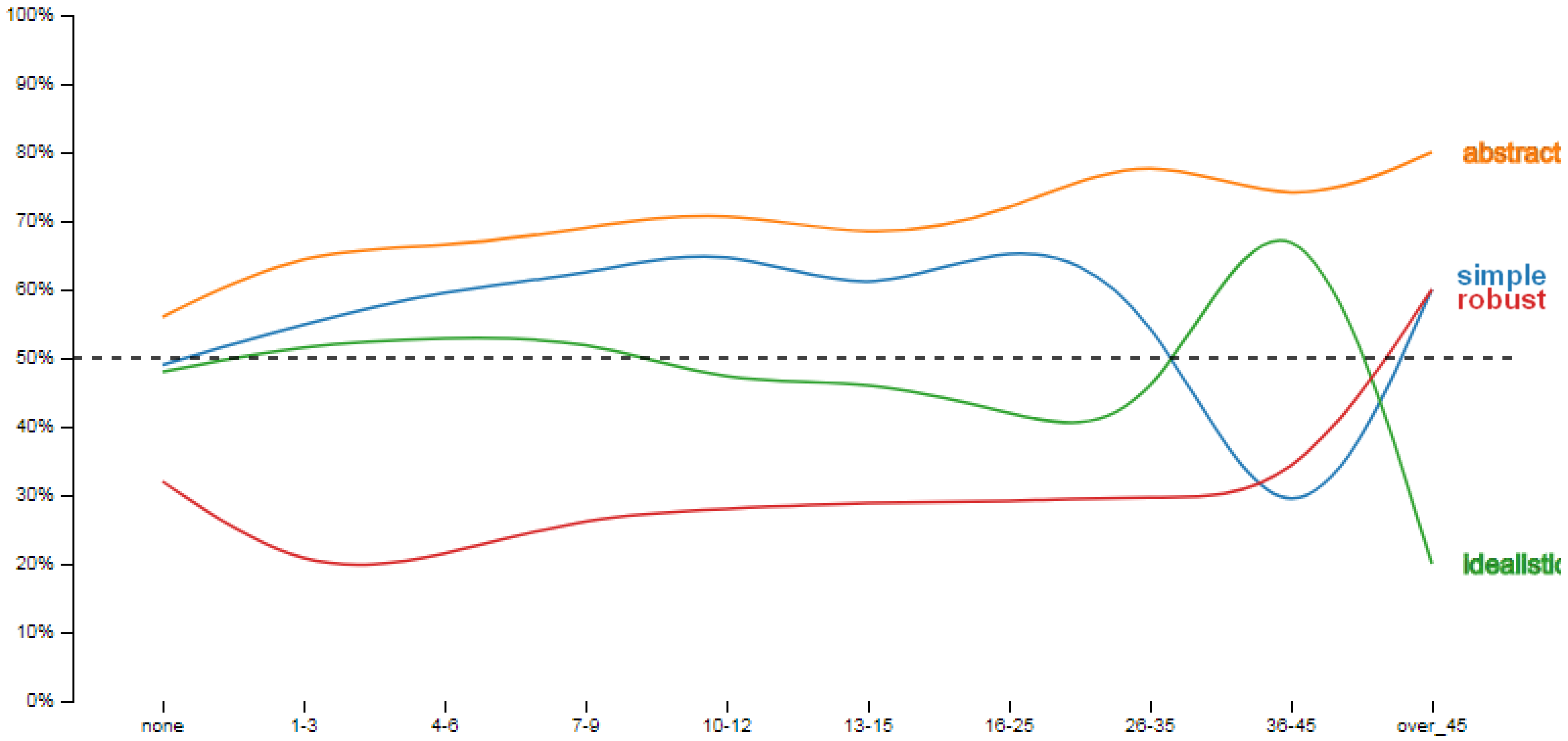
Web: www.design-types.net

Mail: email@design-types.net

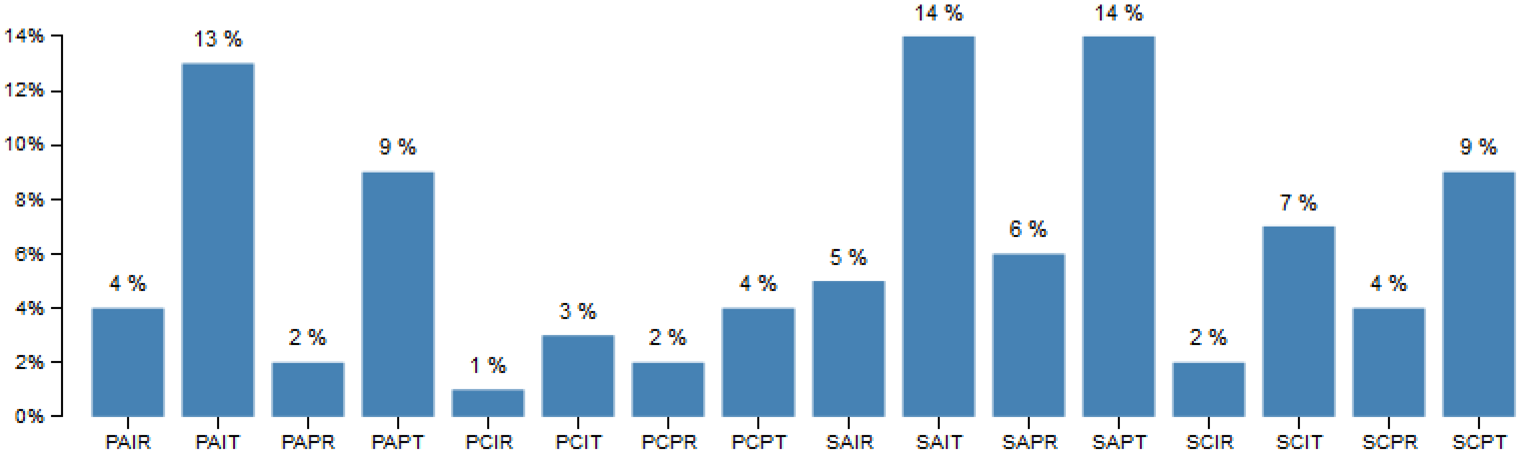
