31.8.–3.9.2015
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Built to last

Prinzipien für nachhaltige Software-Architekturen

Frank Pientka

MATERNA GmbH

MATERNA
Information & Communications

Built To Last
Nachhaltige Software-Entwicklung

Wer ist Frank Pientka?

Dipl.-Informatiker (TH Karlsruhe)

Principal Software Architect in Dortmund

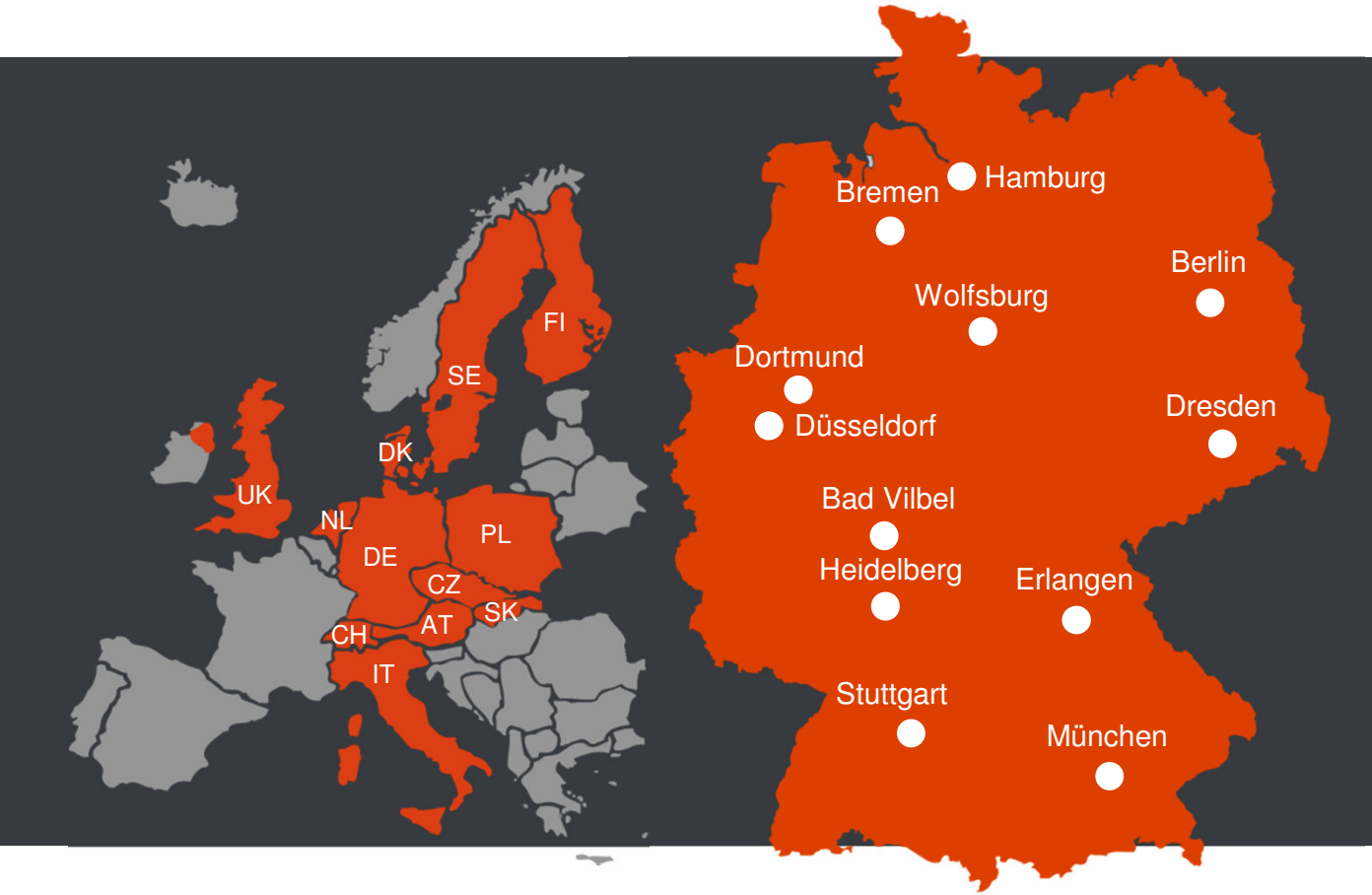
iSAQB-Gründungsmitglied

heise.de/developer/Federlesen-Kolumne

Über 20 Jahre IT-Erfahrung
Veröffentlichungen und Vorträge

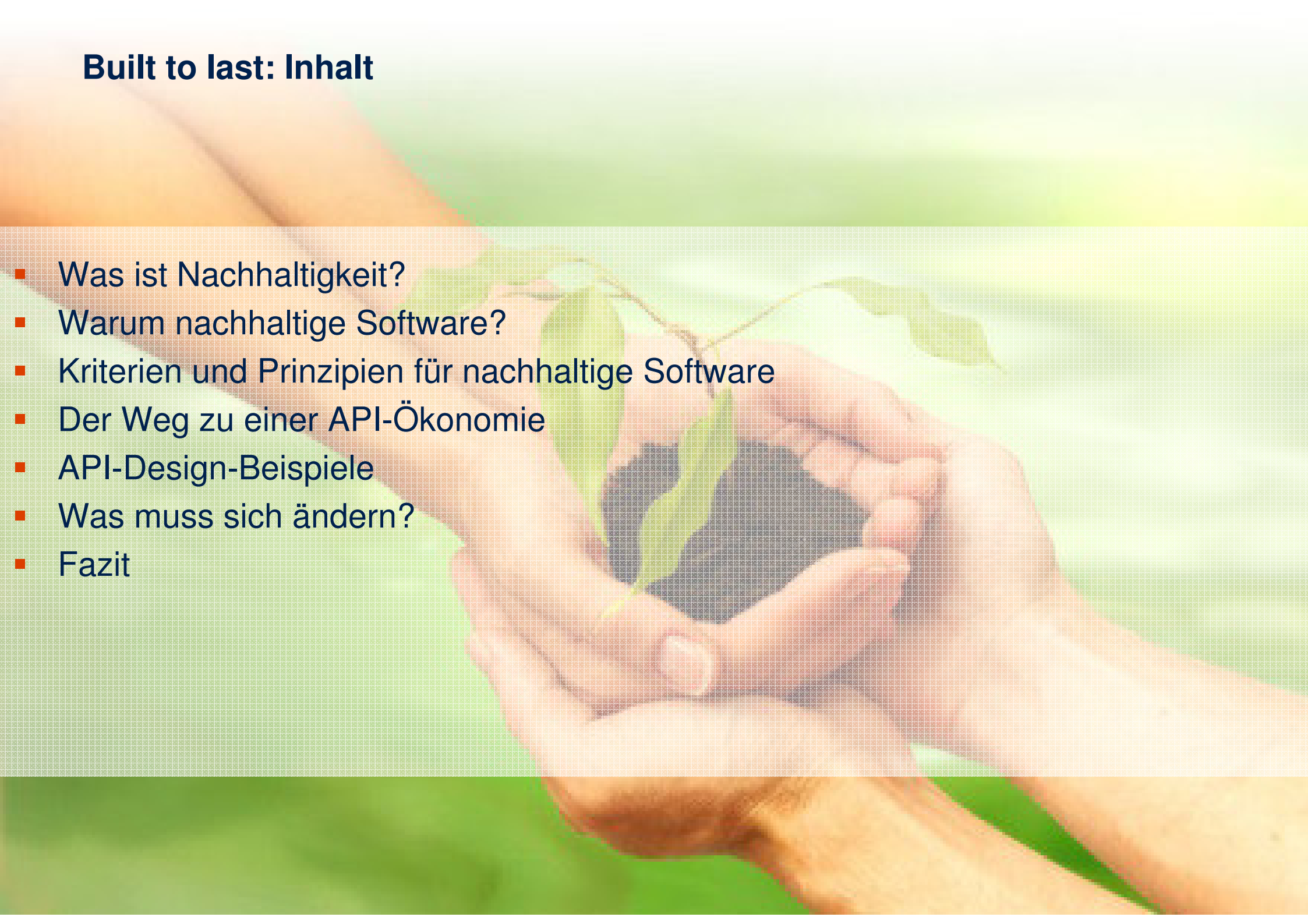
Frank Pientka, Dipl.-Informatiker
frank.pientka@materna.de
+49 (231) 5599 8854
+49 (1570) 1128854
www.materna.de

Wer wir sind.



- Gründung: 1980
  Mitarbeiter: 1.500
  Umsatz 2014: 192 Mio. €
- Inhabergeführtes Familienunternehmen der ITK-Branche
- Full-Service-Dienstleister im Premium-Segment
- Zielgruppe: IT-Organisationen und Fachabteilungen in Privatwirtschaft und Behörden

Built to last: Inhalt



- Was ist Nachhaltigkeit?
- Warum nachhaltige Software?
- Kriterien und Prinzipien für nachhaltige Software
- Der Weg zu einer API-Ökonomie
- API-Design-Beispiele
- Was muss sich ändern?
- Fazit

Gebaut für die Ewigkeit: Was ist eine nachhaltige Architektur?

Die meiste Zeit geht dadurch verloren, dass man nicht zu Ende denkt.
Alfred Herrhausen

Planen oder evolutionär entwickeln – Alt trifft Neu

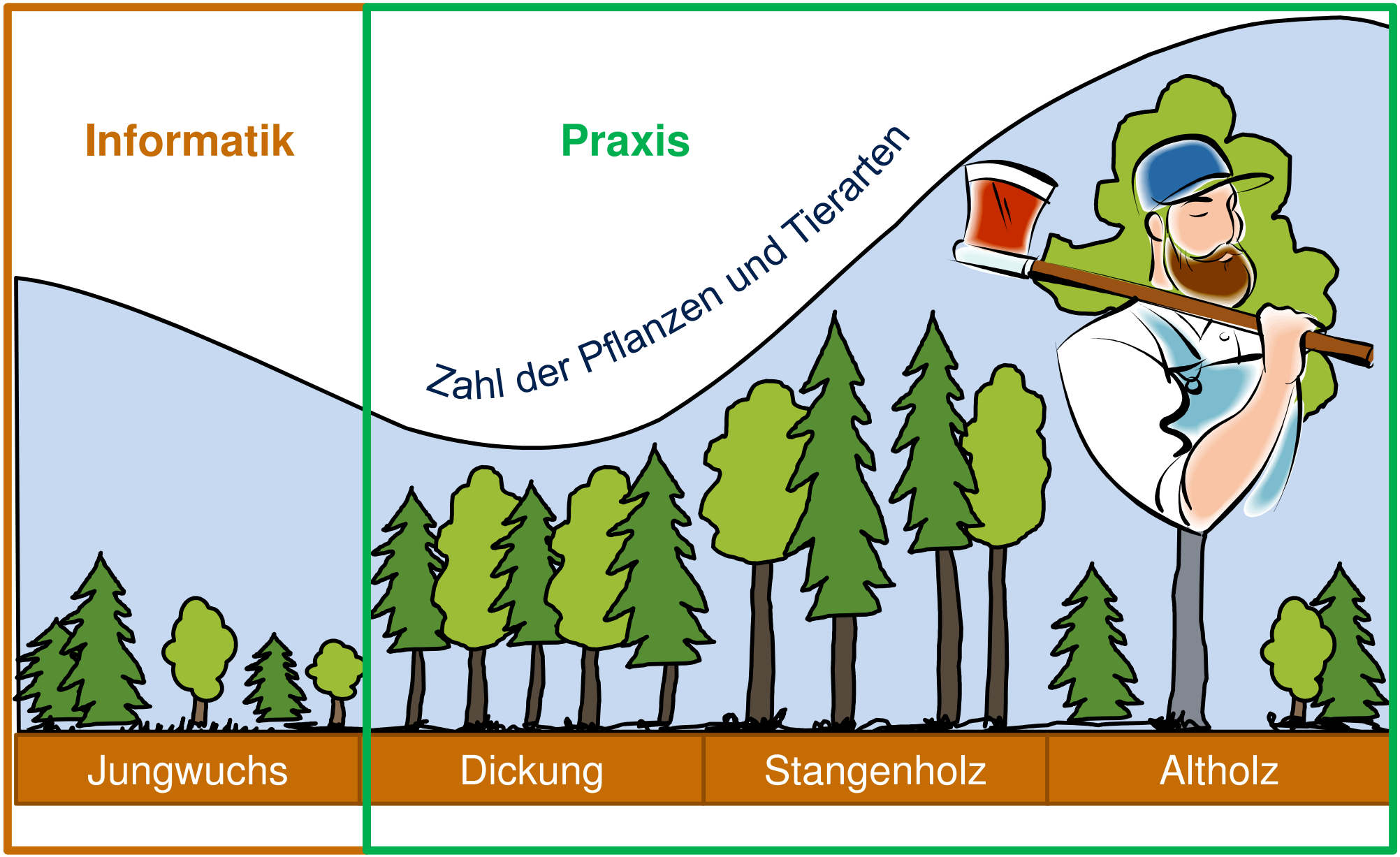
Design-by-Songtext

Gib mir die Hand,
ich bau Dir
ein Schloss aus Sand
Irgendwie
Irgendwo
Irgendwann

(Nena 1984)

PIXELIO-ID: 468044

Schema der nachhaltigen Entwicklung seit über 300 Jahren



Digitale Nachhaltigkeit Aspekte.

*“development that meets the **needs of the present** without compromising the **ability of future** generations”*

*(*Brundtland UN-Report 1987*)*

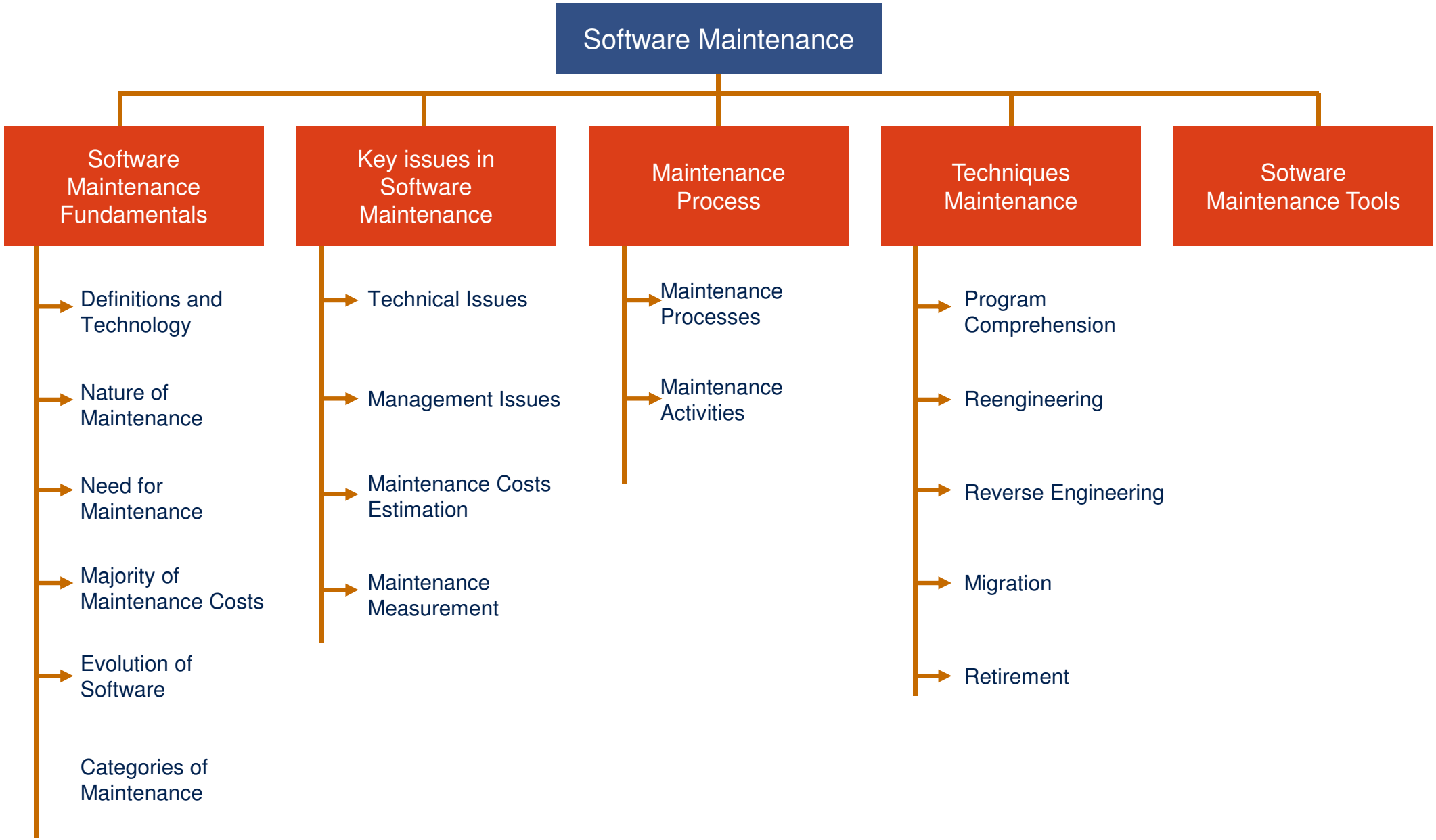
http://www.computer.org/portal/web/swebok 3.0 (2014)

