

Software Engineering in Industrial Practice (SEIP)

Dr. Ralf S. Engelschall



: not covered in lectures



: key aspect of lectures

Custom Software Development

Commercial development of **non-standardised, fully individualised, and non-reusable company-specific** software for a **single customer**.

CSD



Standard Software Development

STD

Commercial development of **standardised, partially customisable, and fully reusable domain-specific** software for **many customers**.



Open Source Software Development

OSS

Non-commercial development of **standardised, highly customisable, and fully reusable generic** software for **many customers**.



Class: Graphics & Media

target audience: consumers & enterprises

Graphics Editing Application

GEA

Software for editing and rendering graphics in vector and bitmap format.



Examples: Cinema4D, Maya, Blender, After Effects, Illustrator, Inkscape, Scribus, Photoshop, GIMP, etc.



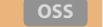
Graphics Animation Engine

GAE

Software for animating the 2D/3D virtual worlds of games and overlays of TV productions.



Examples: Unity, Unreal Engine, CryENGINE, Godot, HUDS, SPX-GC, Holographics, H2R Graphics, etc.



Audio/Video-Processing System

AVS

Software for live-processing and post-production of audio/video based multimedia streams.



Examples: vMix, OBS Studio, VLC, Lossless Cut, Handbrake, Adobe Premiere, FFmpeg, Nimble, etc.



Class: Business & Data

target audience: consumers & enterprises

Office Productivity Application

OPA

Software for productivity in the desktop-based office environment.



Examples: PowerPoint, Excel, Word, Visio, OmniGraffle, LibreOffice, Outlook, XMind, Firefox, Chrome, etc.



Business Information System

BIS

Software for driving business processes through interactive information management.



Examples: Vote, CampS, Mission Control, IPW, KEZ-PSC, TimeSheet, SAP ERP, OpenProject, etc.



Class: Machinery & Network

target audience: consumers & enterprises

Technical Control System

TCS

Software for controlling a physical machinery or technical system.



Examples: AquaTherm, AVM! FritzBox Firmware, BirdDog Camera Firmware, etc.



Network Communication System

NCS

Software for protocol-based communication of data over a computer network.



Examples: Apache, NGINX, HAProxy, Mosquitto, RabbitMQ, Node-RED, KeyCloak, etc.



Operating System Kernel

OSK

Software kernel for low-level operating a physical or virtual device and run programs on it.



Examples: Windows, macOS, iOS, Linux, FreeBSD, QNX, ChibiOS/RT, Kubernetes, Wildfly, etc.



Class: Development & Tools

target audience: vendors & suppliers

Software Development Kit

SDK

Software libraries and frameworks of reusable functionality for developing software.



Examples: NDI SDK, HAPI, GraphQL-IO, Sequelize, JDK, Spring, Hibernate, etc.



Software Development Tools

SDT

Software tools for editing, linting, compiling, packaging, distributing, and installing software.



Examples: Visual Studio Code, Sublime Text, GCC, GNU Binutils, NPM, JDK, Docker, Helm, etc.



Operating System Tools

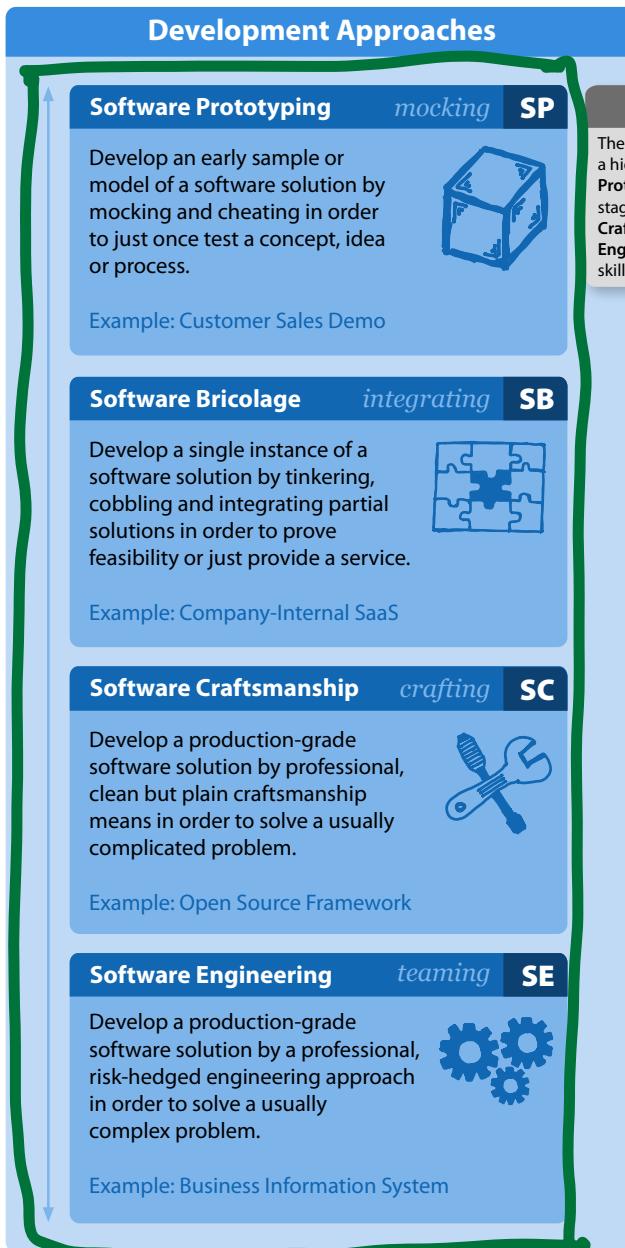
OST

Software tools for high-level operating a physical or virtual computing device.



Examples: Coreutils, Bash, Vim, TMux, FZF, cURL, RSYNC, OpenSSH, etc.





Development Approaches: Characteristics Comparison *

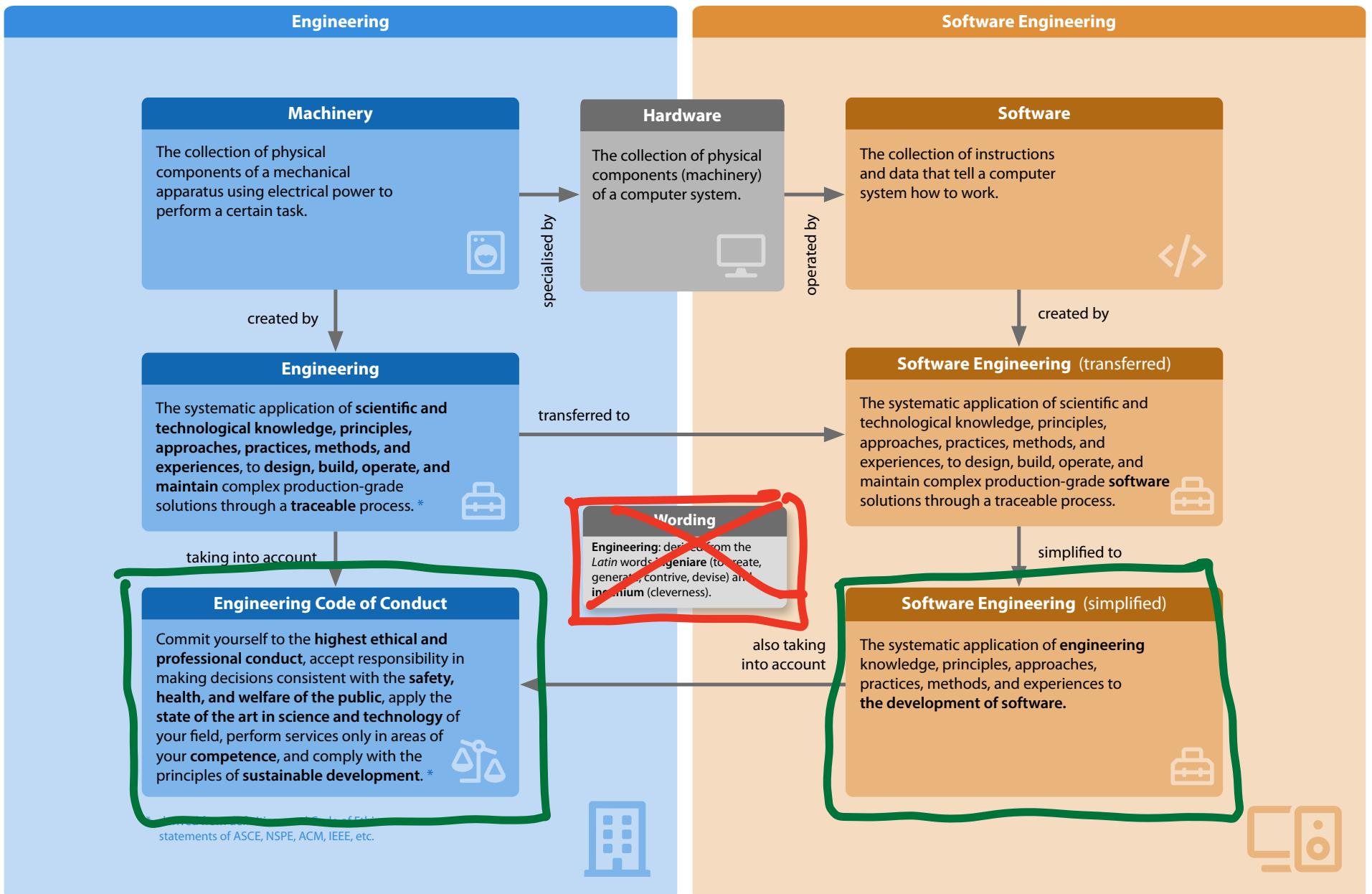
	Effort: Person-Days	Effort: Persons	Process: Risk-Hedge	Process: Traceability	Solution: Target Technology	Solution: Production-Grade	Solution: Sustainability	Solution: Claim	Solution: Life-Time Months	Solution: Lines of Code (k)
Software Prototyping	1-20	1-2	-	-	-	-	5%	0-3	0-3	
Software Bricolage	5-100	1-2	-	-	x	(x)	-	60%	3-24	1-10
Software Craftsmanship	5-100	1-2	-	-	x	x	x	100%	24-48	5-25
Software Engineering	>150	5-50	x	x	x	x	x	80%	>48	>25

* All figures are just rough orders of magnitude for indication and illustration purposes.