Twitter: @SWDesignKnights

Anhang

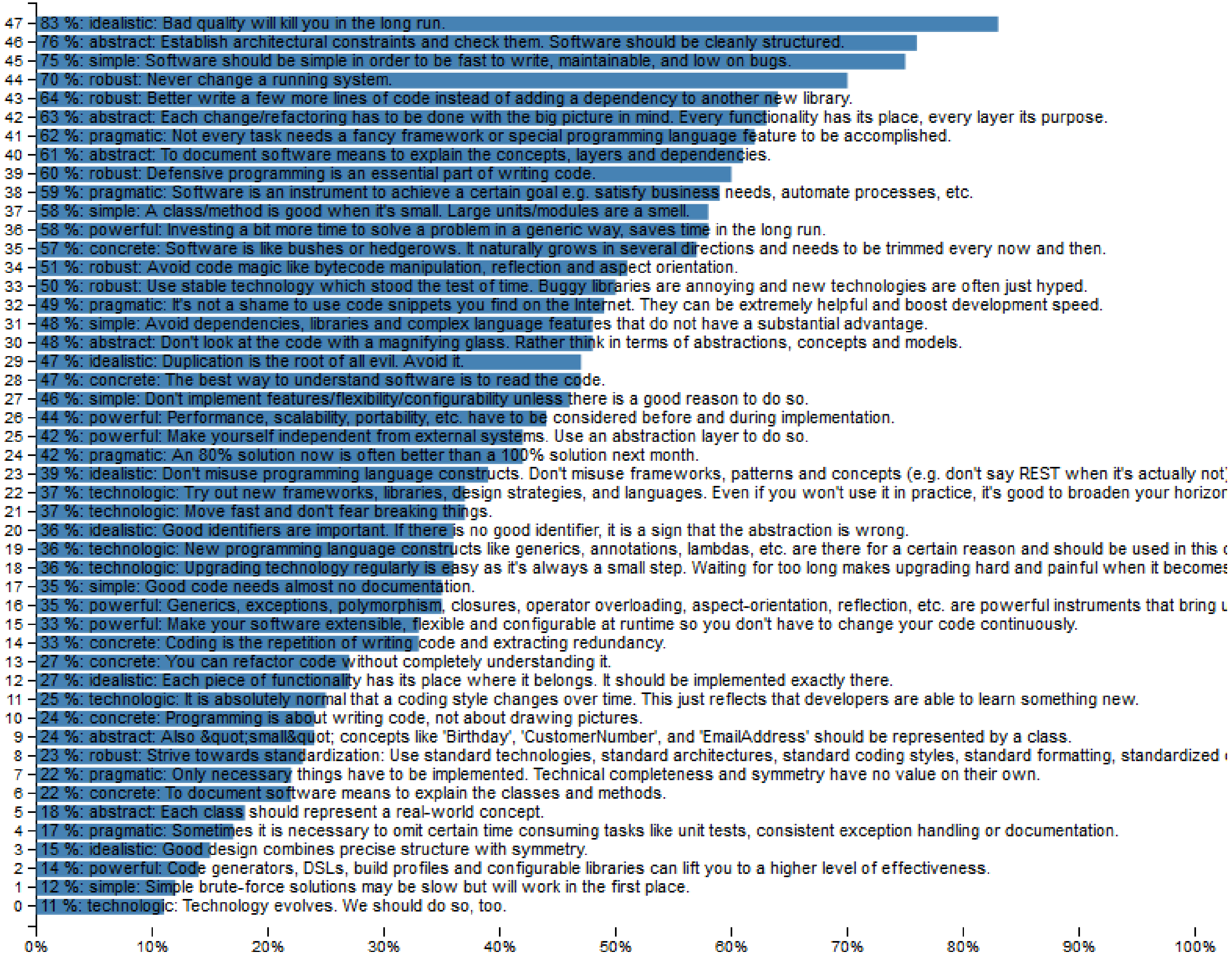
Statistiken – Veränderung durch Berufserfahrung