1. Software Requirements
2. Software Design
3. Software Construction
4. Software Testing

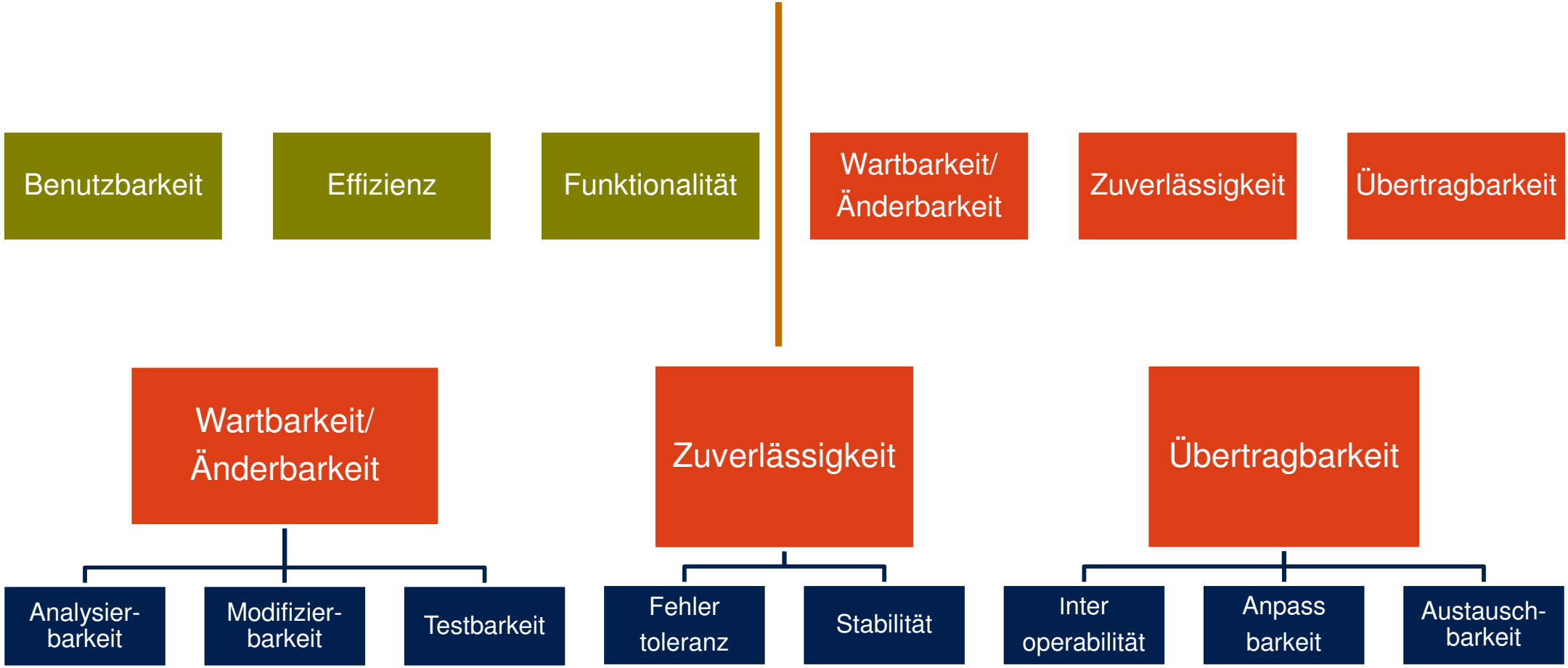
---

5. **Software Maintenance**
6. Software Configuration Management
7. Software Engineering Management
8. **Software Engineering Process**
9. Software Engineering Models and Methods
10. **Software Quality**
11. Software Engineering Professional Practice
12. **Software Engineering Economics**

Software Maintenance SWEBOK Guide V3.0



ISO 25010 für Softwarequalität



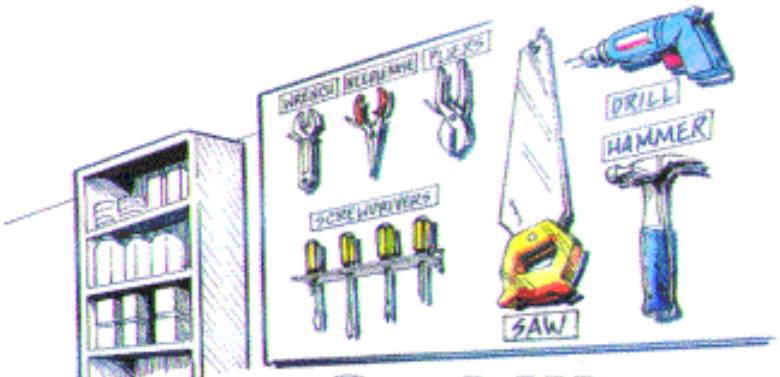
IEEE Standard Glossary of Software Engineering Terminology 610.12-1990

"**Software engineering** is the application of a *systematic, disciplined*, *quantifiable* approach to the **development**, **operation**, and **maintenance** of software"

5 Säulen der Total Productive Maintenance (1950 Japan)

1.
2.
3.
4.
5.

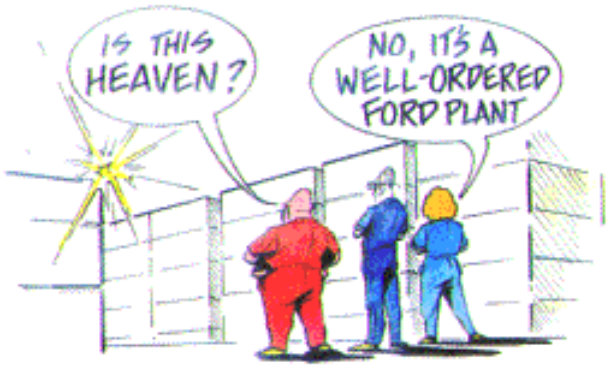
1. Sort



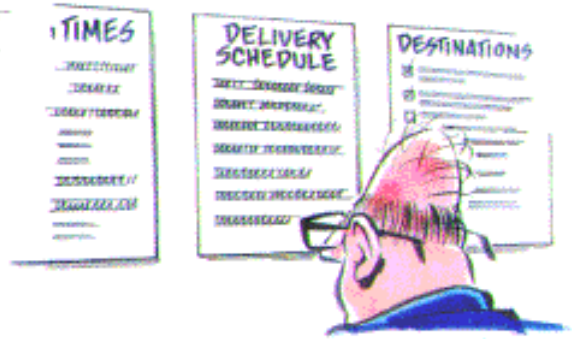
2. Stabilize

5 S's

3. Shine

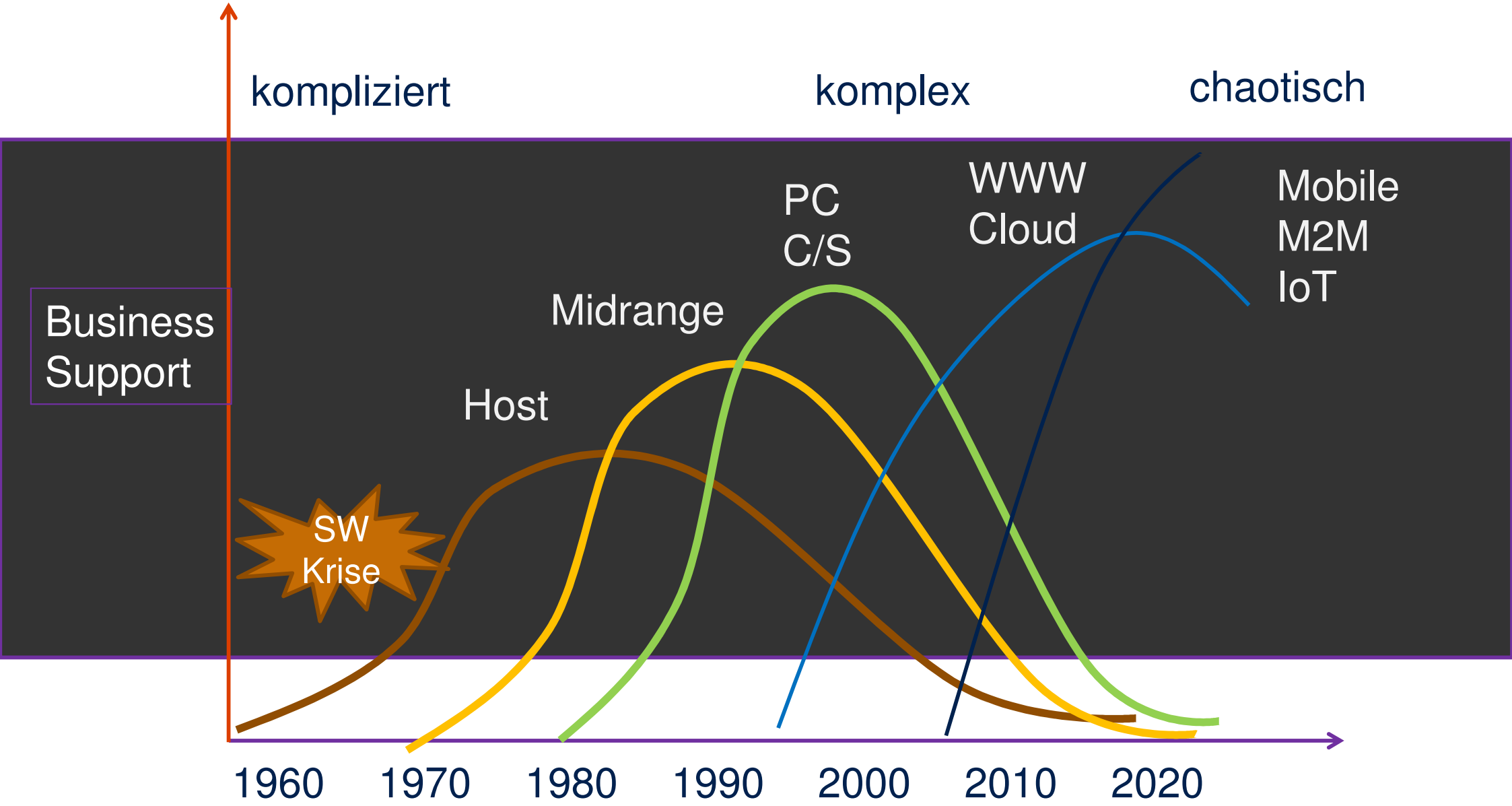


5. Sustain

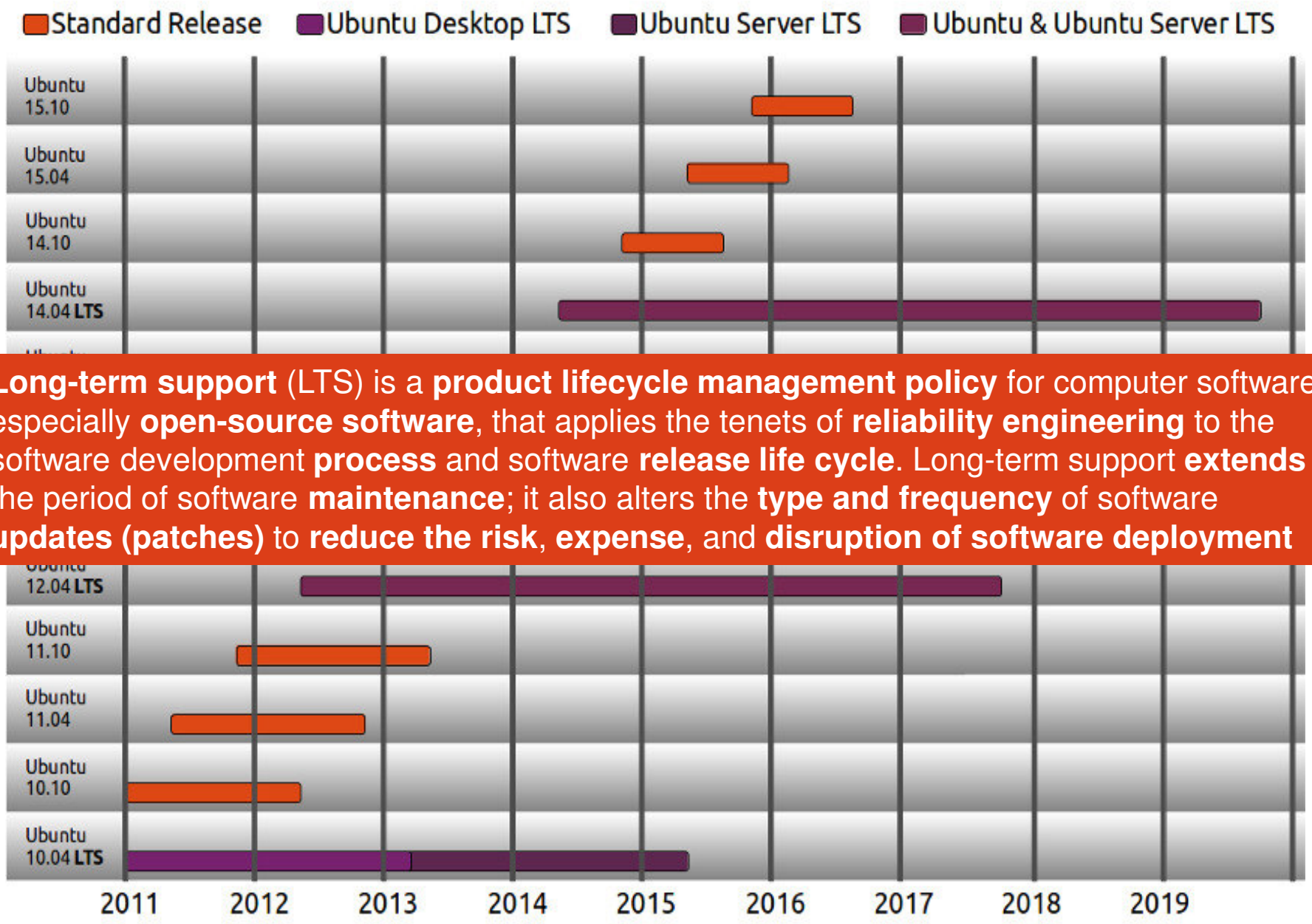


4. Standardize

Die Entwicklung der IT



http://en.wikipedia.org/wiki/Long-term_support



Long-term support (LTS) is a **product lifecycle management policy** for computer software, especially **open-source software**, that applies the tenets of **reliability engineering** to the software development **process** and software **release life cycle**. Long-term support **extends** the period of software **maintenance**; it also alters the **type and frequency** of software **updates (patches)** to **reduce the risk**, **expense**, and **disruption of software deployment**

http://wiki.eclipse.org/LTS/LTS_Ready

A **build** that builds on **Eclipse Foundation** hardware and

- Can be cloned/checked out with **one step**
- Is **documented**
- Is **version controlled**
- Is **automated**
- Is **deterministic** given the same source code and third party libraries
- Is easily **reproducible** on **suitably-configured systems**
- Can refer to compilers and other tools **from a configurable location**
- Capable of **building without** an active Internet connection
- Capable of pulling **dependencies** from a **known controlled source**

Adheres to Eclipse IP **policies**

Bug Tracking:

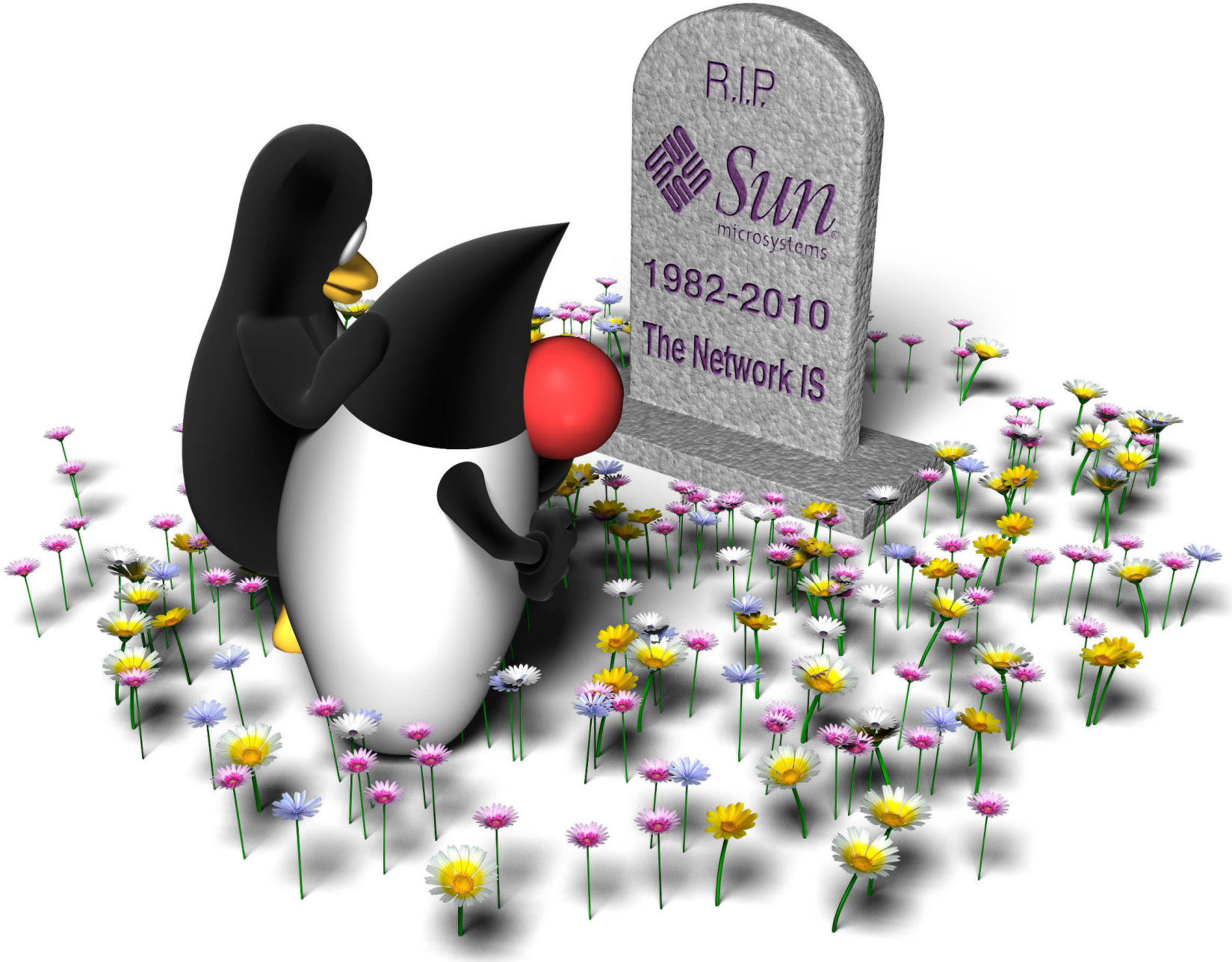
no code change can be released without proper bugzilla entry

Release Management: part of the annual simultaneous release

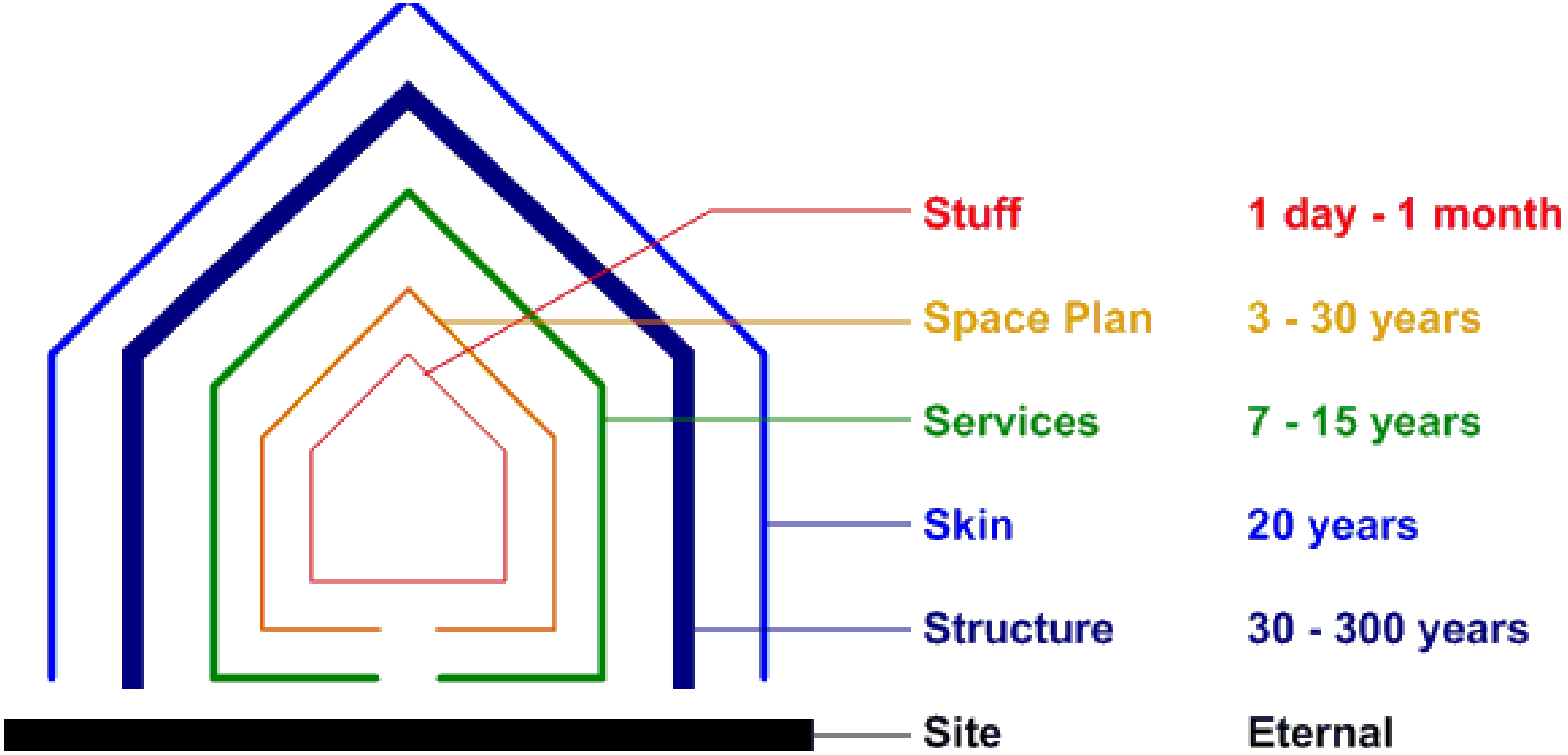
Supply & Demand:

At least **two** LTS IWG member companies **offering support**

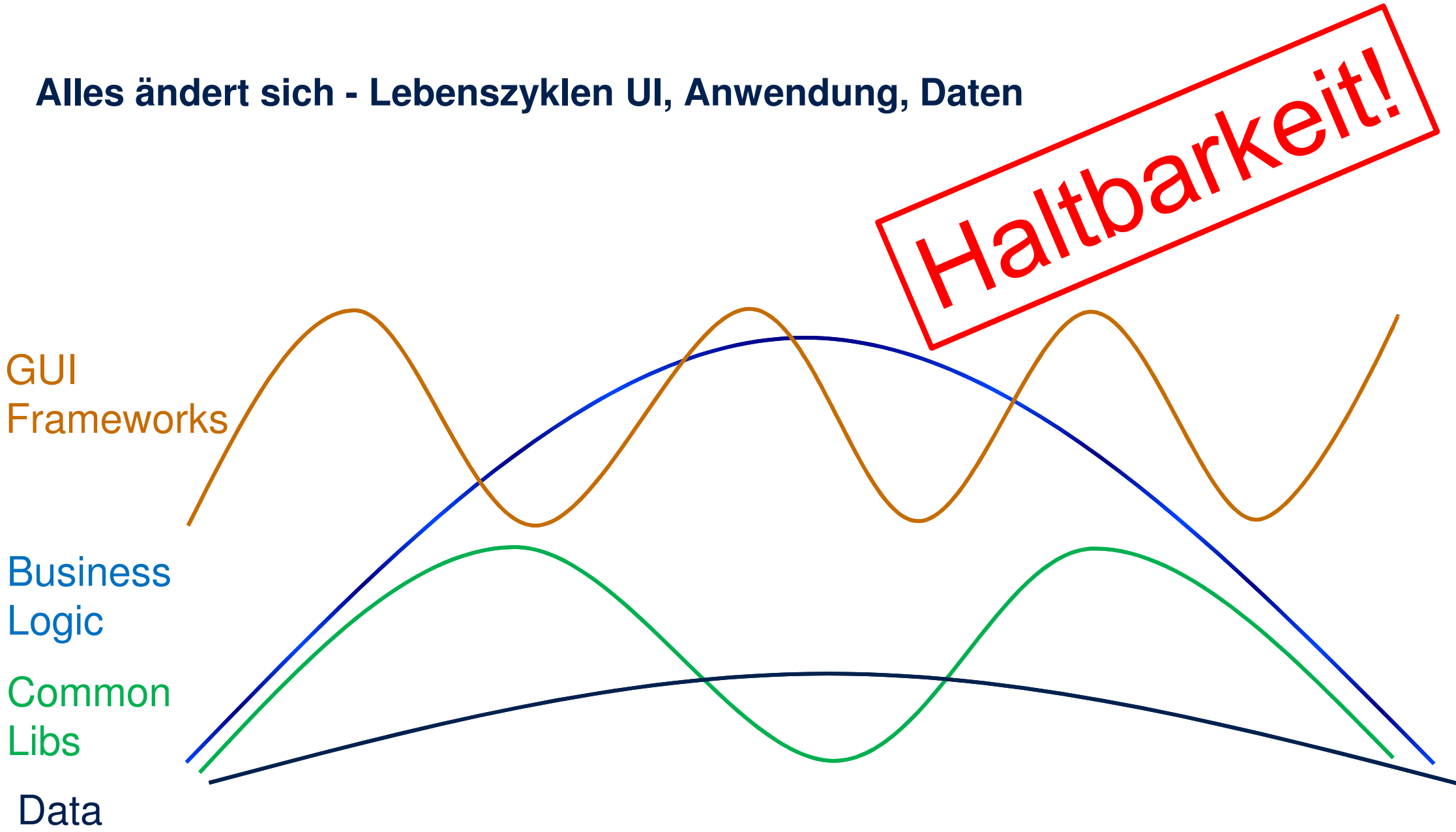
Was wenn Anbieter, Produkte, Partner … verschwinden?



How Buildings Learn: What Happens After They're Built (Stewart Brand 1994)



Alles ändert sich - Lebenszyklen UI, Anwendung, Daten



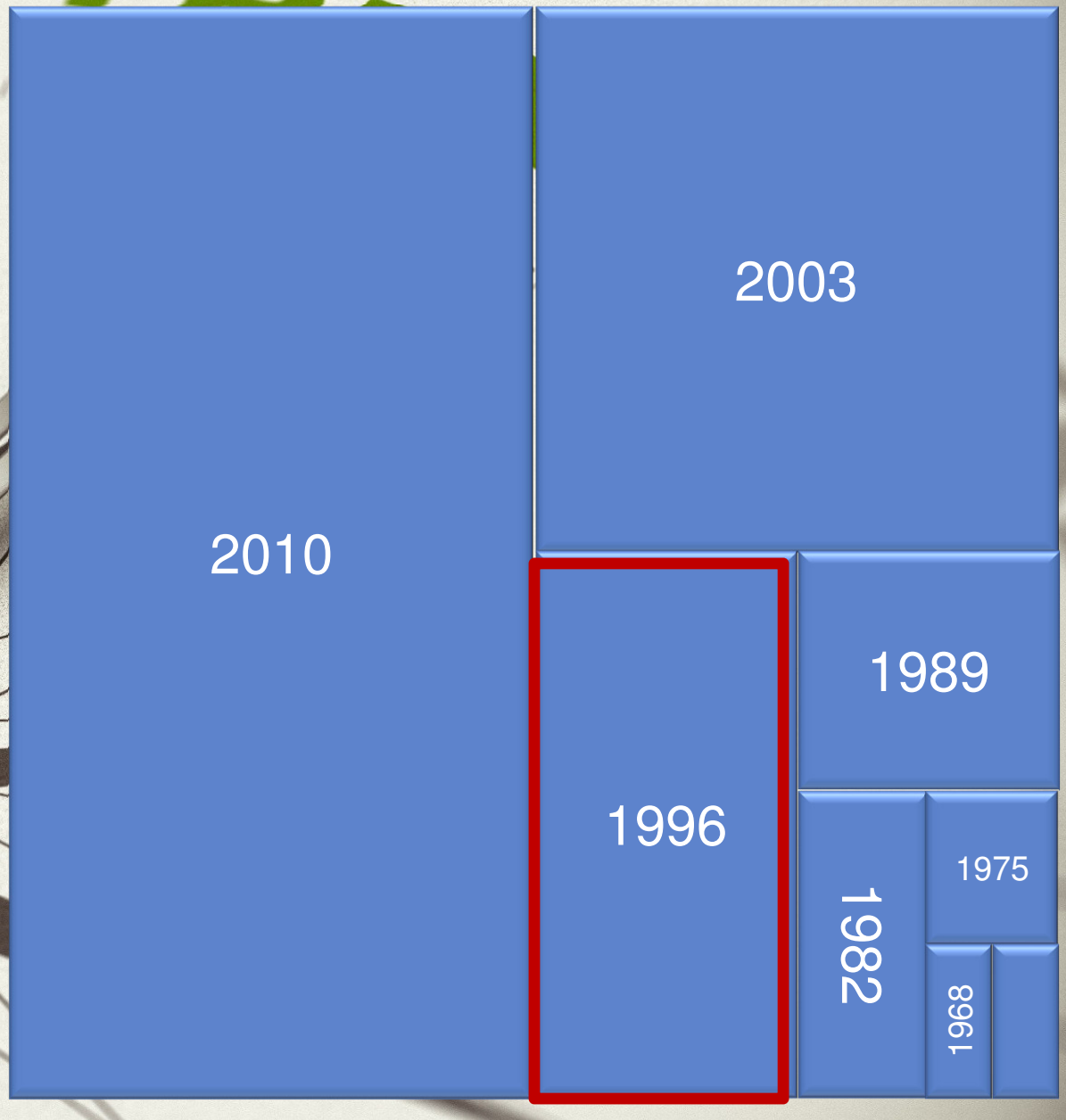
Altbewährt und trotzdem gut

…the most boring technology you can find in use for years and years…

In Defense of Boring, Grady Booch, May/June 2013, IEEE Software

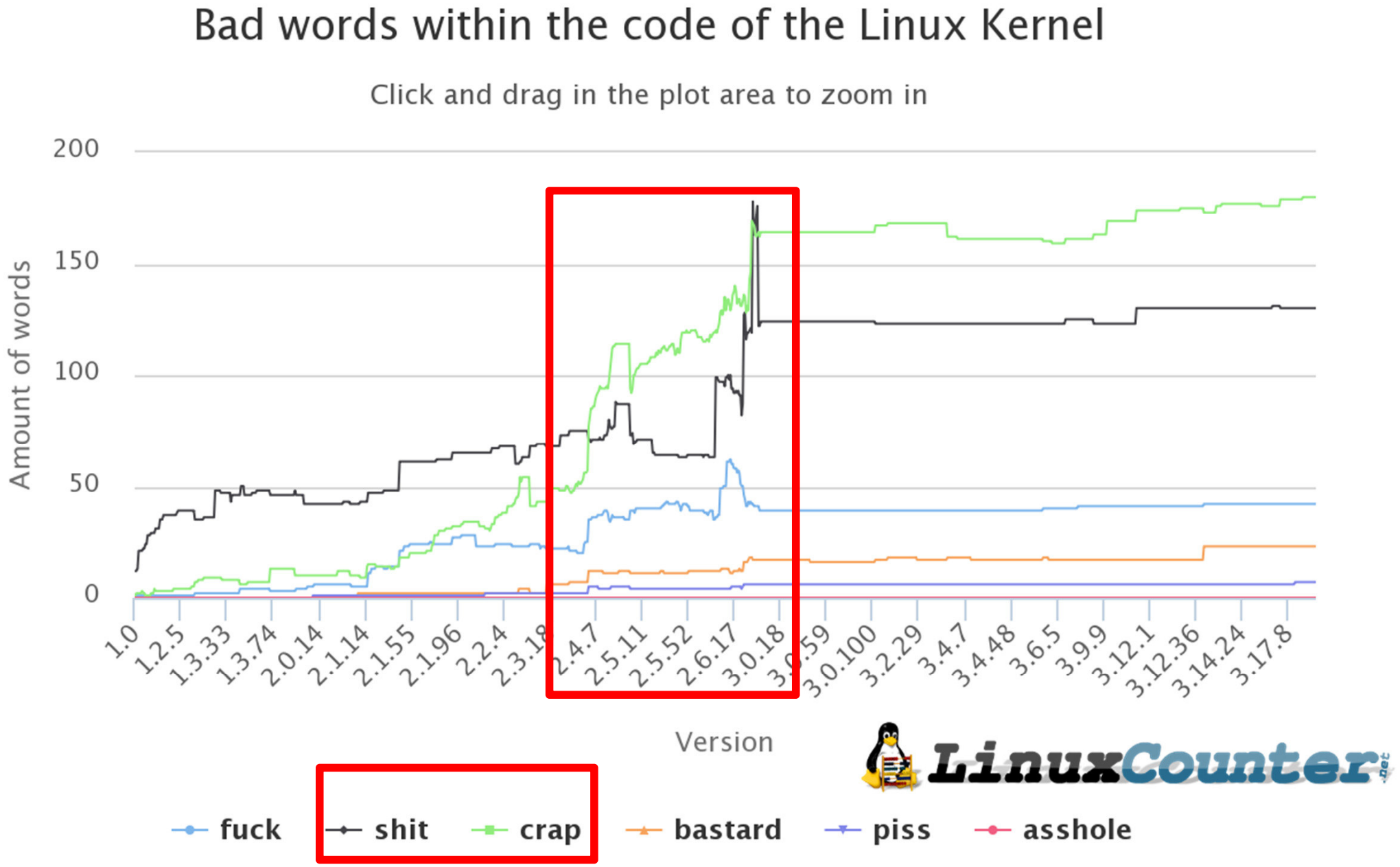
Systemevolution: Wie entwickelt sich Software über die Zeit?