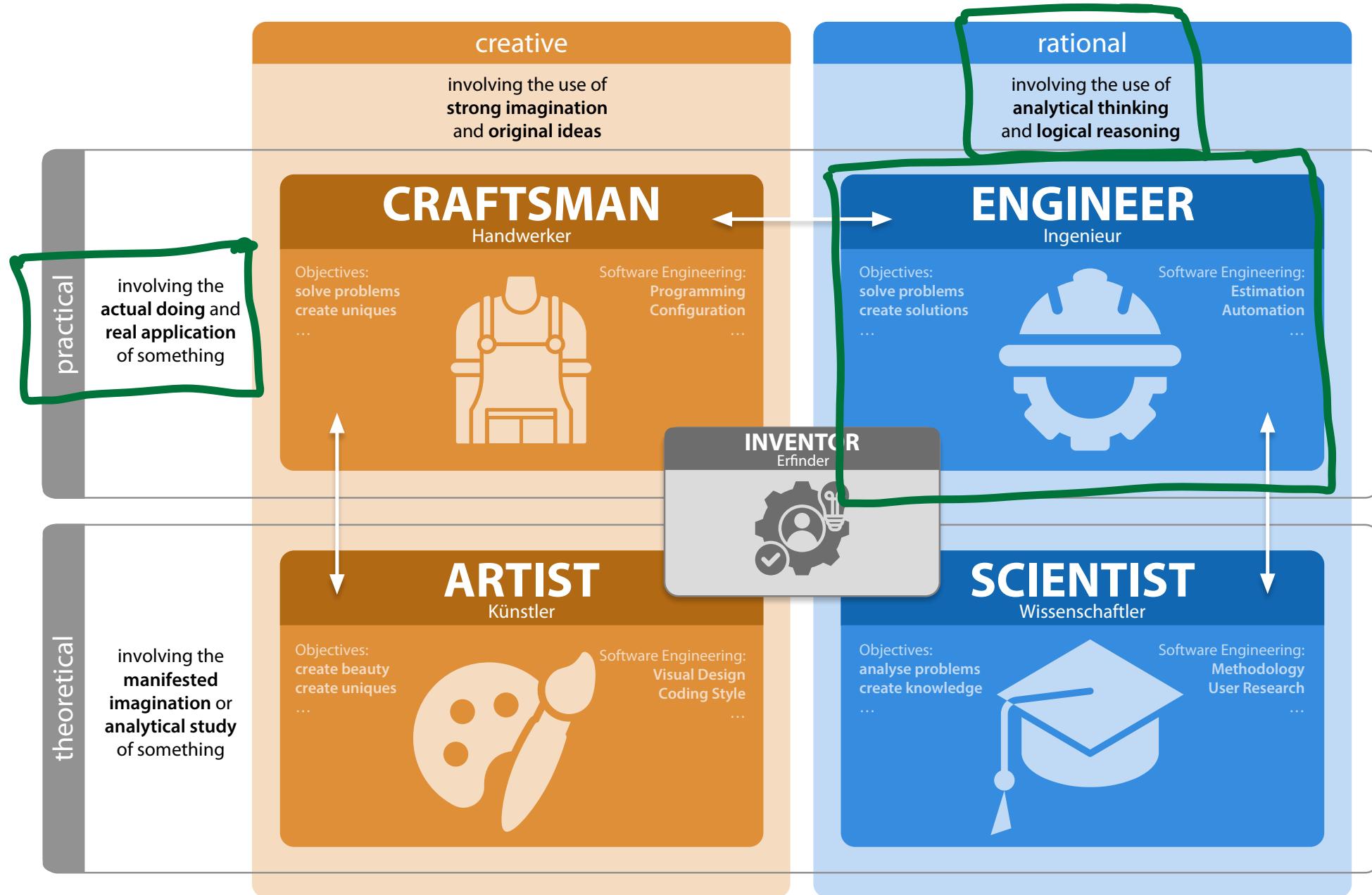
Development Approaches: Success Patterns

	Software Prototyping	Software Bricolage	Software Craftsmanship	Software Engineering
Performance Responsibility Model	One-Man-Show Single Mental	One-Man-Show Single Mental	One-Man-Show Single Mental/Documented	Team Play Separated Documented
Decisions Process Optimisation	Implicit Minimized Time	Implicit Partial Efficiency	Implicit/Explicit Partial Effectiveness	Explicit Complete Economics
Risks Stakeholders Mastering	Ignore Ignore Time-Constraint	Ignore Ignore Complexity	Ignore Ignore Complication	Mitigate Manage Complexity
Solutions Standards Efforts	Use Full Use Configuration	Use Partial Use Integration	Use Partial Potentially Create Programming	Use Partial Use Programming
Target Sustainability Traceability	Demo No No	Solution Partial No	Product Full Partial	Product Full Full

goal & approach



Profession Characteristics



Discipline Claim

FA **FARSIGHTED**
weitblickend

Be farsighted in your solution finding.
Sei weitblickend in deiner Lösungsfindung.



AR: Scalable Node Spoke
DV: Plugins



TE **TENET-ORIENTED**
grundsatzorientiert

Oriентate yourself on fixed tenets in your approach and solution finding.
Orientiere dich an festen Grundsätzen in deinem Vorgehen und deiner Lösungsfindung.



AR: Separation of Concern
DV: Strict Coupling-Style



TH **THOUGHTFUL**
wohlüberlegt

Act thoughtful in your approach and solution finding.
Agiere wohlüberlegt in deinem Vorgehen und deiner Lösungsfindung.



AR: Modularization
DV: Algorithmic Control Structure

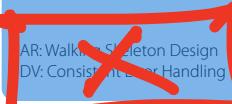


HO **HOLISTICALLY**
ganzheitlich

Think holistically and in the long-term when finding your solutions.
Denke ganzheitlich und langfristig in deiner Lösungsfindung.



AR: Walking Skeleton Design
DV: Consistent Error Handling



AD **ADEQUATE**
angemessen

Ensure that your approach and solutions are adequate to the boundary conditions.
Sorge dafür, daß dein Vorgehen und deine Lösungen angemessen zu den Rahmenbedingungen sind.



AR: No Cloud Native Complexity
DV: No Over-engineered Abstraction



FE **FEASIBLE**
machbar

Ensure that your approach and solutions can be realised at reasonable costs.
Sorge dafür, daß dein Vorgehen und deine Lösungen mit vernünftigen Kosten realisiert werden können.

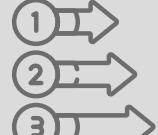


AR: Existing Framework Functionality
DV: Realistic Programming Model



IN **INCREMENTAL**
inkrementell

Apply the depth of your discipline incrementally.
Wende die Tiefe deiner Disziplin inkrementell an.



AR: Identified Solution Cruxes
DV: Minimum viable Product



VA **VALUEABLE**
wertvoll

Provide clearly recognizable added values with your approach and solutions.
Liefere klar ersichtliche Mehrwerte mit deinem Vorgehen und deinen Lösungen.



AR: Technology Stack Design
DV: User-Story backlog Functionality



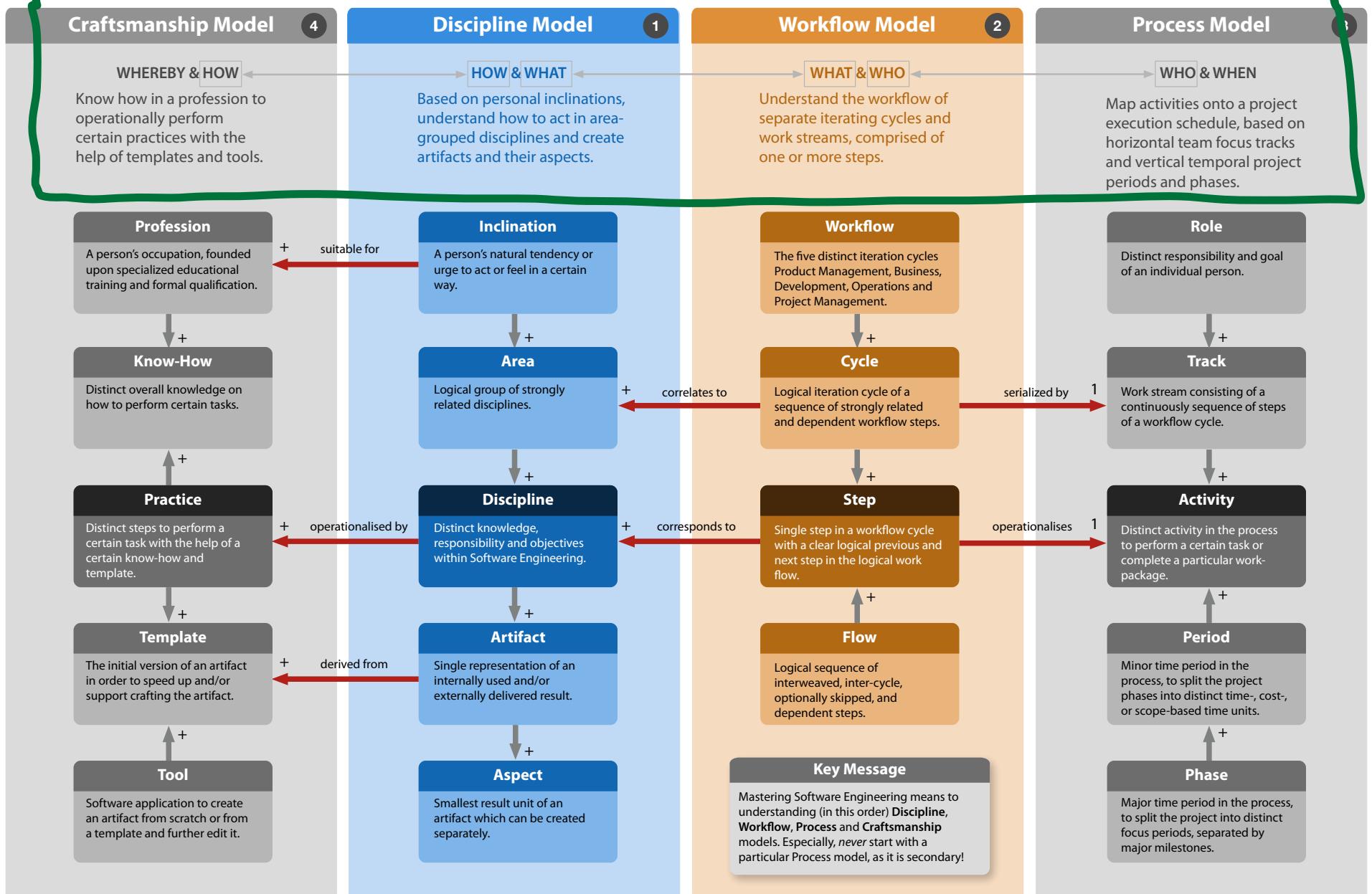
SU **SUSTAINABLE**
nachhaltig

Create sustainable solutions that are well integrated into their environment.
Erschaffe nachhaltige Lösungen, die gut in ihre Umgebung integriert sind.