Statistiken – Verteilung der Typen



Statistiken – Häufigkeit der Antworten


47 83 %: idealistic: Bad quality will kill you in the long run.
46 76 %: abstract: Establish architectural constraints and check them. Software should be cleanly structured.
45 75 %: simple: Software should be simple in order to be fast to write, maintainable, and low on bugs.
44 70 %: robust: Never change a running system.
43 64 %: robust: Better write a few more lines of code instead of adding a dependency to another new library.
42 63 %: abstract: Each change/refactoring has to be done with the big picture in mind. Every functionality has its place, every layer its purpose.
41 62 %: pragmatic: Not every task needs a fancy framework or special programming language feature to be accomplished.
40 61 %: abstract: To document software means to explain the concepts, layers and dependencies.
39 60 %: robust: Defensive programming is an essential part of writing code.
38 59 %: pragmatic: Software is an instrument to achieve a certain goal e.g. satisfy business needs, automate processes, etc.
37 58 %: simple: A class/method is good when it's small. Large units/modules are a smell.
36 58 %: powerful: Investing a bit more time to solve a problem in a generic way, saves time in the long run.
35 57 %: concrete: Software is like bushes or hedgerows. It naturally grows in several directions and needs to be trimmed every now and then.
34 51 %: robust: Avoid code magic like bytecode manipulation, reflection and aspect orientation.
33 50 %: robust: Use stable technology which stood the test of time. Buggy libraries are annoying and new technologies are often just hyped.
32 49 %: pragmatic: It's not a shame to use code snippets you find on the Internet. They can be extremely helpful and boost development speed.
31 48 %: simple: Avoid dependencies, libraries and complex language features that do not have a substantial advantage.
30 48 %: abstract: Don't look at the code with a magnifying glass. Rather think in terms of abstractions, concepts and models.
29 47 %: idealistic: Duplication is the root of all evil. Avoid it.
28 47 %: concrete: The best way to understand software is to read the code.
27 46 %: simple: Don't implement features/flexibility/configurability unless there is a good reason to do so.
26 44 %: powerful: Performance, scalability, portability, etc. have to be considered before and during implementation.
25 42 %: powerful: Make yourself independent from external systems. Use an abstraction layer to do so.
24 42 %: pragmatic: An 80% solution now is often better than a 100% solution next month.
23 39 %: idealistic: Don't misuse programming language constructs. Don't misuse frameworks, patterns and concepts (e.g. don't say REST when it's actually not
22 37 %: technologic: Try out new frameworks, libraries, design strategies, and languages. Even if you won't use it in practice, it's good to broaden your horizo
21 37 %: technologic: Move fast and don't fear breaking things.
20 36 %: idealistic: Good identifiers are important. If there is no good identifier, it is a sign that the abstraction is wrong.
19 36 %: technologic: New programming language constructs like generics, annotations, lambdas, etc. are there for a certain reason and should be used in this
18 36 %: technologic: Upgrading technology regularly is easy as it's always a small step. Waiting for too long makes upgrading hard and painful when it becomes
17 35 %: simple: Good code needs almost no documentation.
16 35 %: powerful: Generics, exceptions, polymorphism, closures, operator overloading, aspect-orientation, reflection, etc. are powerful instruments that bring
15 33 %: powerful: Make your software extensible, flexible and configurable at runtime so you don't have to change your code continuously.
14 33 %: concrete: Coding is the repetition of writing code and extracting redundancy.
13 27 %: concrete: You can refactor code without completely understanding it.
12 27 %: idealistic: Each piece of functionality has its place where it belongs. It should be implemented exactly there.
11 25 %: technologic: It is absolutely normal that a coding style changes over time. This just reflects that developers are able to learn something new.
10 24 %: concrete: Programming is about writing code, not about drawing pictures.
9 24 %: abstract: Also &quot;small&quot; concepts like 'Birthday', 'CustomerNumber', and 'EmailAddress' should be represented by a class.
8 23 %: robust: Strive towards standardization: Use standard technologies, standard architectures, standard coding styles, standard formatting, standardized
7 22 %: pragmatic: Only necessary things have to be implemented. Technical completeness and symmetry have no value on their own.
6 22 %: concrete: To document software means to explain the classes and methods.
5 18 %: abstract: Each class should represent a real-world concept.
4 17 %: pragmatic: Sometimes it is necessary to omit certain time consuming tasks like unit tests, consistent exception handling or documentation.
3 15 %: idealistic: Good design combines precise structure with symmetry.
2 14 %: powerful: Code generators, DSLs, build profiles and configurable libraries can lift you to a higher level of effectiveness.
1 12 %: simple: Simple brute-force solutions may be slow but will work in the first place.
0 11 %: technologic: Technology evolves. We should do so, too.