“If I had more time
I would have written less code” ☺

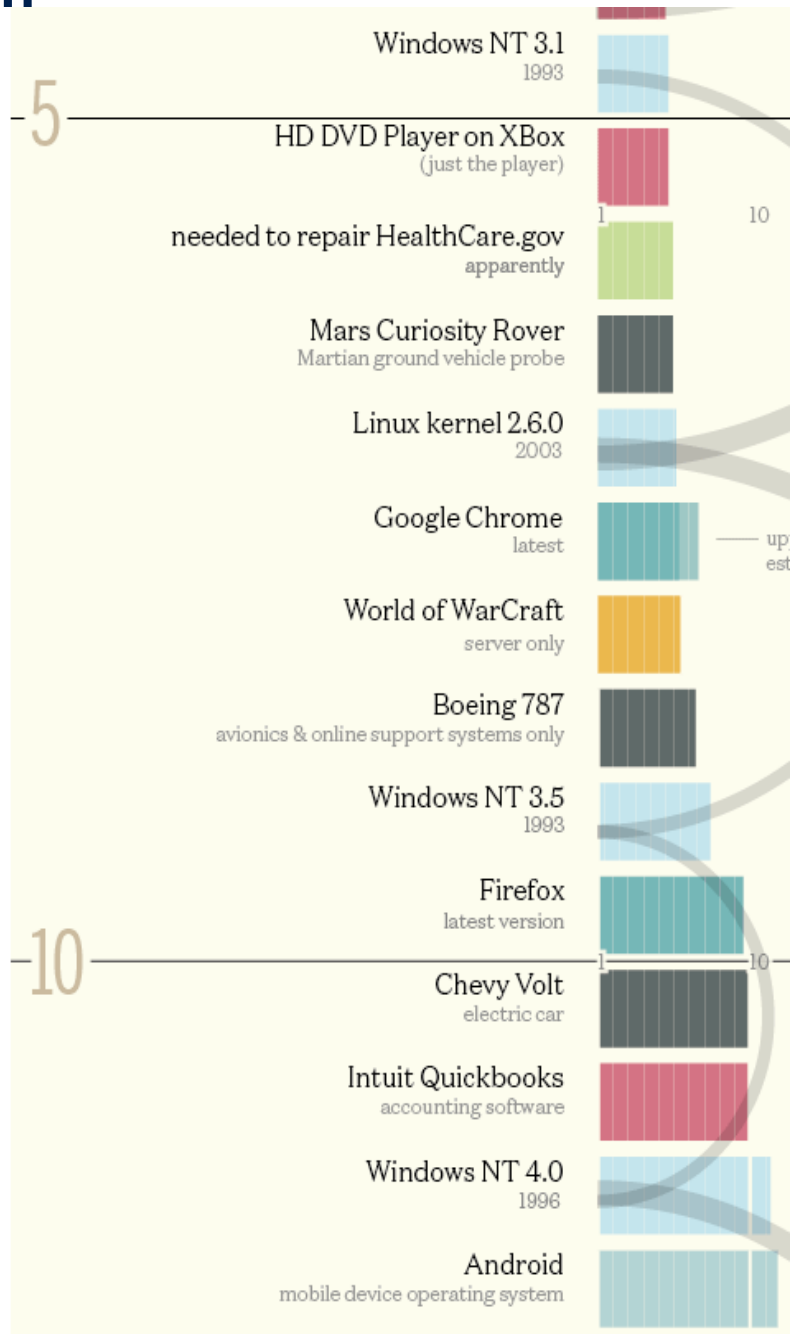


http://users.jyu.fi/~koskinen/smcosts.htm

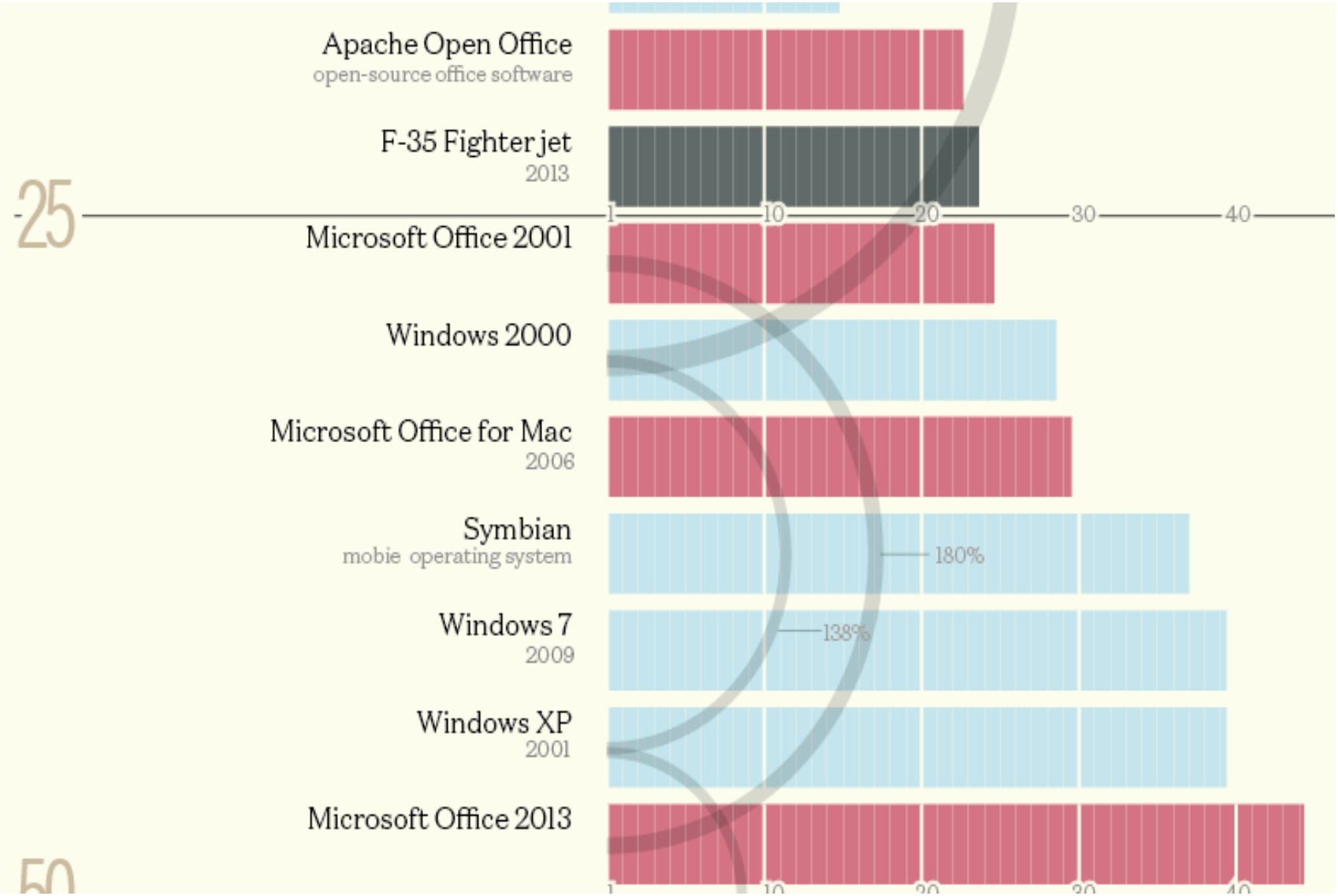
Linux Kernel - Wie entwickelt sich Software über die Zeit?



5 – 10 Millionen Codezeilen

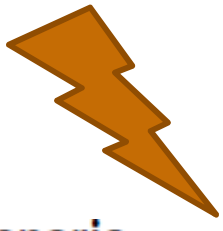


25 Millionen Codezeilen: Office&Windows



Too big to scale !

Qualitätszenarien für nachhaltige Architektur entwickeln (Heiko Koziolek)

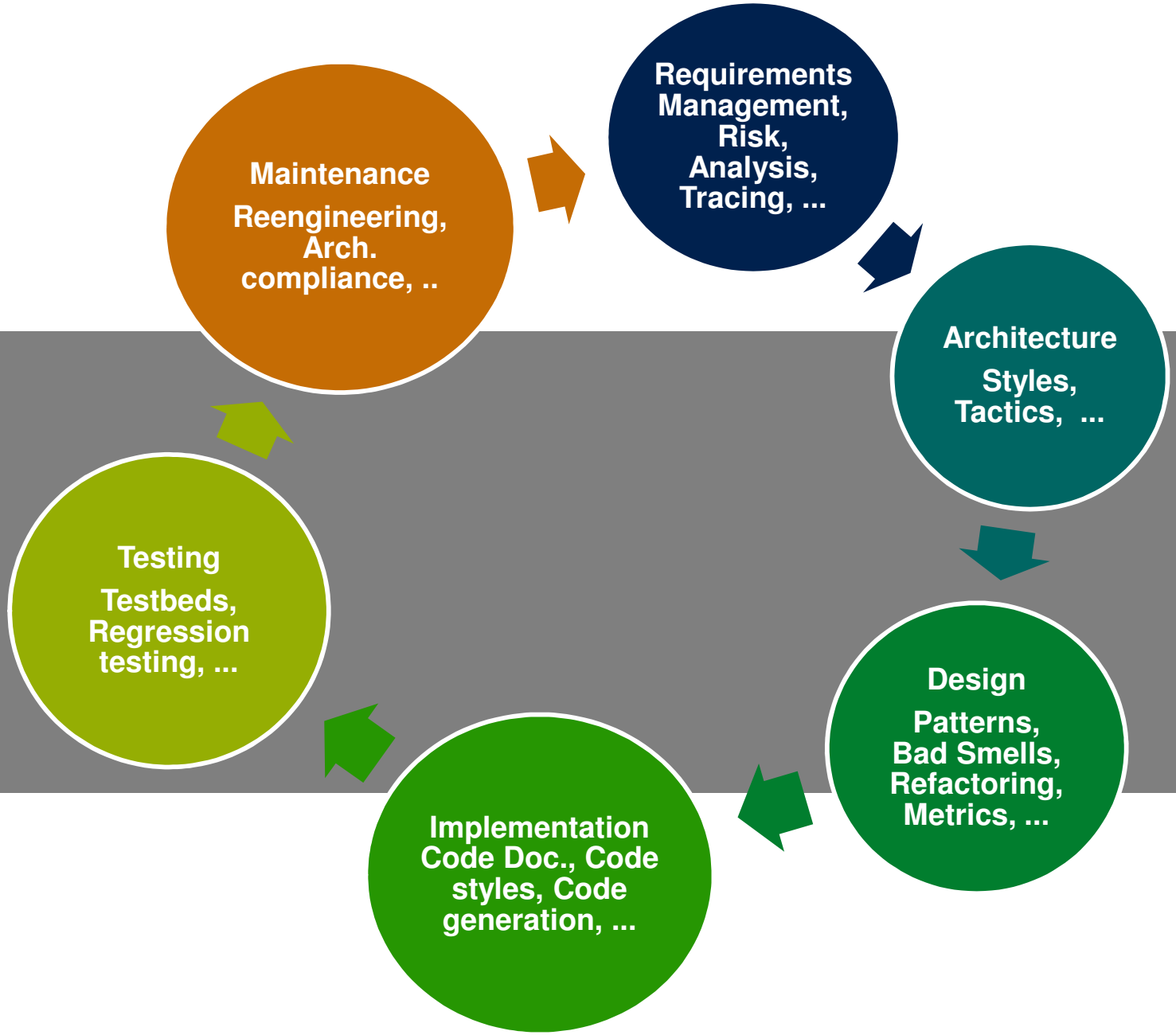


Risikomangement

Evolution Scenario	Assessment 2011	Assessment 2012	Assessment 2013
Change of the deployment platform	Scenario recognized, but no immediate preparation desired	Scenario refined, likelihood increased	Large research project started
Change of a client-facing plug-in interface	Unlikely, but impact would be high	Did not occur, still possible	Did not occur, still possible
Change of the third-party middleware implementation	Medium likelihood, limited impact	Did not occur, still possible	Did not occur, still possible
GUI framework change	Unlikely, but impact would be high	Research project executed	Planned as an extension

Änderungs-/Wachstumsszenarien entwerfen → Entscheidungen validieren, Alternativen betrachten
Kompromisse festhalten

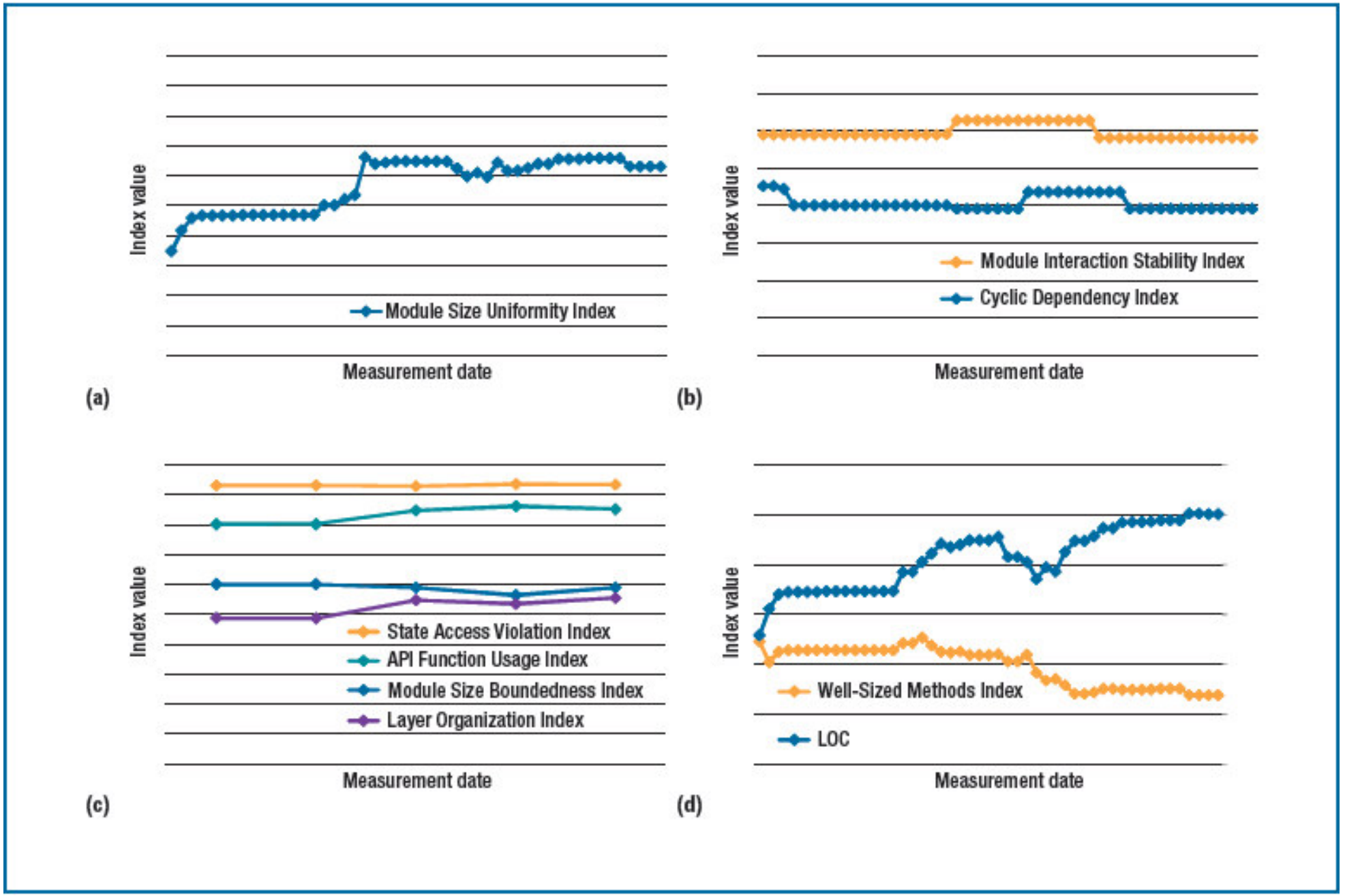
Sustainability Guidelines for Long-Living Software Systems



Phase independent:
Documentation, Knowledge, Management, Process, Improvement, Organizational Structures, ...