AR: Interoperable Interfaces
DV: Maintainable Code

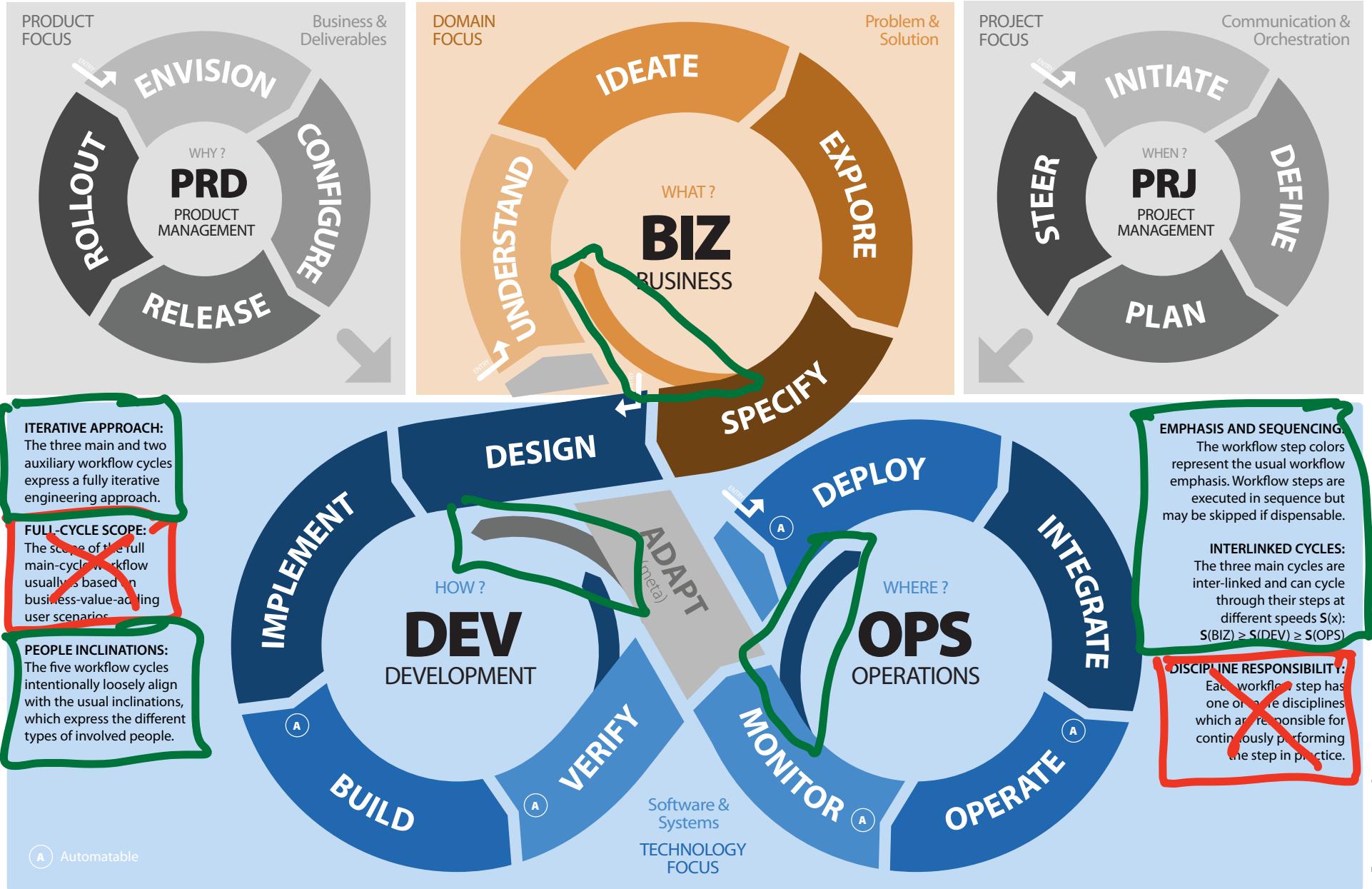




Software Engineering Disciplines

ANALYSIS	AN	ARCHITECTURE	AR	CONFIGURATION	CF	ANALYTICS	AC	MANAGEMENT	MG
Software Requirements	REQ	Software Architecture	SWA	Software Versioning	VER	Software Reviewing	REV	Product Management	PRD
Identify Needs: We understand which outcomes of the solution are most valuable to users.	3 BB	Design Software: We design an orthogonal, well-balanced and well-considered solution.	1 WB	Version Artifacts: We place every artifact of the solution under strict version control.	1 BB	Review Code: We regularly and semantically peer-review the source code of the solution.	4 WB	Push Product: We continuously push the development and release of the solution to the users.	3 BB
Requirements Engineer / Business Analyst		Software Architect		Configuration Manager		Software Tester		Product Manager / Product Owner	
Domain Modeling	DOM	System Architecture	SYA	Software Assembly	ASM	Software Testing	TST	Project Management	PRJ
Determine Solution: We model and specify the solution through involved functional and non-functional aspects.	2 WB	Design Systems: We ensure that the solution fits optimally into its environment.	2 BB	Assemble Artifacts: We build and package the solution through an automated and repeatable mechanism.	1 BB	Test Solution: We adequately test the functional and non-functional aspects of the solution.	2 BB	Steer Process: We rigorously balance time, cost and scope to react on changes and reach the goals.	3 BB
Business Analyst / Business Architect		System Architect / Enterprise Architect		Build Manager / Build Engineer		Software Tester		Project Manager	
business-oriented & domain-specific		constructive & technological		infrastructural & technological		analytical & domain-specific		people-oriented & process-oriented	
EXPERIENCE	EX	DEVELOPMENT	DV	DELIVERY	DL	COMPREHENSION	CP	ADJUSTMENT	AD
User Experience	UXP	Software Development	DEV	Software Deployment	DPL	Usage Documentation	DOC	Project Coaching	COA
Optimize Workflows: We align the solution to the perspective of the target audience.	3 BB	Implement Code: We develop the solution outside-in, from coarse to fine aspects.	1 WB	Deploy Artifacts: We ship and deploy the solution through an automated and repeatable mechanism.	1 BB	Document Solution: We adequately document the usage and operation of the solution.	2 WB	Support Members: We ensure that project members use state-of-the-art methodology, technology, and tools.	4 BB
User Experience Expert		Software Engineer / Software Developer		System Engineer		Technical Writer		Project Coach / Methodology Master	
User Interface	UID	Software Refactoring	REF	System Operations	OPS	User Training	TRN	Change Management	CHG
Design User Interfaces: We design a useful, intuitive, and beautiful user interface for the solution.	2 WB	Refactor Code: We regularly and holistically refactor the solution to ensure long-term quality.	4 BB	Operate Solution: We ensure that our infrastructures and the solution can be operated in a resilient and secure way.	4 BB	Train Users: We adequately train the users and operators of the solution.	4 BB	Involve Stakeholders: We ensure that all stakeholders of the solution are suitably involved.	3 BB
User Interface Designer / Graphics Designer		Software Engineer / Software Developer		System Administrator / System Operator		Product Expert		Change Manager	
scalability layer (from 4/most to 1/least dispensable)					WB white-box view (details before whole) BB black-box view (whole before details)				

Software Engineering Workflow



Software Engineering Steps



1. WORKFLOW CYCLES

The workflow has five cycles which continuously iterate through their steps. Workflow steps are executed in each cycle in sequence, but may be skipped if dispensable in a particular iteration of the process. The length of an iteration is arbitrary, but can be e.g. about 1/3 of a Scrum sprint.

2. WORKFLOW STEPS:

The workflow steps describe a logical activity which has to be performed. Each step relates to one or more discipline areas and their corresponding disciplines, which express the operative responsibilities for each workflow step. In each discipline individual roles act.

3. WORKFLOW ROLES:

The workflow roles are held by individual persons. Each role is primarily responsible for a particular workflow step. In addition, each role can be secondarily responsible for other workflow steps or at least actively support those steps.

4. PROJECT SCHEDULE:

To create a particular project execution schedule, the five cycles, their iterations and their steps have to be mapped onto a timeline. The cycles are mapped onto (horizontal) timeline tracks, the iterations are mapped onto (vertical) timeline phases, and the steps are mapped onto timeline activities.

5. PROCESS FLOWS (THE CRUX):

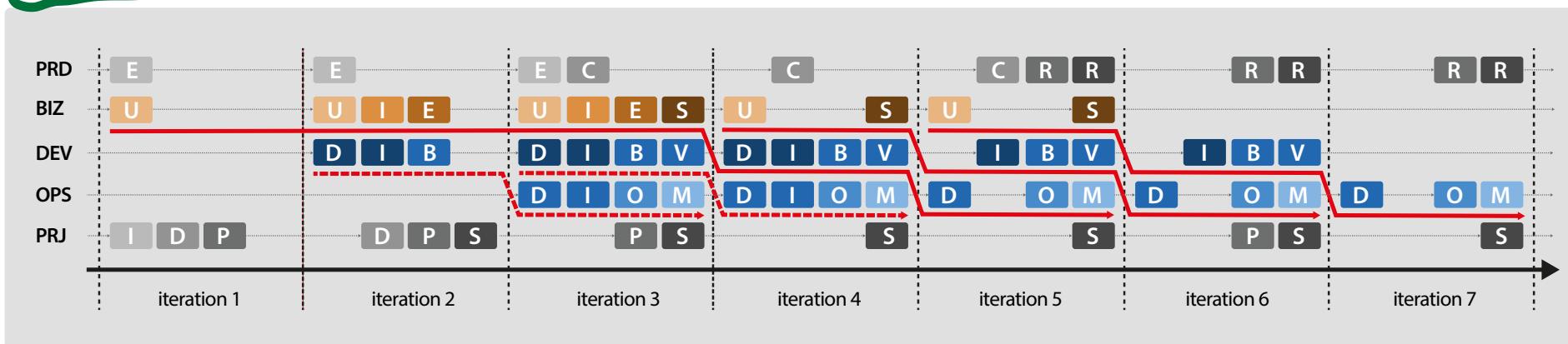
The activities across the cycles can (and should) be linked into individual (diagonal) waterfall-like flows, although the execution schedule, from the perspective of the cycles, is fully iterative. There are multiple such flows in parallel and they are usually highly interleaved on the project timeline in order to maximally utilize the team.

6. PROCESS ADAPTION:

In the meta-step ADAPT, the process is adapted by choosing which workflow steps are required for the next iteration. The major input for this decision is the current solution state and the feedback on it by the customer.

	business-oriented & domain-specific				constructive & technological				infrastructural & technological				analytical & domain-specific				people-oriented & process-oriented			
	REQ	AN	EX	SWA	AR	DV	CF	DL	AC	CP	MG	PRD	PRU	COA	AD	CGH				
Requirements Engineer																				
Business Architect		DOM	UXP	UID																
User Experience Expert																				
User Interface Designer																				
Software Architect																				
System Architect																				
Software Developer																				
Configuration Manager																				
Build Manager																				
System Engineer																				
System Administrator																				
Software Tester																				
Software Tester																				
Technical Writer																				
Product Trainer																				
Product Owner																				
Project Manager																				
Project Coach																				
Change Manager																				

* responsible (primarily)
* responsible (secondarily)
+ supporting



1 Software Requirements Specification			input / what	REQ
Requirements: Customer Journey	REQ	UXP	PRD	2 ENSIGHT
Requirements: Solution Vision	REQ	UXP	PRD	2 ENSIGHT
Requirements: Functional Requirements	PRD	UXP	REQ	1 UNDERSTAND
Requirements: Non-Functional Requirem.	PRD	SWA	REQ	1 UNDERSTAND
Domain Model: Data Model	REQ	DOM		1 SPECIFY
Domain Model: Use Cases	UXP	REQ	DOM	1 SPECIFY
Domain Model: Use Case Scenarios	UXP	REQ	DOM	2 SPECIFY
			User Interface:	UXP UID
			Usage Concept	2 SPECIFY
			Language Conventions	3 SPECIFY
			Dialog Patterns	UXP UID
			Dialog Storyboard	REQ UXP UID
			Visual Design	PRD UXP UID
				3 SPECIFY

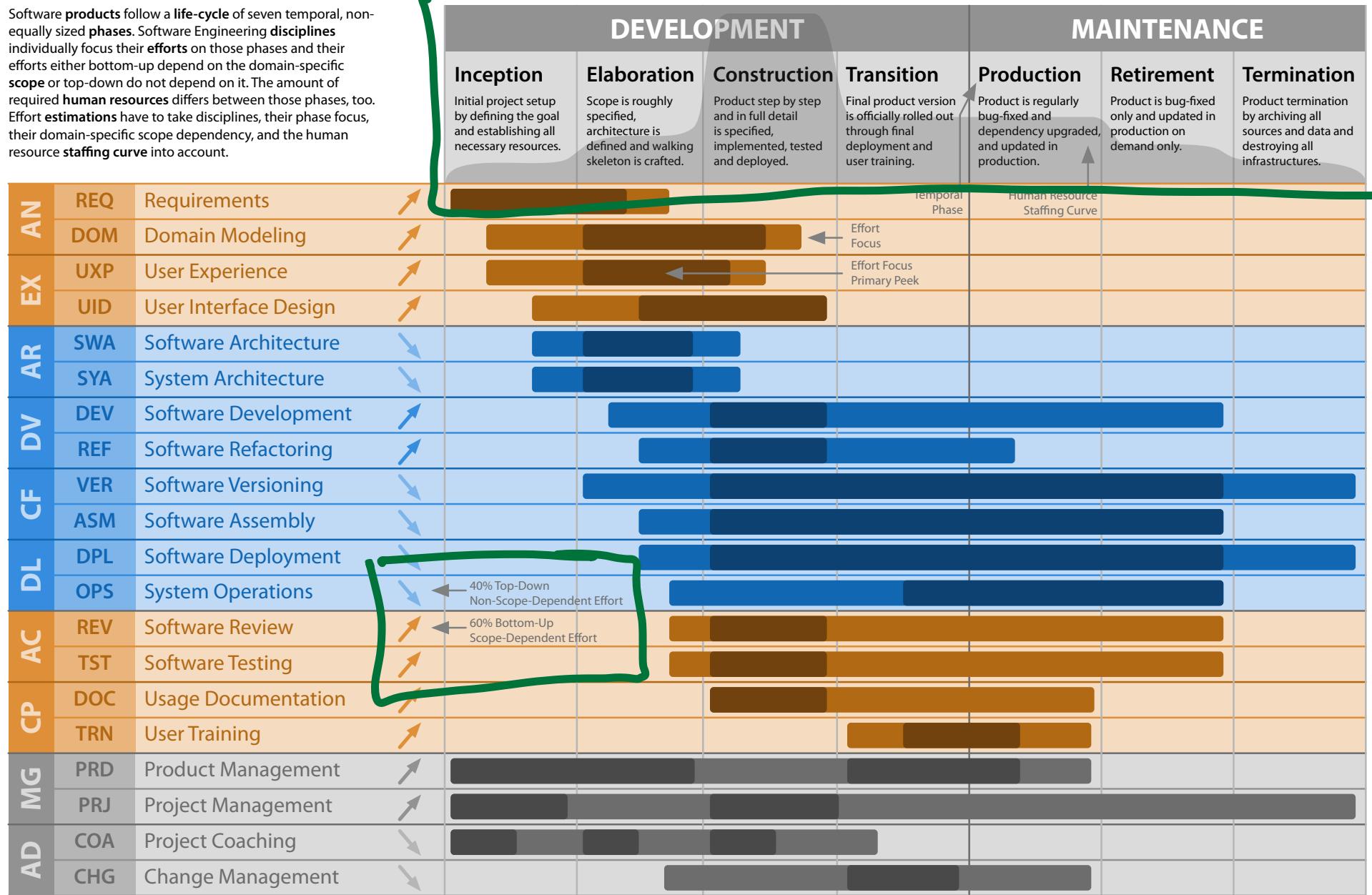
2 Software Architecture Specification			input / how	ARC
Viewpoint: Context View	SYA	SWA	PRD	2 ENSIGHT
Viewpoint: Functionality View	DOM	SYA	SWA	1 DESIGN
Viewpoint: Information View	DOM	SWA		1 DESIGN
Viewpoint: Concurrency View	DEV	SYA	SWA	2 DESIGN
Viewpoint: Development View	ASM	VER	SWA	DEV
Viewpoint: Deployment View	SYA	SWA	OPS	DPL
Viewpoint: Operations View	SYA	SWA	DPL	OPS
			Perspective:	DEV SWA DPL
			Configurability & Extensibili.	3 INTEGRATE
			Perspective:	OPS DPL SYA SWA
			Performance & Scalability	1 DESIGN
			Perspective:	SYA SWA DPL OPS
			Availability & Recoverability	2 OPERATE
			Perspective:	DEV SYA SWA
			Reliability & Resilience	2 DESIGN
			Perspective:	DEV SWA
			Interoperability & Compatib.	3 DESIGN
			Perspective:	PRD TST SWA DEV
			Compliance & Tracability	3 IMPLEMENT
			Perspective:	DEV SWA
			Security & Safety	1 DESIGN

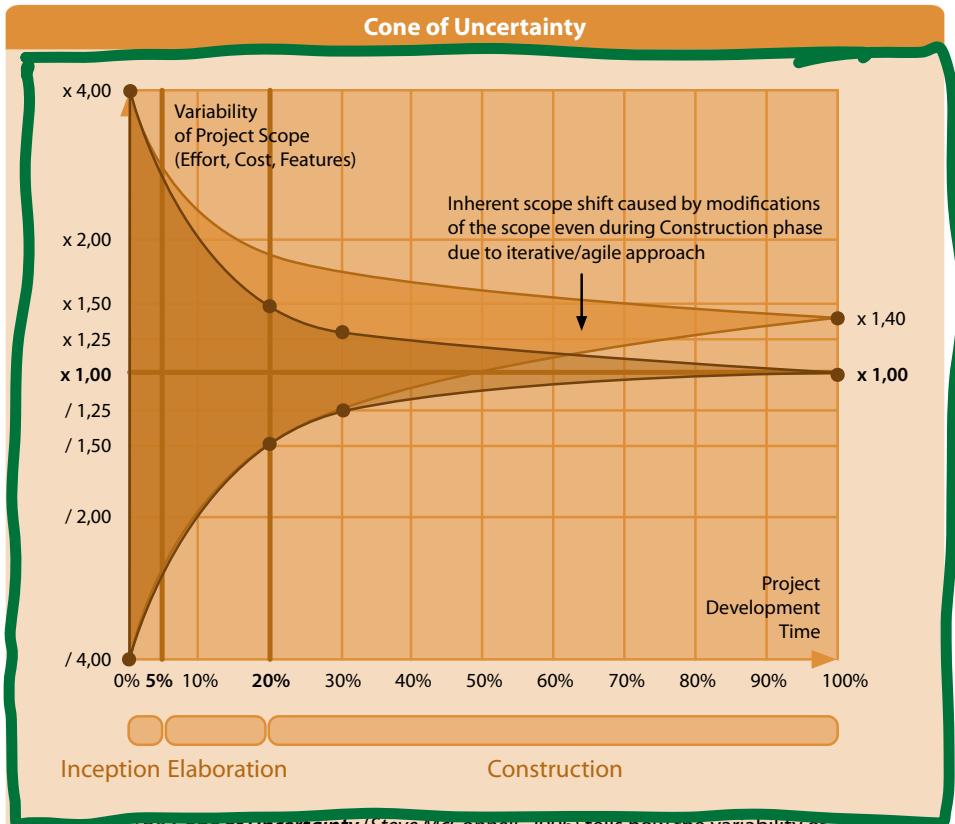
3 Software Implementation Results			output / what	IMP
Source Code: Application	REF	DEV	1 IMPLEMENT	ASM
Source Code: Build Automation	DEV	VER	ASM	1 BUILD
Source Code: Test Automation	DEV	TST	3 VERIFY	DEV VER DPL
Source Code: Deployment Automation	DEV	VER	3 DEPLOY	OPERA
Source Code: Operation Automation	DEV	OPS	3 OPERATE	OPERATE
			Notice: Artifacts vs. Aspects	Notice: Internal vs. External
			The four Artifact Sets shown here just cluster the individual Artifacts and their contained Aspects . The Artifacts can be represented in an arbitrary graphical and/or textual form and be provided in an arbitrary format. The Aspects just structure an individual Artifact internally.	In a Software Engineering project, additional internal Artifacts are created by the Disciplines in order to perform their work efficiently and effectively. The Artifacts shown here are the external ones which glue together the Disciplines and which are part of the delivery set.

4 Software Documentation Results			output / how	DOC
User Guide: Usage Tutorial	UXP	TRN	DOC	2 SPECIFY
User Guide: Functionality Reference	DOM	TRN	DOC	1 SPECIFY
User Guide: Release Information	PRD	VER	RELEASE	3 RELEASE
Operation Guide: Configuration Reference	TRN	DOC	DPL DEV	1 IMPLEMENT
Operation Guide: Deployment Procedure	TRN	DOC	OPS DPL	1 DEPLOY
Operation Guide: Operation Procedures	TRN	DOC	DPL OPS	2 OPERATE
			Notice: Artifact Tagging	Notice: Domain vs. Technology
			Each Artifact is tagged with the primarily and secondarily responsible Disciplines, the primary Step of the Workflow where the Artifact is developed, and the Scalability Layer (1 to 3, indicating more to lesser importance).	The Software Requirements Specification and the Software Documentation Results primarily have a domain-specific focus. The Software Architecture Specification and the Software Implementation Results primarily have a technological focus.