Quelle: Heiko Koziolek

Überwachen wichtiger Metriken für nachhaltige Architektur (Heiko Koziolek)



Wie Systeme zerlegen?

„modularization as a mechanism for improving the flexibility of a system“

David Parnas (1972)

Modularität statt Monolithen gefragt

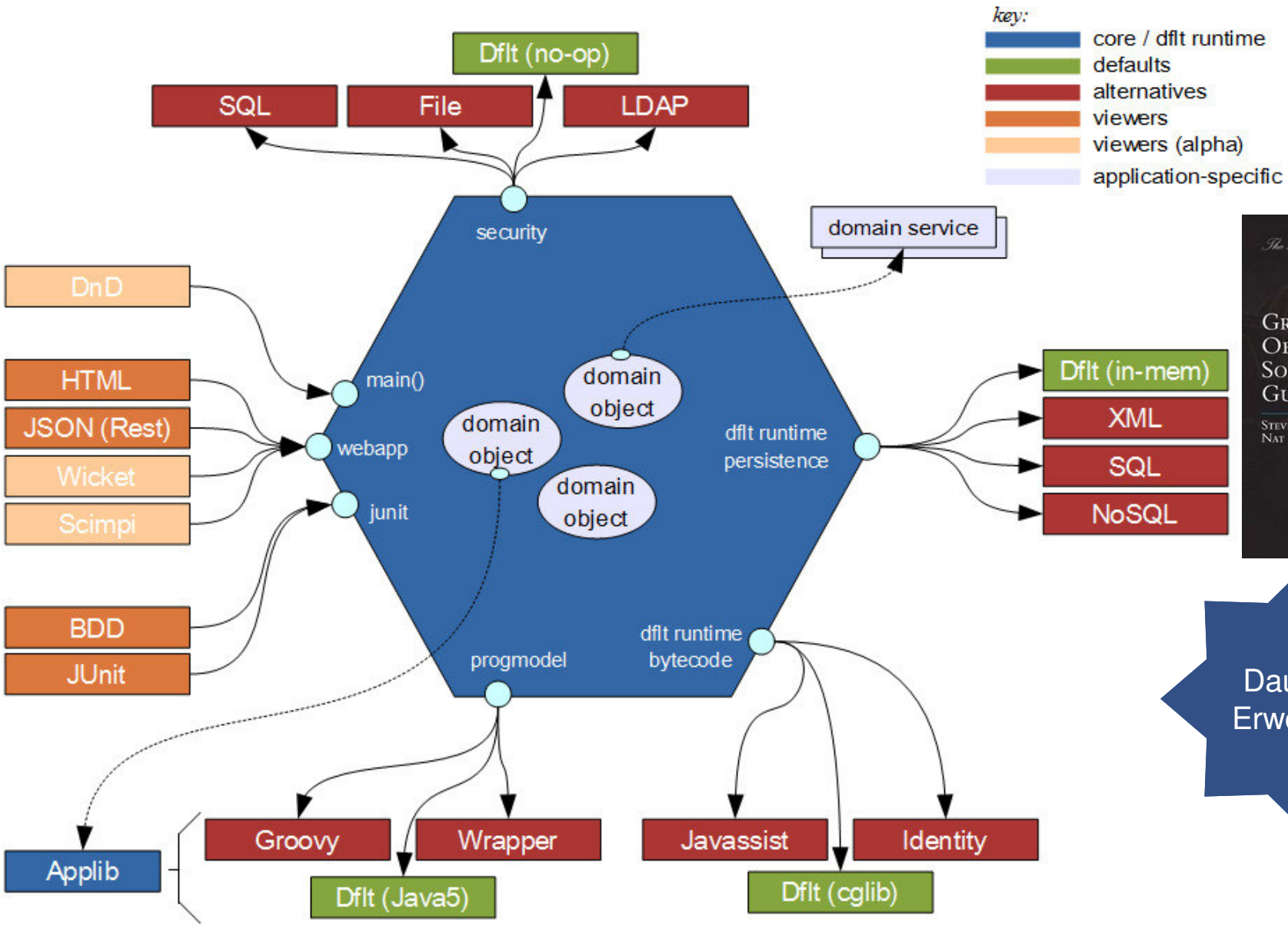
Foto: Ehapa

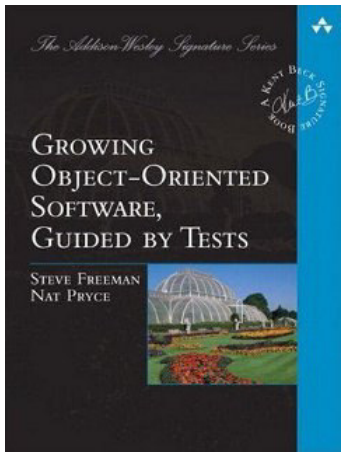
Pasta-Theorie in der Softwareentwicklung: Schichtenarchitektur

Spaghetti-Orientierte Architektur (SOA) vs. Micro Services

Nonoservice is an Anti-pattern where a service is too fine grained.
Nanoservice is a service whose overhead out-weights its utility.
(SOA Patterns, Arnon Rotem-Gal-Oz, 2012)

Trennung von Fachlichkeit & Technik: Port-Adapter-Architektur (Apache Isis)





Dauerhaft Erweiterbar

Nachhaltigkeit bei Architekturstil

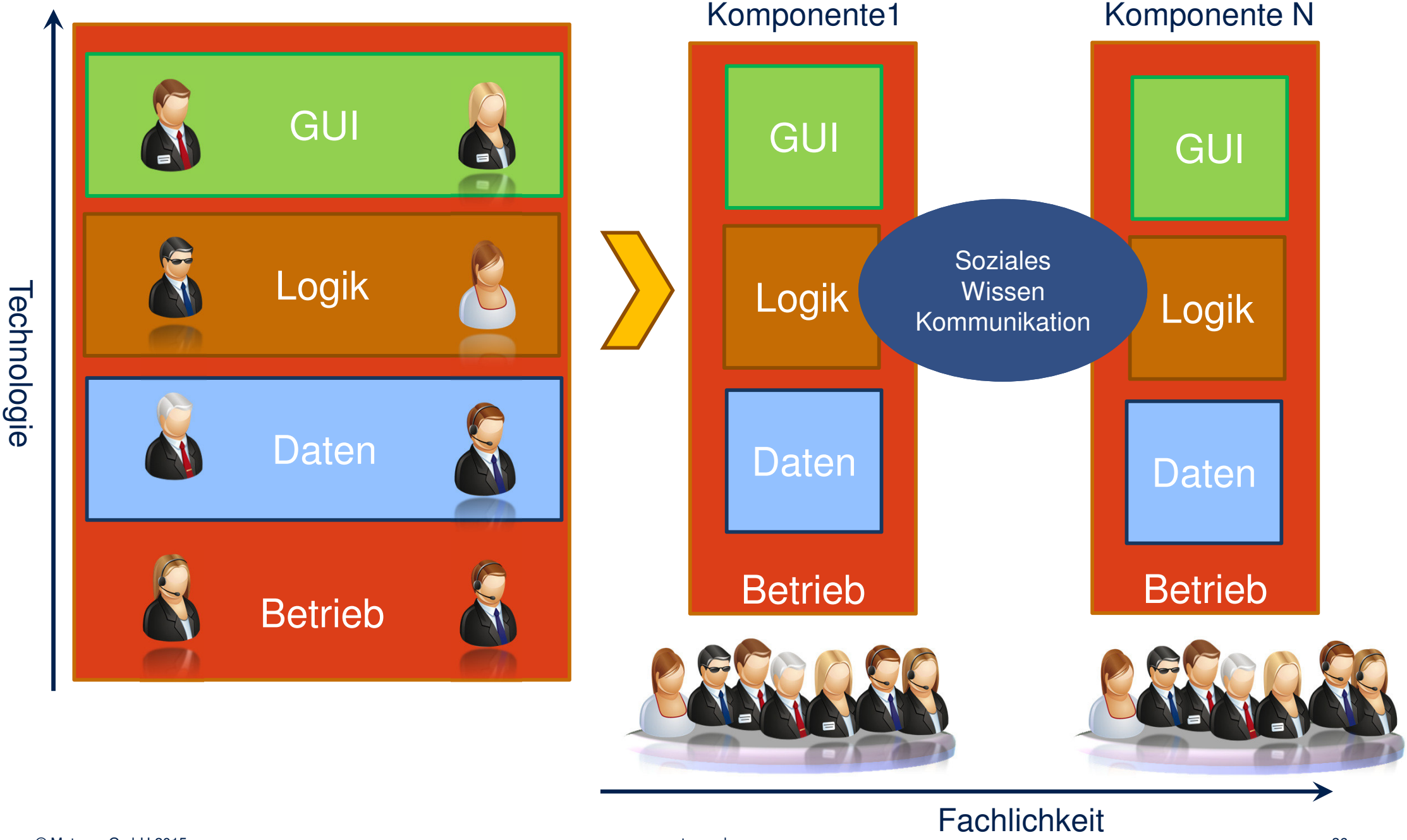
	Schichten	Ereignisorientiert	Microkernel	Microservices
Wartbarkeit	↓	↓	↑	↓
Erweiterbarkeit	↑	↑	↑	↑
Anpassbarkeit	↓	↓	↑	↑
Austauschbarkeit	↑	↓	↓	↑

Software Architecture Patterns, Mark Richards, 2015, O'Reilly

Softwarequalität

Prozess | Produkte | Infrastruktur | Personen | Kultur

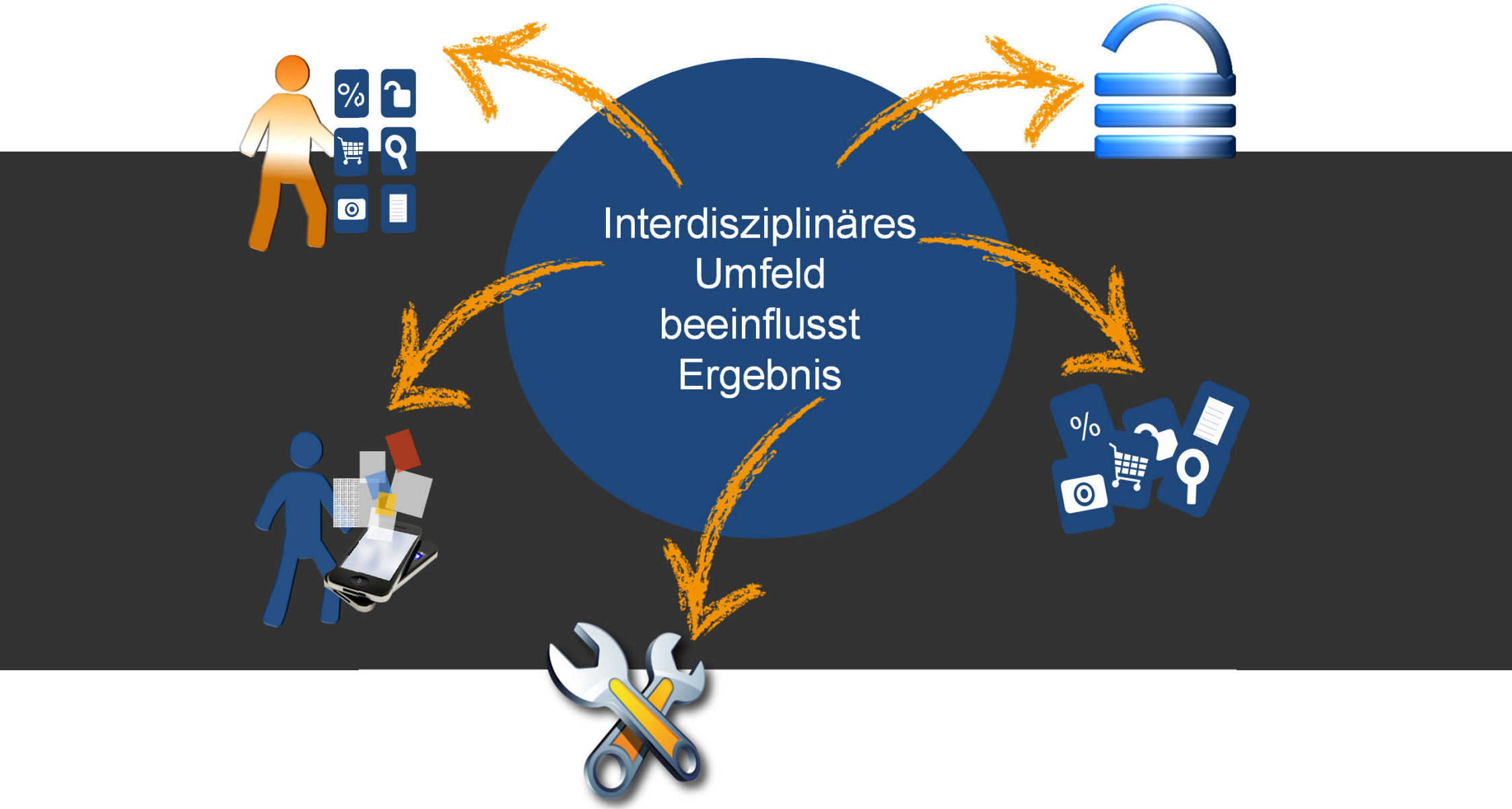
Weniger Spezialisten, mehr Generalisten



Das agile Manifest: 8-tes Prinzip

Agile Prozesse fördern **nachhaltige Entwicklung**. Die Auftraggeber, Entwickler und Benutzer sollten ein **gleichmäßiges Tempo** auf **unbegrenzte Zeit** halten können.

Peopleware - fachübergreifende Teams



Fachübergreifende Teams

Architektur-Entwickler-Tandem: wechselnde Position, immer im Tritt

Ziel & Weg im Auge

„Two-Person-Teams are magical“ (Frederick P. Brooks Jr.)

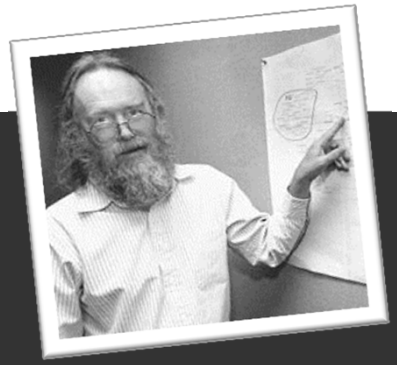
Risikopotential APIs

- Schnittstelle
- Protokoll
- Implementierung
- Infrastruktur
- Betrieb
- Überwachung
- Änderung

- Hohe Kosten, Risiken

- Besonders Cloud

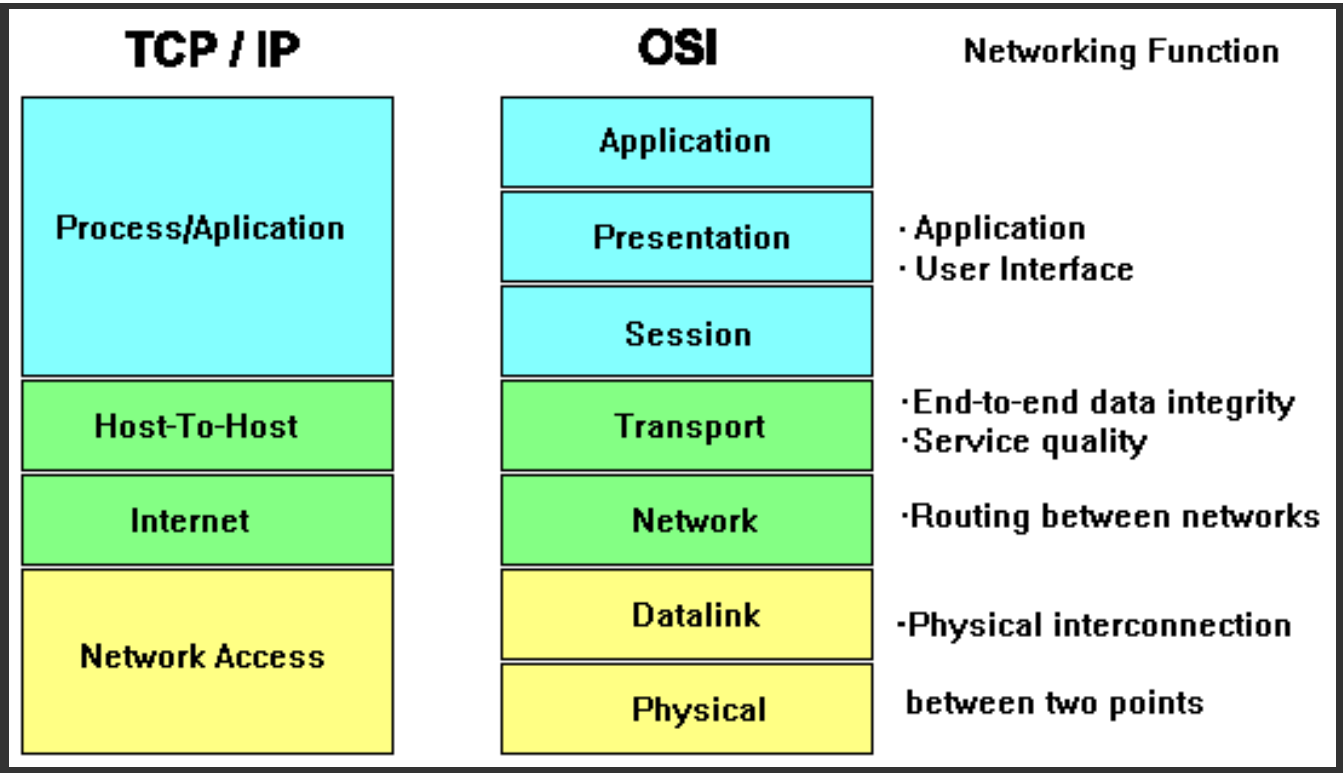
Wie Schnittstelle designen? – das OSI-Schichtenmodell



TCP Robustness Principle (RFC793)

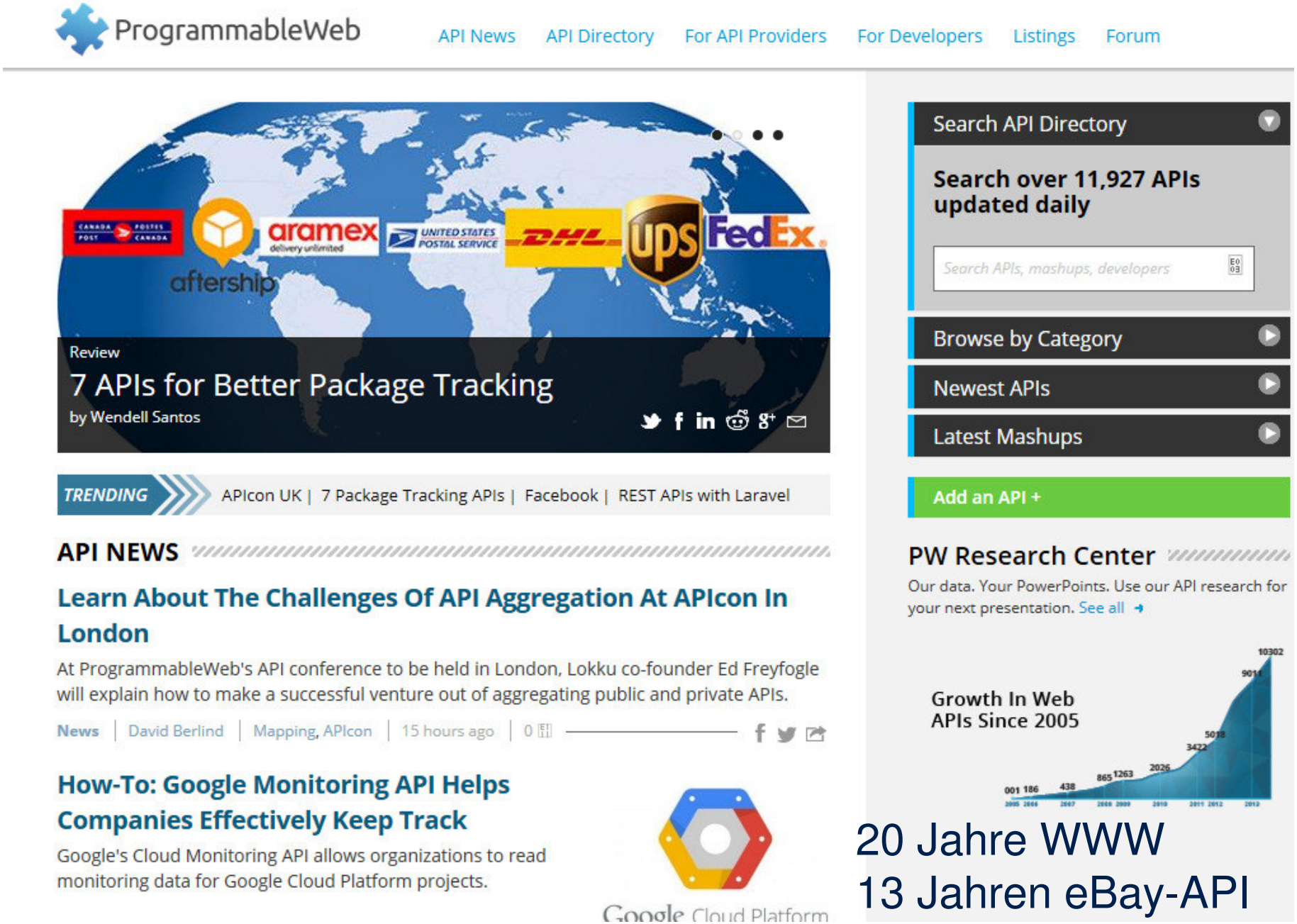
Jon Postel (1981)

*"be **conservative** in what you do, be **liberal** in what you accept from others"*



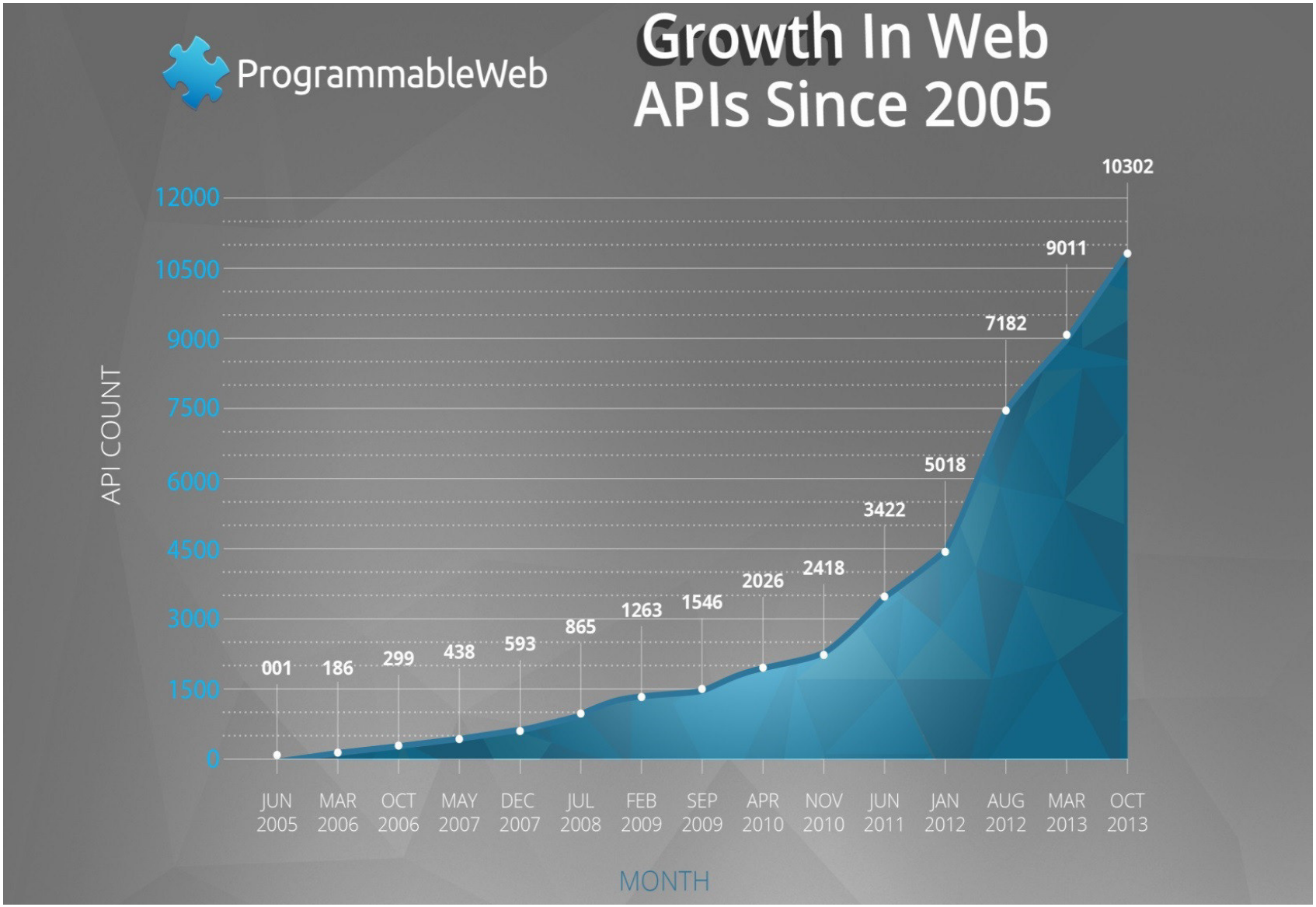
„Ein guter Zaun schafft gute Nachbarn“ Robert Frost

http://www.programmableweb.com



20 Jahre WWW
13 Jahren eBay-API

Über 10.000 Web-APIs



http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

7 Kompatibilitätsarten

1. Abwärtskompatibilität
2. Aufwärtskompatibilität
3. Binärkompatibilität
4. Quelltextkompatibilität
5. Verhaltenskompatibilität
6. Fehlerkompatibilität
7. Inkompatibilität

Compatibility Guide for JDK 8

Compatibility is a complex issue. This document discusses three types of potential incompatibilities relating to a release of the Java platform:

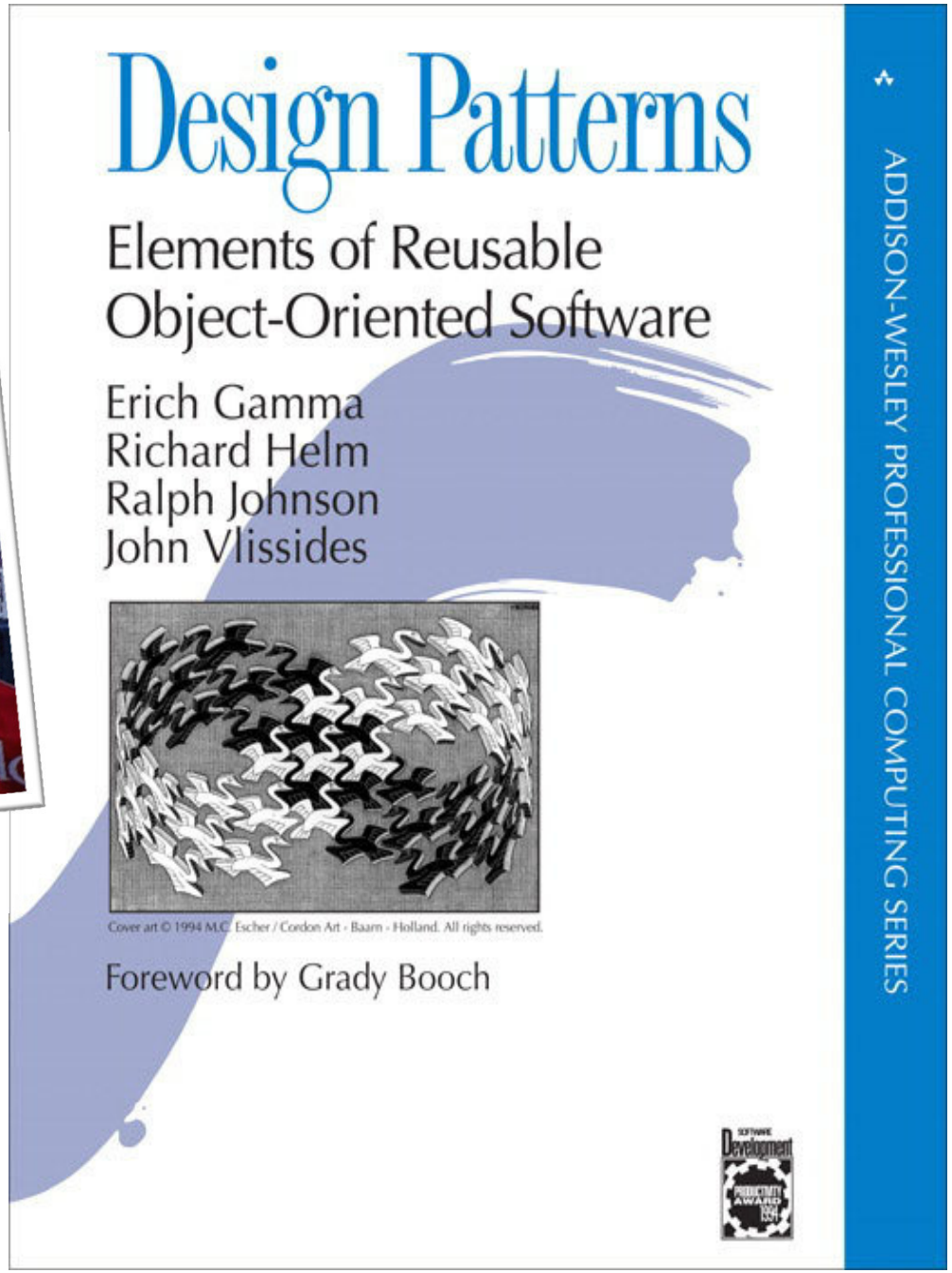
- Source: Source compatibility concerns translating Java source code into class files including whether or not code still compiles at all.
- Binary: Binary compatibility is defined in The Java Language Specification as: 'A change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error.'
- Behavioral: Behavioral compatibility includes the semantics of the code that is executed at runtime.

For more information, see Kinds of Compatibility, a section in the OpenJDK Developer's Guide.