Software Engineering Efforts

Software products follow a life-cycle of seven temporal, non-equally sized phases. Software Engineering disciplines individually focus their efforts on those phases and their efforts either bottom-up depend on the domain-specific scope or top-down do not depend on it. The amount of required **human resources** differs between those phases, too. Effort estimations have to take disciplines, their phase focus, their domain-specific scope dependency, and the human resource **staffing curve** into account.

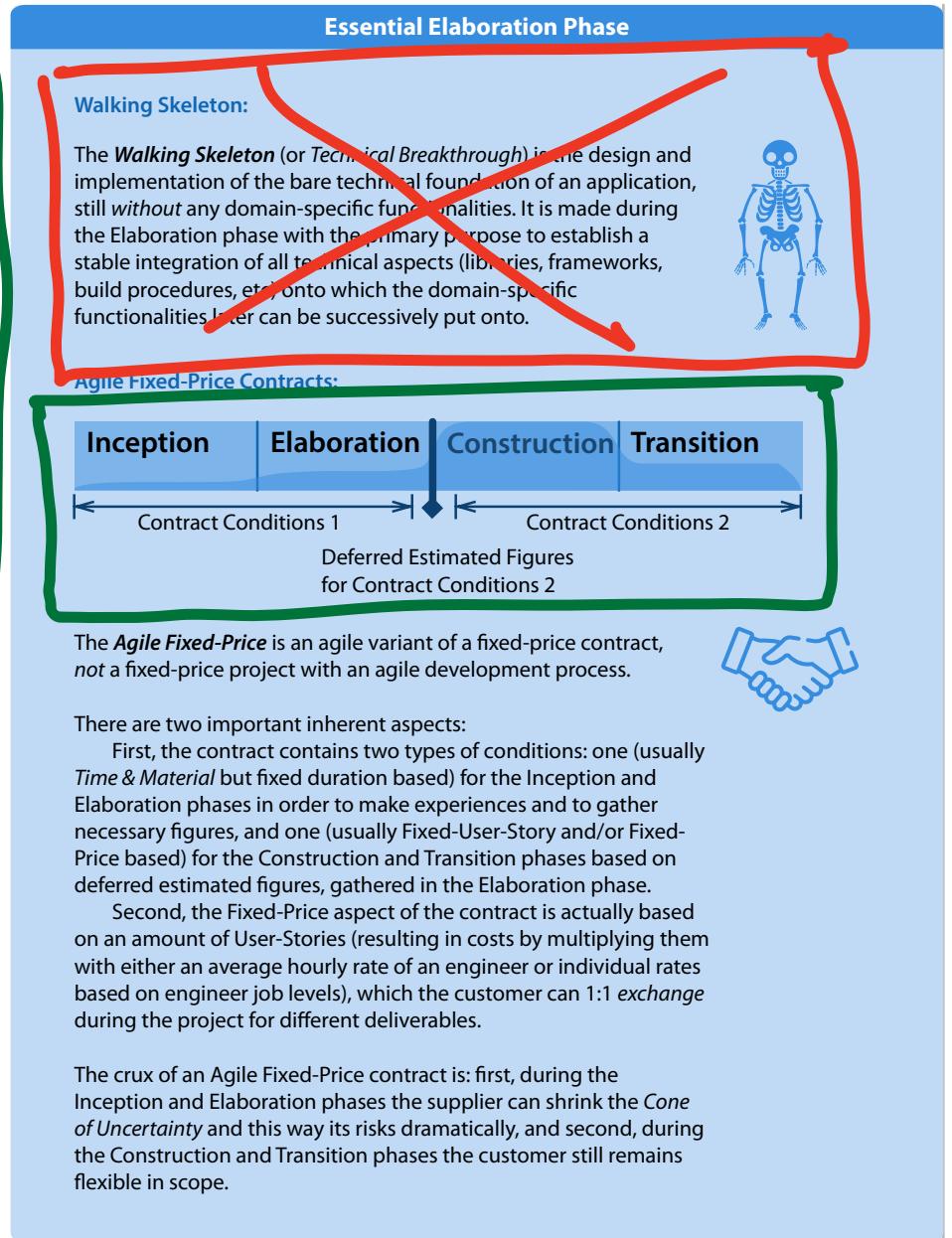




The **Cone of Uncertainty** (Steve McConnell, 2006) tells how the variability of the project scope (measured in Effort, Cost or Features) in Software Development changes over time. Initially, it usually is within the range of +/- 400% of the final scope.

The early development phases Inception and Elaboration especially have to ensure that within the first 20% of the project, the variability is reduced noticeably to just +/- 50%. During the initial iterations of the Construction phase within the first 30% of the project, the variability usually can be further reduced to about +/- 25%.

For iterative/agile approaches, experience showed that during the Construction phase inherently the final scope further shifts by about + 40% due to the just step-by-step learned required details of the required solution. This especially has to be taken into account for estimations.



Estimation & Variability

Three-Point Estimation and Estimation Variability Classes:

$$e = (b + 4m + w) / 6 \quad \text{expected effort (weighted average)}$$

$$s = (w - b) / 6 \quad \text{standard deviation (effort variation)}$$

b: best-case (optimistic)
 m: most-likely (realistic)
 w: worst-case (pessimistic)

Insane Variability: +/- 10%
 Very Good Variability: +/- 15%
 Good Variability: +/- 20%
 Acceptable Variability: +/- 25%

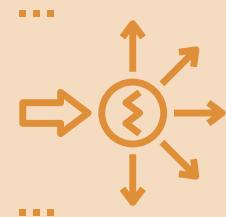


Sizes & Variability

Estimation Sizes and Estimation Variability:

T-Shirt-Size (Logically)	XXS	XS	S	M	L	XL	XXL	XXXL
Fibonacci-Size (PD or SP)	0,50	1	2	3	5	8	13	21
Size Variability (-)	0,25	0,25	0,50	0,50	1,00	1,50	2,50	4,00
Size Variability (+)	0,25	0,50	0,50	1,00	1,50	2,50	4,00	8,00

Notice: Estimations can be done in *Person-Days (PD)* or *Story-Points (SP)*. In both cases, keep in mind to use something like the *Fibonacci numbers* which increase in a non-linear fashion and express the increasing variability with the increasing total amount of estimated effort.



Conversion & Normalization

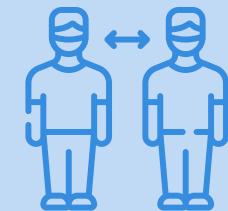
1. Ask Estimator:

"How many Person-Days do you need when you can focus on this task?"

2. Convert from Estimator to Performer:

(see also CAP model, <http://cap-model.com>)

		Performer					
		Non-Linear Effort Reduction	0%	Practitioner 10%	Master 25%	Expert 45%	Guru 80%
Estimator	Novice	1,00	0,90	0,75	0,55	0,20	
	Practitioner	1,11	1,00	0,83	0,61	0,22	
	Master	1,33	1,20	1,00	0,73	0,27	
	Expert	1,82	1,64	1,36	1,00	0,36	
	Guru	5,00	4,50	3,75	2,75	1,00	



Risk Mitigation & Upscaling

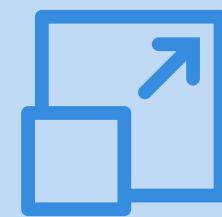
3. Adjust for Reality:

Estimator Optimism: +30%

Performer Meetings: +20%

4. Adjust for Uncertainty:

Domain	Inception	Elaboration	Construction	Technology				
				unknown	20%	60%		
Process	unknown	30%	40%	20%	partially known	15%	20%	10%
	partially known	15%	20%	10%	fully known	0%	0%	0%
	fully known	0%	0%	0%	unknown	60%	40%	10%
People	unknown	30%	20%	5%	partially known	30%	20%	0%
	partially known	30%	20%	5%	fully known	0%	0%	0%
	fully known	0%	0%	0%	unknown	60%	40%	0%



Requirements Basics

Requirements Specification

A binding document that specifies the requirements for a solution, by focusing on the **WHAT** and **WHY** of the solution — and *not* giving instructions for the **HOW**.

The documented set of requirements has to be: correct, unambiguous, complete, consistent, ranked, verifiable, modifiable, and traceable.



Requirement Classes

FR Functional (Shall Do)	NFR Non-Functional (Shall Be)
--	---

FR Functional (Shall Do): A condition or capability that a solution must have to provide its service in terms of its behaviour and information. Think: Functionality.



NFR Non-Functional (Shall Be): A condition, property or quality that a solution must have to satisfy a contract, standard, or other formally imposed obligation. Think: Constraints and “*-ilities”.



Requirement Interdependencies

POS Positive (Backing)	NEG Negative (Trade-Off)
--------------------------------------	--

POS Positive (Backing): One requirement supports the other (e.g. for NFRs: Maintainability and Comprehensibility usually support Adaptability, Portability, Modifiability, etc., and Scalability usually supports Availability, etc.)



NEG Negative (Trade-Off): One requirement interferes with the other (e.g. for NFRs: Security usually interferes with Efficiency, Usability, Performance, etc., and Orthogonality can interfere with Usability)



Requirement Characteristics

- S Specific**: The requirement is precise, unambiguous, and clear on what should be done.
- M Measurable**: The requirement can be verified when it has been achieved by use of a particular test.
- A Achievable**: The requirement is achievable given existing circumstances and feasible and viable solutions.
- R Relevant**: The requirement is relevant to the goals of the context.
- T Time-Bound**: The requirement can be achieved within a reasonable time frame.

Requirement Life-Time

- E Enduring**: The requirement lasts forever, as it is derived from core activities and organisational structure.
- V Volatile**: The requirement can be temporary, as it might change over time.