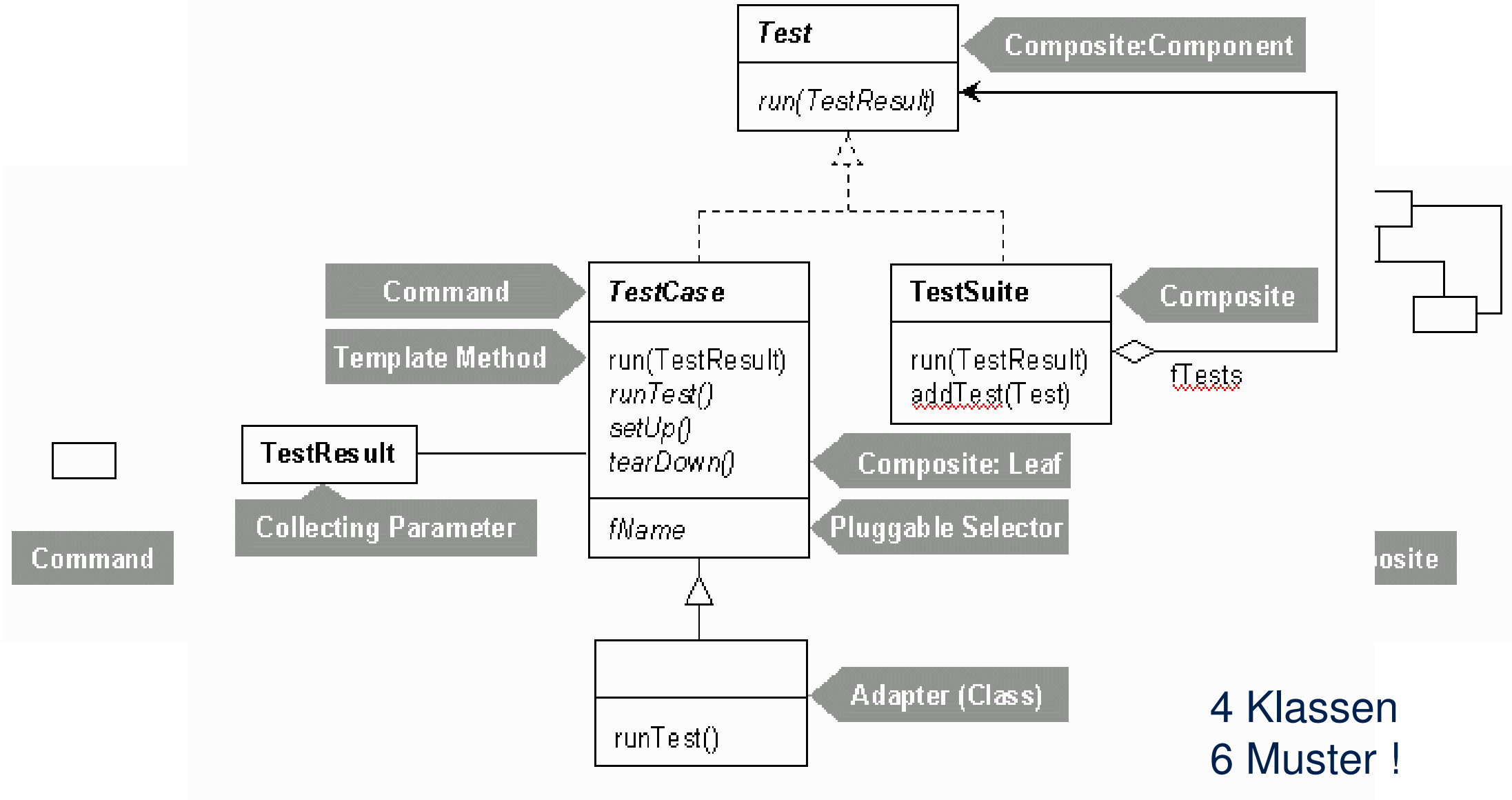
- "Binary Compatibility"
- "Source Compatibility"
- "Behavioral Compatibility"
- "Java Class Files"
- "Incompatibilities between Java SE 8 and Java SE 7"
- "Incompatibilities between JDK 8 and JDK 7"
- "Features Removed from Java SE 8"
- "Features Removed from JDK 8"
- "Deprecated APIs"

http://www.oracle.com/technetwork/java/javase/8-compatibility-guide-2156366.html

Design Patterns seit 20 Jahren



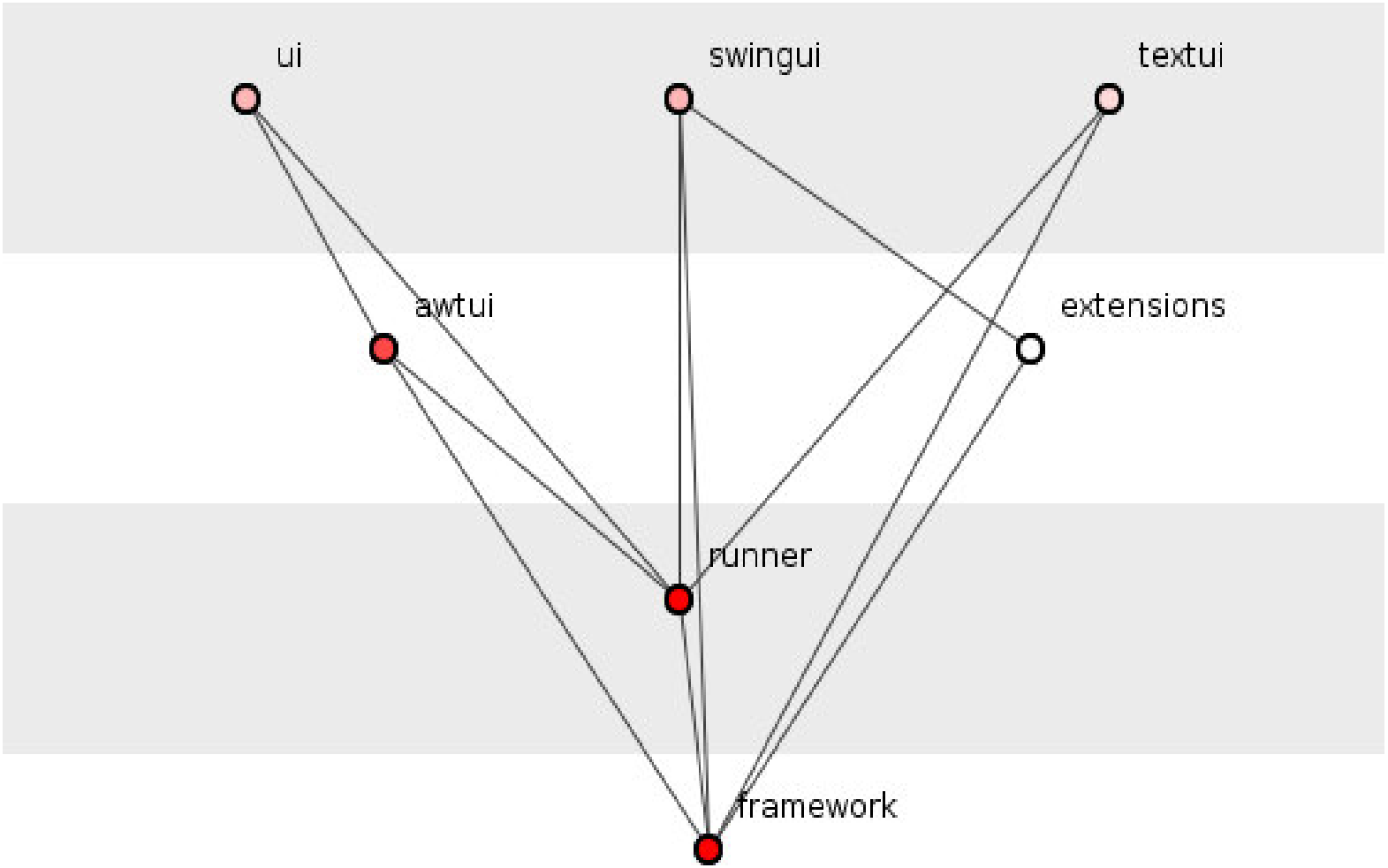
JUnit Design (1998, Kent Beck, Erich Gamma): Sauberes Design



http://junit.sourceforge.net/doc/cookstour/cookstour.htm

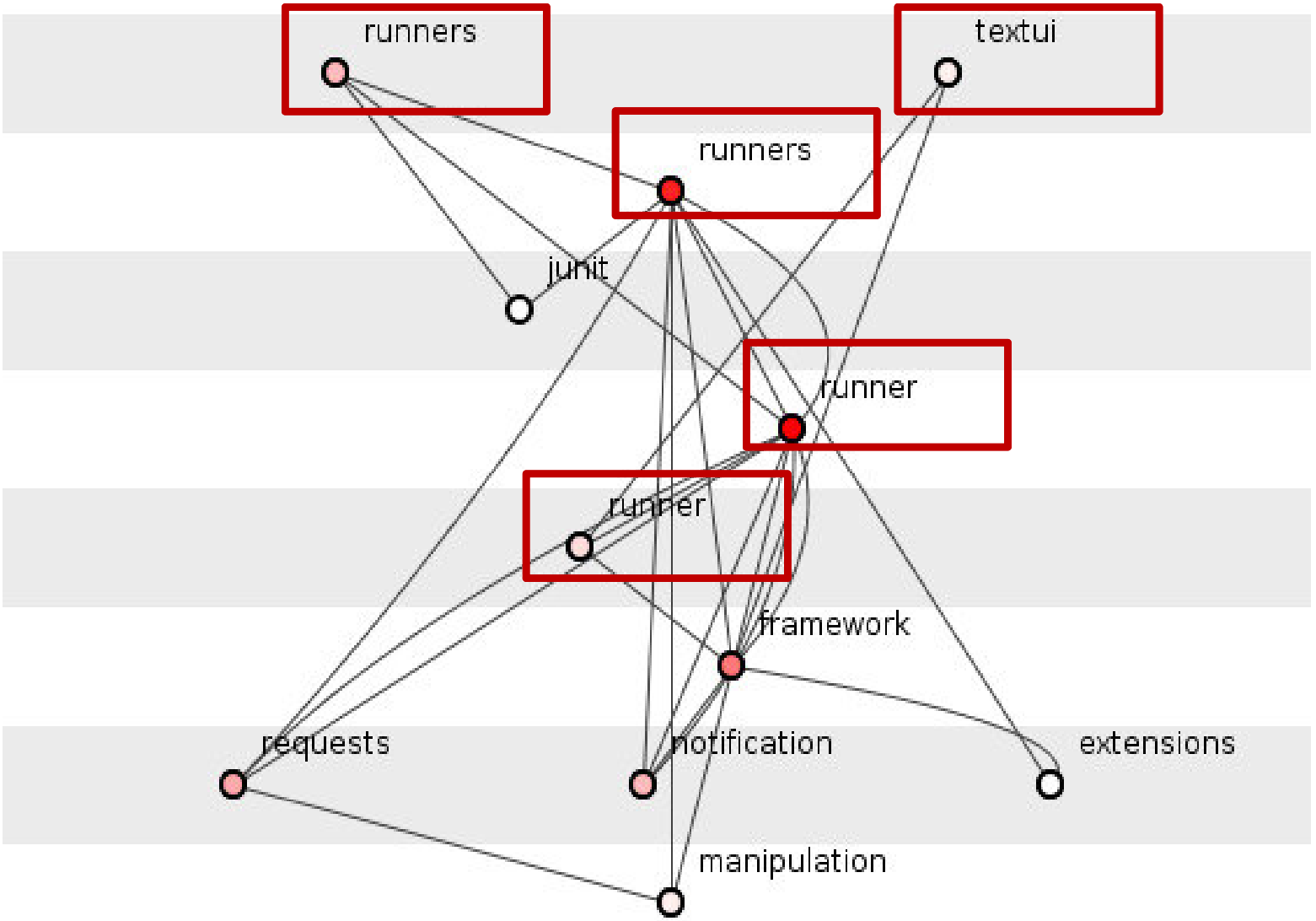
Test Infected: Programmers Love Writing Tests, Java Report, July 1998, Volume 3, Number 7

JUnit 3.7 (2001) – wie alles begann



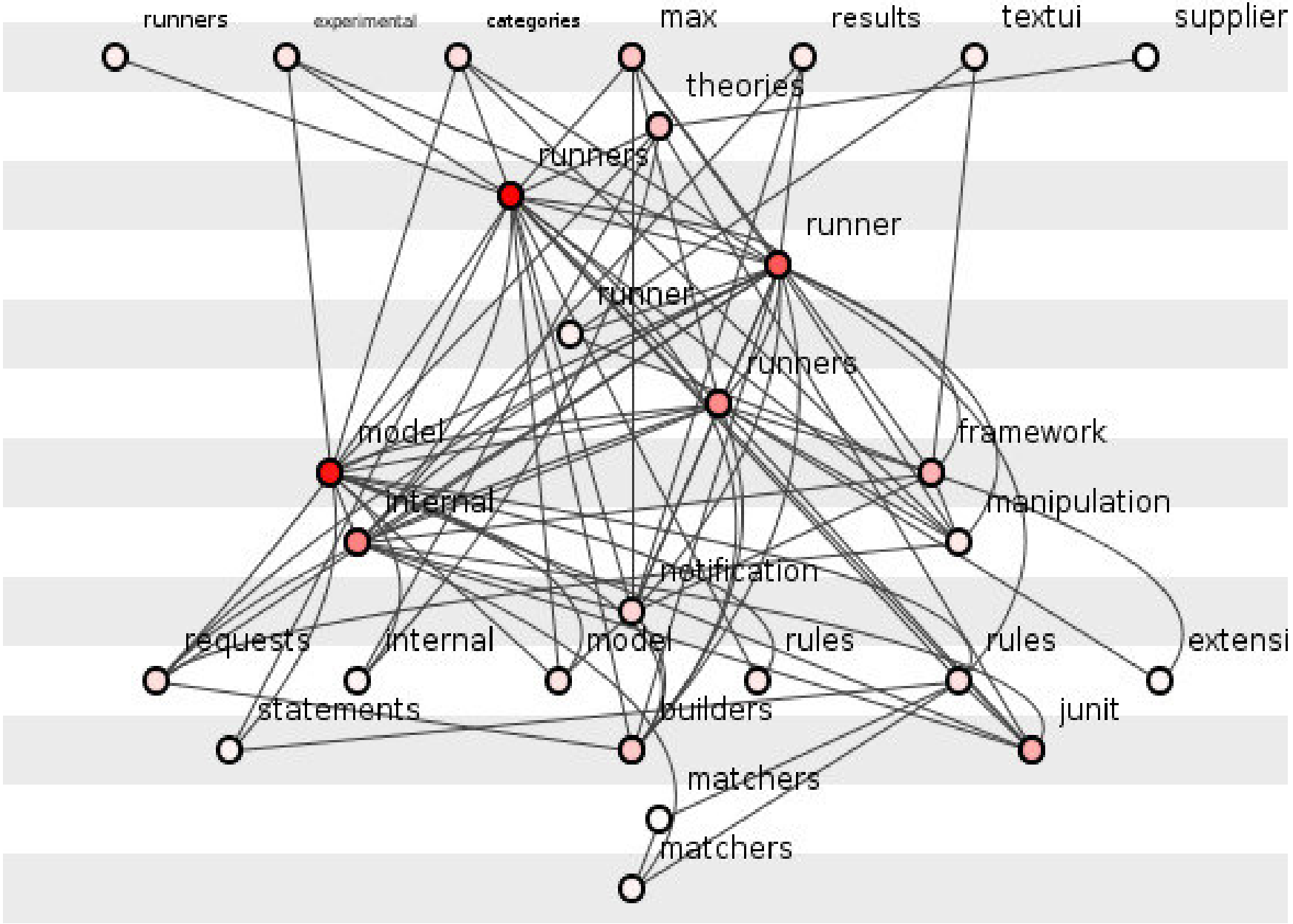
http://edmundkirwan.com/general/junit.html

JUnit 4.0 (2006) – entwickelt sich weiter



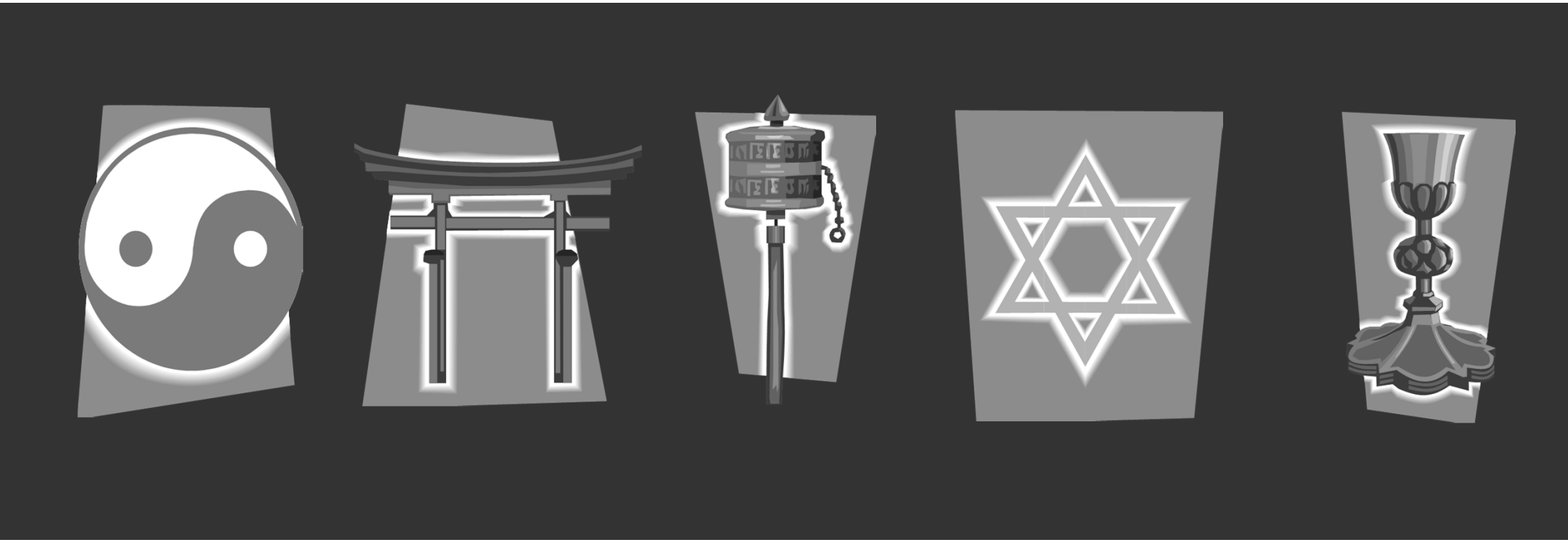
http://edmundkirwan.com/general/junit.html

JUnit 4.11 (2012) – und weiter 4.12 (2014)



GOF ist keine …

Magie oder Ersatz für gutes API-Design?