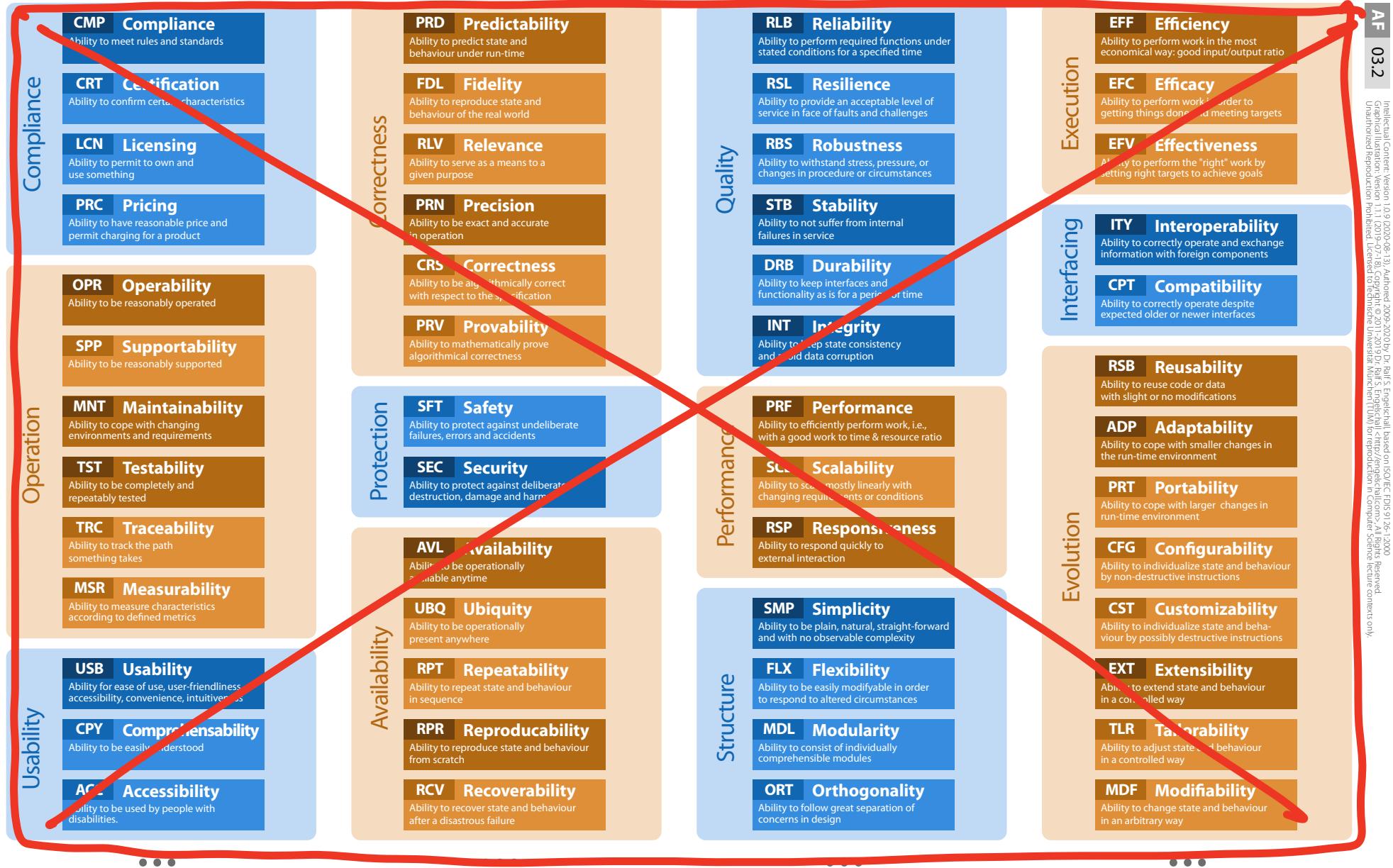
Requirement Expression

```

<req-id> <req-name>:
<subject/actor>
SHALL
<result/action/condition>
BECAUSE
<rationale>
    
```



Non-Functional Requirements



Architecture Stargate

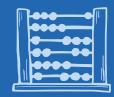
Requirements
Process
Estimation
Calculation
Automation
...

Engineering



Algorithms
Data Structures
Network Protocols
Formal Languages
Design Patterns
...

Science



Craftsmanship



Coding
Debugging
Tooling
Assembling
Deploying
Testing
...

Art



User Experience
Interface Usability
Interface Accessibility
Graphics Design
Coding Style
...

Architecture

**“firmitas,
utilitas
et
venustas”**
— Vitruvius
Marcus Vitruvius Pollio
90-70 B.C.

structural
operative
attractive

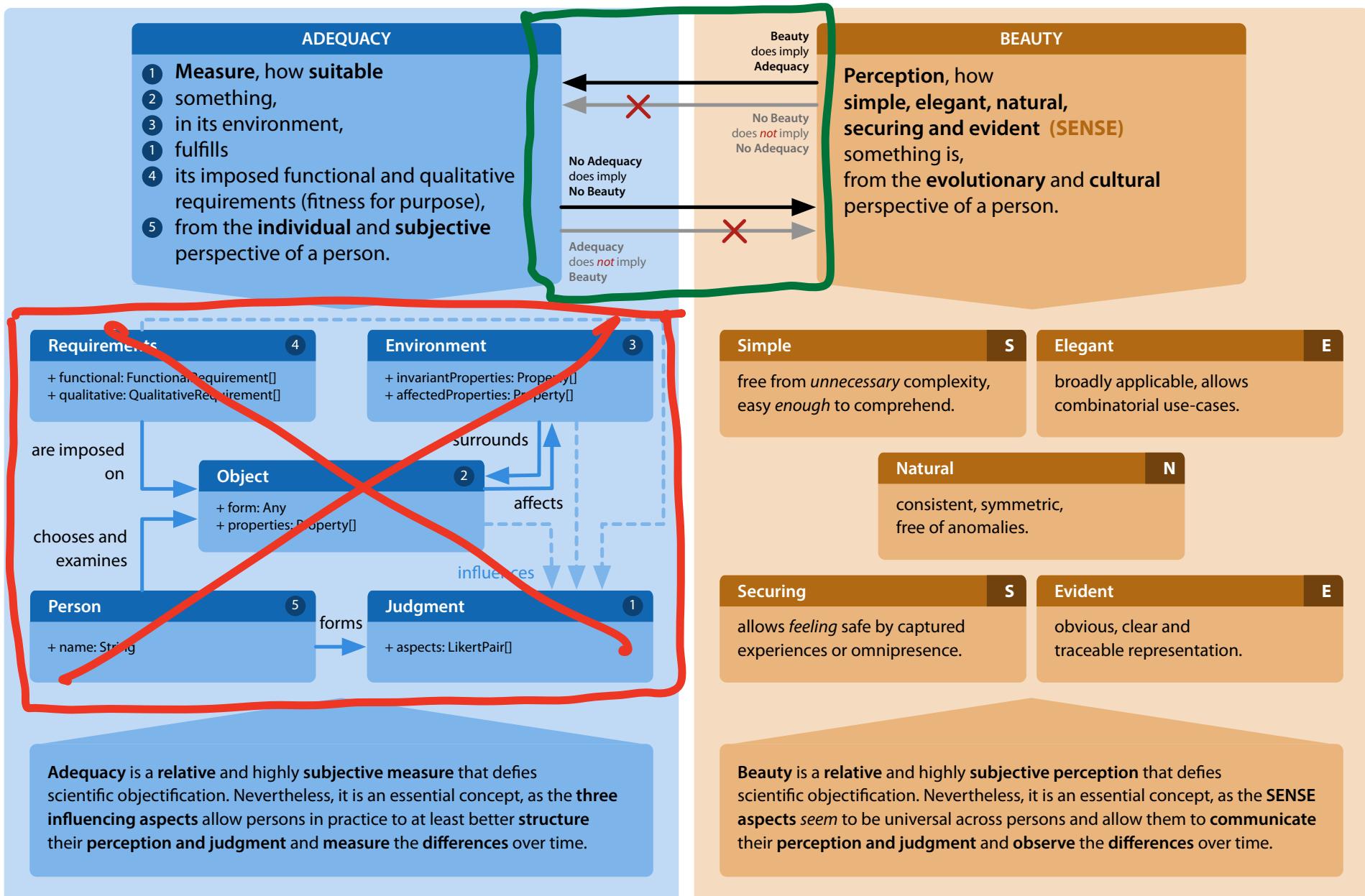
prescriptive

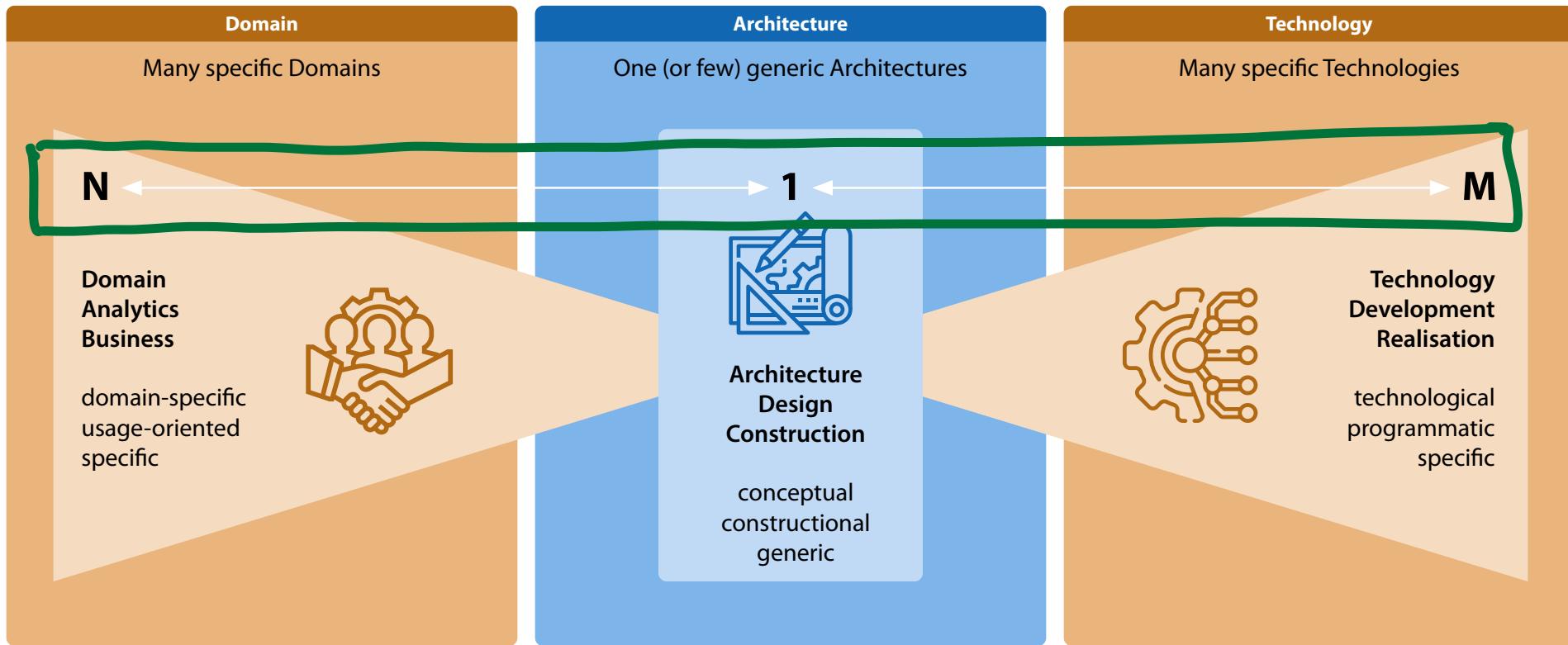
stability,
usefulness,
appearance

creative

Structural Scientific Definition of Architecture:
Architecture of a system is the set of fundamental **concepts** and **properties** of the system in its environment, embodied in its **elements**, **interfaces**, **relationships** and **behaviours** and the principles of its **design** and **evolution**.
(based on industry standard ISO/IEC 42010)

Holistic Artistic Definition of Architecture:
Great architecture is achieved **harmony** and **accord** of all **parts**, where you no longer can add, modify or remove anything without impairing the **whole**.
(based on a quote by Leon Battista Alberti on beauty)





Manifesto for IT Architecture

Continuously Raising the Bar

Mission

As IT Architects we guide the design, implementation and evolution of IT solutions.

Entitlement

We continuously strive to raise the bar of professional IT architecture by practicing it and helping others to learn our craft.
We achieve maximum value for our clients through our work.

Values

Through this work we have come to value aspects of our craft. While we acknowledge the beneficial values in the items on the right, we appreciate the stronger values in the items on the left even more.

Sustainable Concepts

over **Latest Technologies**

Pragmatic Making

over **Theoretical Consideration**

Constructive Craftsmanship

over **Analytical Engineering**

Accredited Creativity

over **Achieved Industrialization**

Proactive Improvement

over **Reactive Correction**

Inherent Quality

over **Tested Robustness**

Operational Delight

over **Useful Functionality**



Complex vs. Complicated

FOCUS

RATIONALE

CHALLENGE

INSIGHT

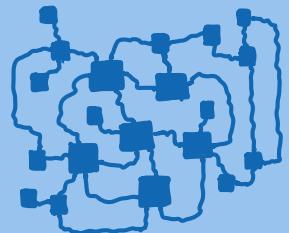
complex

refers to the **extrinsic** and **higher-** or **macro**-level difficulty of a system,

because the system involves many different and **connected** parts

which take time to **comprehend** and **master in total**,

and which nevertheless are **easy** to explain.

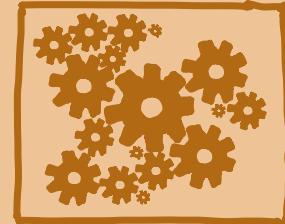
**complicated**

refers to the **intrinsic** and **lower-** or **micro**-level difficulty of a system,

because the system involves many different and **difficult** aspects

which take time to **understand** and **learn in detail**,

and which usually are **hard** to explain.

**NOTICE**

Simple (non-complicated) systems can be **complex**.

NOTICE

Clear (non-complex) systems can be **complicated**.

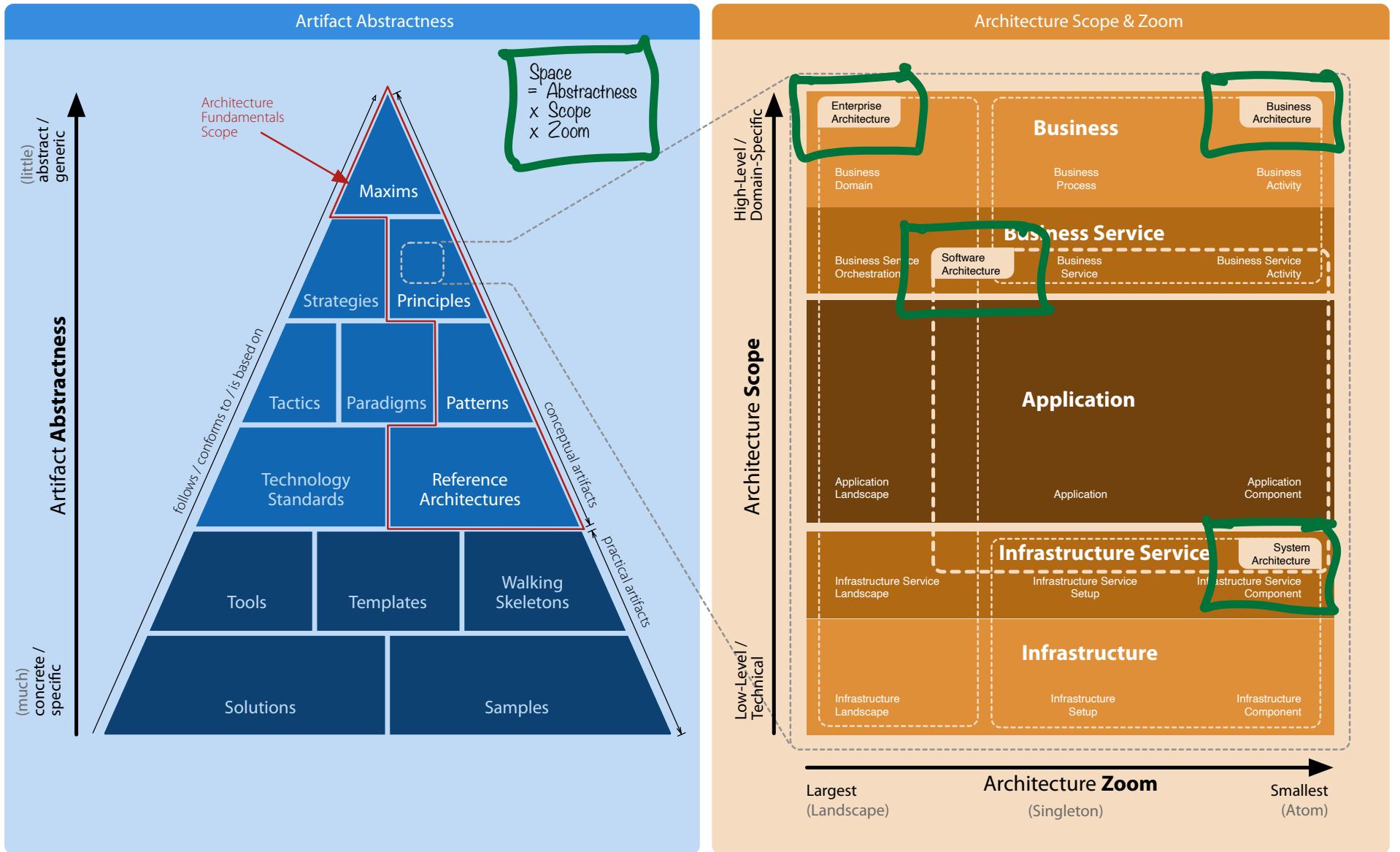
RECOGNIZE

Architecture primarily has to master the **complex** aspects of a system.

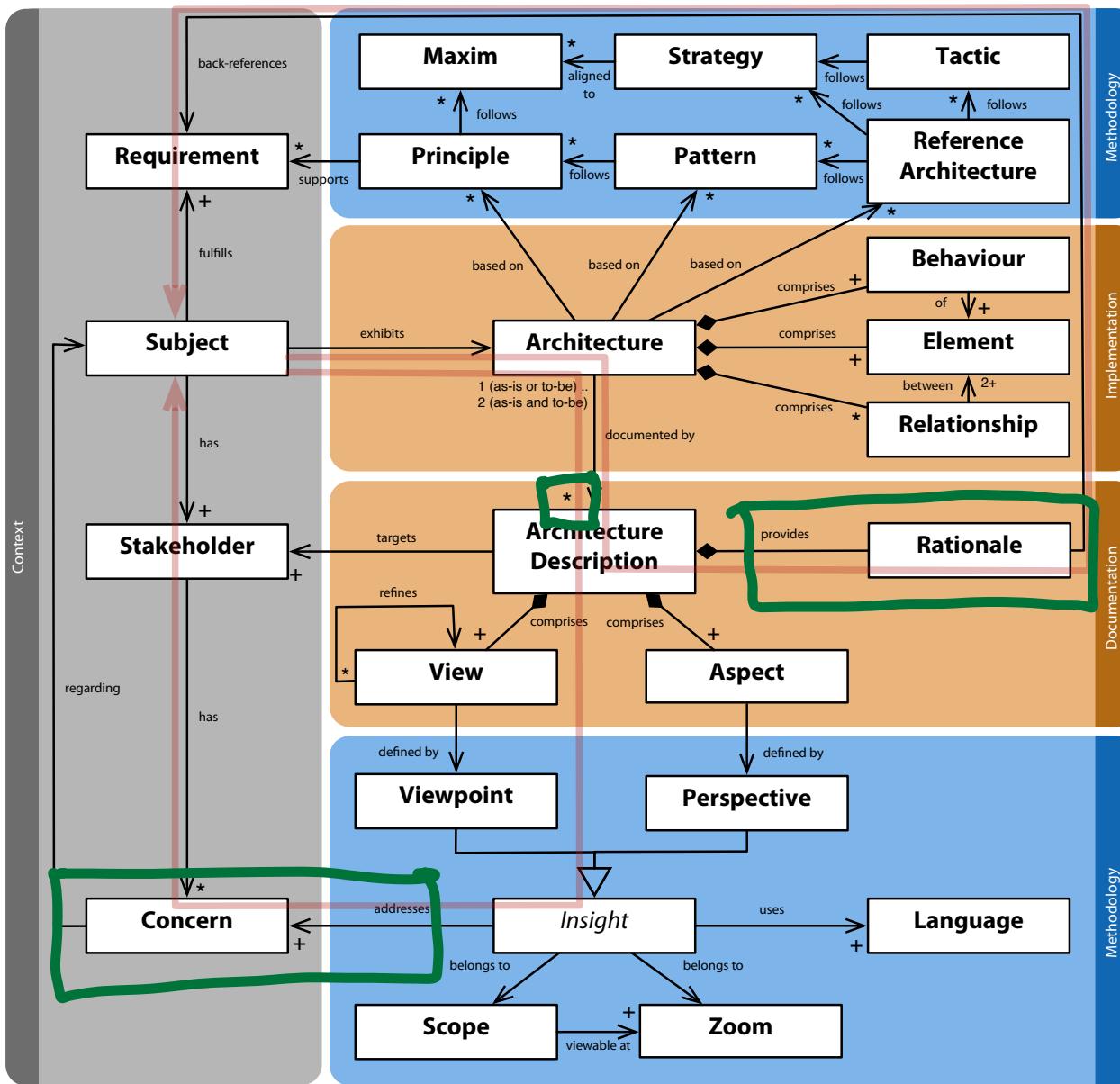
RECOGNIZE

Development primarily has to master the **complicated** aspects of a system.