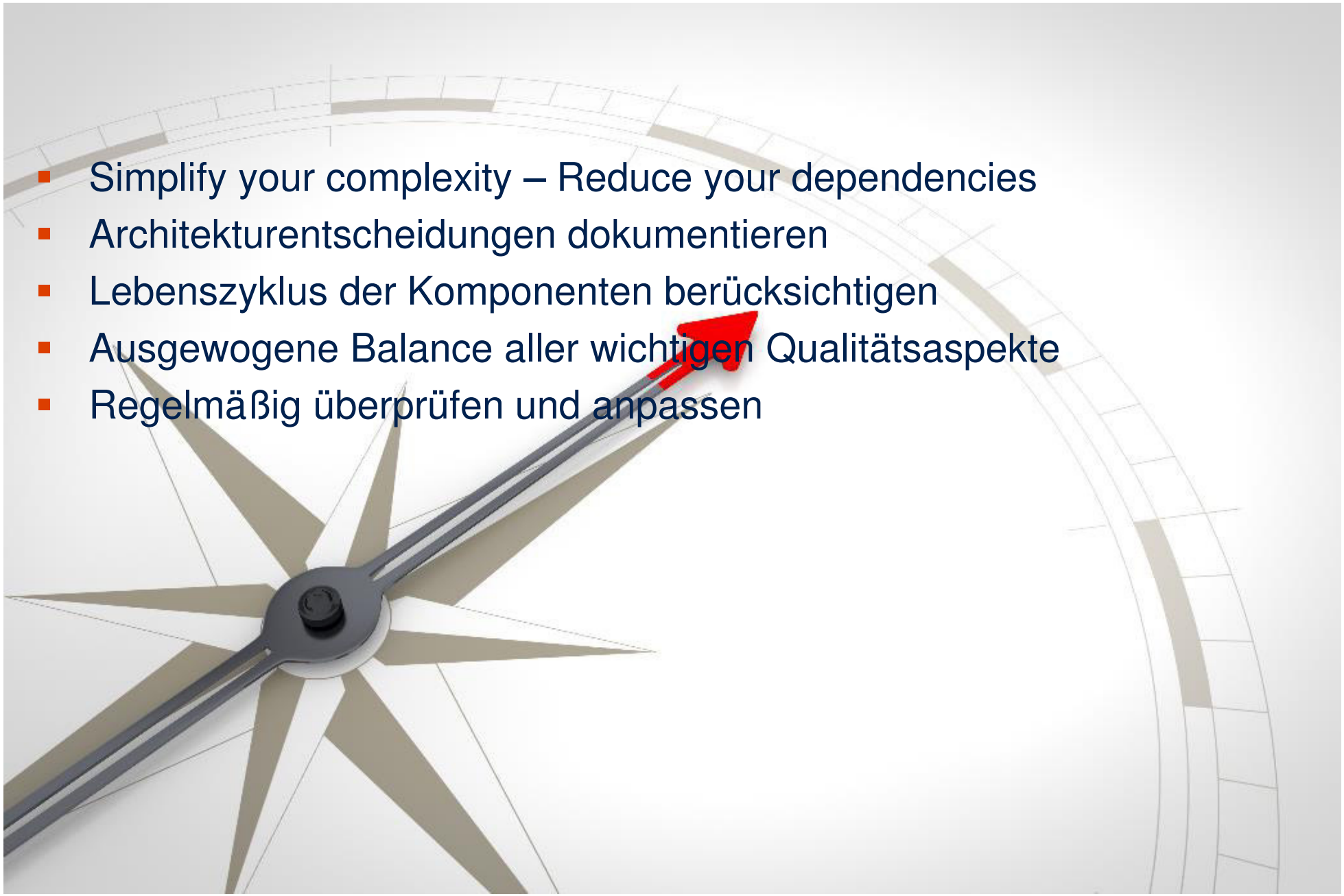
Bsp.: Einige behavioral patterns wurden durch Lambdas in Java obsolet

Wiederverwendbarkeit ist nicht einfach

*„Reuse is something that is **far easier to say than to do**.*
*Doing it requires both good design and very **good documentation**.“*

David Parnas (1964)

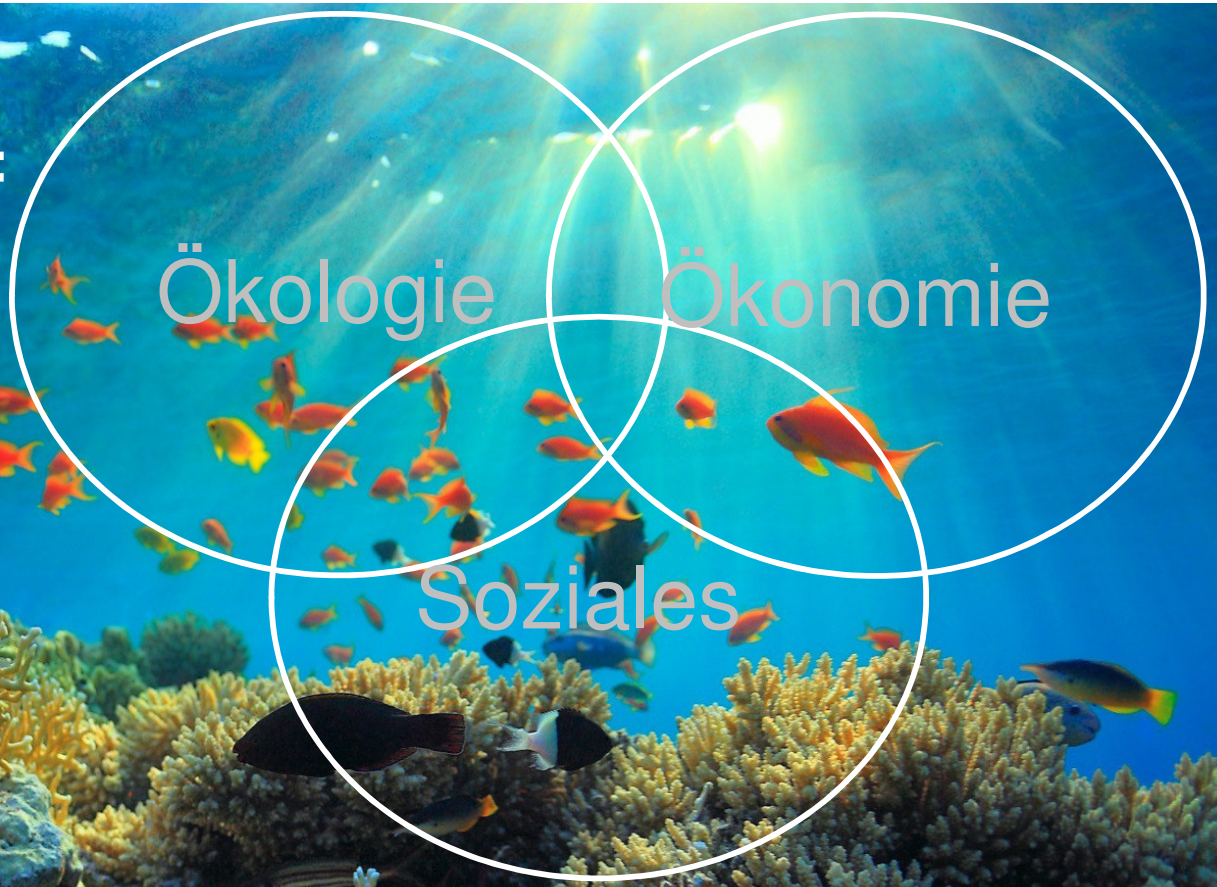
Fazit: Prinzipien für nachhaltige Software



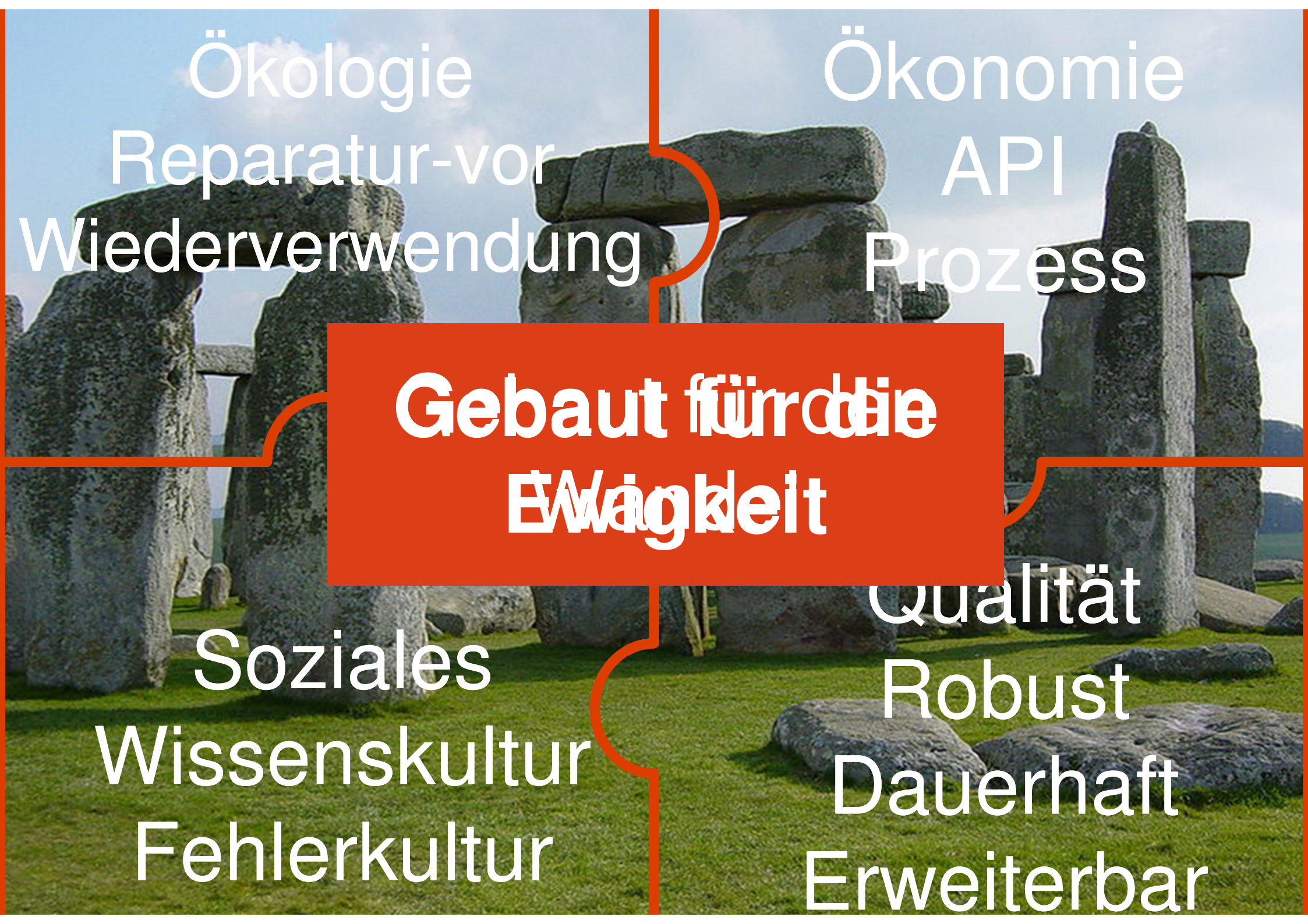
- Simplify your complexity – Reduce your dependencies
- Architekturentscheidungen dokumentieren
- Lebenszyklus der Komponenten berücksichtigen
- Ausgewogene Balance aller wichtigen Qualitätsaspekte
- Regelmäßig überprüfen und anpassen

Nutzen einer nachhaltigen Softwarearchitektur

Nachhaltige Softwareentwicklung:

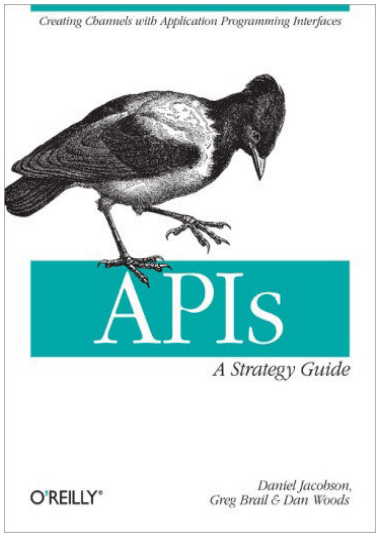


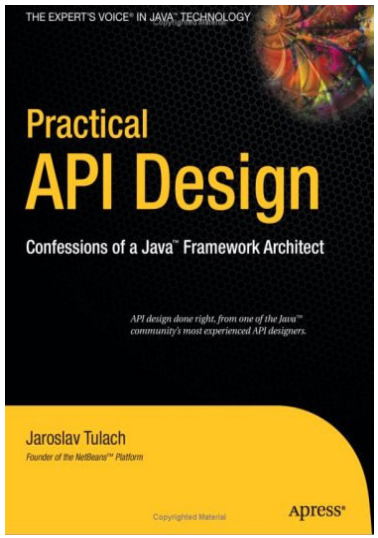
ein komplexes, empfindliches Ökosystem
mit vielen Beteiligten im Einklang

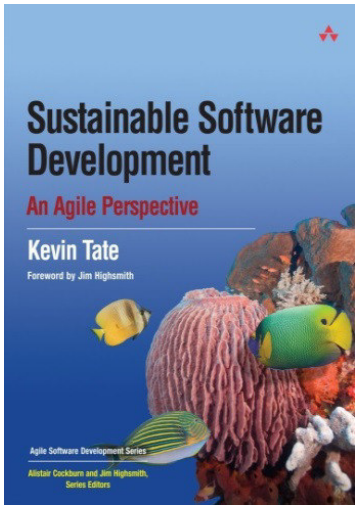

Ökologie
Reparatur-vor
Wiederverwendung
Ökonomie
API
Prozess
Gebaut für die Ewigkeit
Soziales
Wissenskultur
Fehlerkultur
Qualität
Robust
Dauerhaft
Erweiterbar

Weitere Infos:

- Measuring Architecture Sustainability, Heiko Koziolek, et al, IEEE Software, 2013, vol. 30
- Software Engineering with an Agile Development Framework, WikiBook, 2012
- Free and Open Source Software Technology for Sustainable Development: Sulayman Sowe, 2012
- Sustainable Software Development: An Agile Perspective, Kevin Tate, 2005
- Sustainable Software Development With Clean C++, Stephan Roth, leanpub, 2014
- Rüdiger Zarnekow, Fabian Löser, Nachhaltiges IT-Management, dpunkt, 2015
- The Forrester Wave: API Management Platforms, Jeffrey Hammond, 2013
- Frank Pientka, Gebaut für den Wandel, OBJEKTSpektrum 02/2015







MATERNA
Information & Communications

Vernetzt.

Kontakt.

Materna GmbH
Frank Pientka
Voßkuhle 37
44141 Dortmund
+49 231 5599-8854
Frank.Pientka@materna.de