Architecture Space



Architecture Ontology

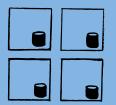
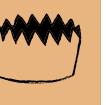
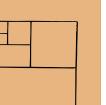
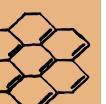
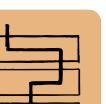
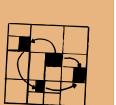
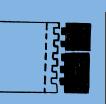


Requirement: A functional or non-functional demand or imposed obligation on the Subject.	Architecture: Inherent static and dynamic structure of a Subject which comprise Elements, the visible Behaviour of Elements and Relationships between Elements.	Maxim: Fundamental, generally valid set of values and rules to guide the architecture discipline (think: law).
Subject: Any type of business process, business service, software application, infrastructure service or infrastructure set up.	Element: Fundamental part from which a Subject can be considered to be constructed.	Strategy: Long-term situation-independent plan of approaches to achieve a particular goal (think: war). Aligned and not in conflict with any Maxim.
Stakeholder: Person, group or entity with an interest in or Concerns about the Architecture.	Behaviour: Run-time characteristic of Architecture Elements of the Subject.	Tactic: Short-term situation-dependent plan of actions to achieve a particular goal (think: battle). Following and supporting a Strategy.
Concern: Requirement, objective, intention, or aspiration a Stakeholder has on an Architecture.	Relationship: Static or dynamic relationship between Elements of an Architecture of the Subject.	Principle: Fundamental truth, rule, tenet or policy an Architecture follows.
Architecture Description: Set of artifacts that document an Architecture in a way Stakeholders can understand and ensures their Concerns are met.	Rationale: Fundamental reasons for a particular chosen Architecture, usually strongly based on non-functional Requirements.	Pattern: Proven recurring theme, structure, approach or behavior an Architecture and its Elements can follow.
Viewpoint: Collection of templates and conventions for constructing one type of View. It addresses Concerns and contains guidelines for constructing a View.	View: Representation of one or more structural aspects of an Architecture that illustrates how the Architecture addresses one or more Concerns.	Reference Architecture: Reusable proven Architecture template, based on a set of Patterns and following one or more Tactics and Strategies.
Perspective: Collection of decisions, guidelines and rules that ensure a Subject exhibits a set of non-functional Requirements, considered across a number of Views.	Aspect: Representation of one or more non-functional aspects of an Architecture that illustrate how the Architecture addresses one or more Concerns.	Scope: Primary area of the Architecture. Have an Insight address, is either Business, Business Service, Application, Infrastructure Service or Infrastructure.
Insight: Supernormal abstract concept to address particular Concerns through Viewpoints and Perspectives.	Language: Formal, semi-formal or even prose language to describe an Insight, so it optimally addresses the Concerns of Stakeholders.	Zoom: Level of view distance to the Architecture an Insight has. Also known as the Insight detail level, ranging from smallest atoms, over singletons to the largest landscape.

Architecture Maxims

Business Drives	BD	Component Orientation CO	Separation of Business and Technology BT	Adequate Description AD
Trigger and support the business with technological feasibilities, but always understand the business domain and its demands and align your architecture accordingly.	\$	Master complexity in your architecture through stringent bottom-up use of components on all scopes and zoom-levels, loose coupling between and strong cohesion within components.	Strictly separate the business, i.e., domain-specific, aspects from the technological, i.e., infrastructural, aspects. Furthermore, ensure the explicit visibility of domain concepts.	Provide as much stakeholder-directed architecture description as necessary, and as little as possible.
Use-Case Driven Design	UC	Analytical and Creative Act AC	Balance Principles Against Requirements PR	Insights through Views & Aspects VA
Design is how it works and runs, so support your customers in their daily work by directly designing your architecture along their domain-specific use-cases.	1 2 3	Recognize that every good architecture is based on both analytical engineering (structure) and creative artistic (beauty) aspects.	By weighing them against one another, find a reasonable balance between fundamental architecture principles and your particular non-functional requirements.	Give insights into your architecture through carefully selected stakeholder-directed separate views and aspects. Express each with the most suitable graphical or textual language.
Proven Basis	PB	Don't Be Too Clever TC	Design for Failure Case DF	Continuous Compliance CC
Never start an architecture from scratch. Instead start from proven reference architectures, patterns and templates. Even if, after some iterations, no initial content is left.	House	Don't be too clever or tricky, both in your higher-level architecture and lower-level design aspects.	Murphy was an architect: everything which can fail will sometime ultimately fail. Hence, already design for the failure case (think: "pessimistic").	Continuously check through qualitative inspections and quantitative measurements whether your architecture and the non-functional requirements are followed and do not drift apart.
No Silver Bullet	SB	Simplicity Trumps ST	Design to Change DC	Integration-Figure Architect IF
There is no "one-size-fits-all" architecture, so accept that although you should reuse proven architecture aspects as much as possible, you will always need to individualize your designs.	Ball	Create solution parts as simple as possible and only as complex as necessary. And remember: simplicity before generality, use before reuse!	Time changes everything, so your solution is already legacy at the first day of release. Hence, already design for its change (think: "agile").	Recognize that you, the architect, are the central integrating figure, having to bridge between the business and technology spheres of people.
Stepwise Refinement	SR	Perfect is the Enemy of Good Enough GE	Explicit Decisions ED	Eat Your Own Dog-Food OF
Start with the "big picture" and perform a stepwise top-down refinement of your architecture by going from coarse to fine aspects.	Three people	Beware of the perfection pitfall and design your architecture only as good as necessary and not as good as ultimately possible.	Record your major architecture decisions and rationales by taking into account and back-referencing the non-functional requirements.	Theory and practice usually differ. Hence it is vital that every architect has good hands-on experience and must both be able to craft the solution and is willing to hypothetically intensively use it himself.

Architecture Principles

FL Factual Locality	Resources are as spatially and temporally local-scoped to solution components as possible		ES Exclusive Sovereignty	Exclusive resource sovereignty by the enclosing component	
CA Contextual Adequacy	Neither insufficient nor exaggerated solutions for each context		SP Solution-oriented Proportionality	Good expected proportionality in each solution context	
HC Holistic Consistency	Full consistency across all aspects of a solution		SH Structural Homogeneity	Maximum homogeneity in the structure of a solution	
CR Constructional Reusability	High reuse of proven structural components and partial solutions		FS Fulfilled Standards	Compliance to standards as much as possible, as long as the benefits predominate the drawbacks	
FA Functional Abstraction	Suitable level of abstraction across all functional aspects of a solution		FT Functional Traceability	Suitable traceability across all functional aspects of a solution	
CI Communicative Interoperability	Maximum interoperability in communication between solutions		EH Environmental Harmony	Maximum harmony in the integration of the solution with its environment	
AR Avoided Redundancy	Minimum total number of copies of a single resource		MS Minimum Special-Cases	Minimum total number of special-cases in a solution	
LS Logical Separation					
Separation of concerns between the components of a solution					
SM Structural Modularity	Splitting of a solution into manageable structural components				
LC Loose Coupling	Loose coupling in communication and referencing between solution components		SC Strong Cohesion	Strong relationship between functionalities within a single solution component	
OE Open Extensibility	Solution components can be extended by third-parties at fixed interfaces		CC Closed Changeability	Solution components are protected against direct change by third-parties	
UI Unique Identification	Unique identification of all components of a solution		UA Uniform Addressing	Uniform addressing of all components of a solution	
OS Overall Simplicity	All design aspects of a solution are as simple as possible and only as complicated as necessary		EC Encapsulated Complexity	Complex related aspects of a solution are encapsulated into a single responsible component	
LA Least Astonishment	All design aspects of a solution are as little astonishing as possible and only as esoteric as necessary		SD Self Documentation	All design aspects of a solution are preferably self-documenting	
OD Operational Delight	The solution provides users true delight even on long-term operation		AA Artistic Aesthetics	The solution has holistic aesthetics and artistic love in details	

~~Definition of a Component (of a Larger Whole):~~

a know-how encapsulating, potentially reusable and substitutable unit of hierarchical composition with explicit context dependencies, which hides the complexity of its optional behavior and state realization behind small contractually specified interfaces, defines its added value in terms of provided and consumed interfaces and optionally belongs to zero or more categories of similar units.

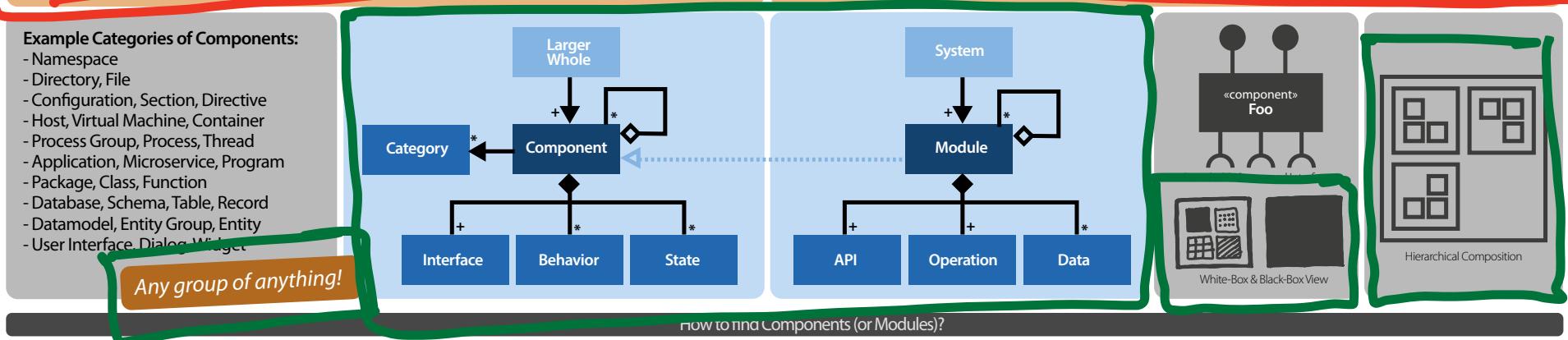
Example Categories of Components:

- Namespace
- Directory, File
- Configuration, Section, Directive
- Host, Virtual Machine, Container
- Process Group, Process, Thread
- Application, Microservice, Program
- Package, Class, Function
- Database, Schema, Table, Record
- Datamodel, Entity Group, Entity
- User Interface, Dialog, Widget

Any group of anything!

~~Definition of a Module (of a System):~~

a know-how encapsulating, potentially reusable and substitutable source code unit of hierarchical composition with explicit context dependencies which hides the complexity of its operation and data implementation behind small contractually specified Application Programming Interfaces (API), defines its added value in terms of provided and consumed APIs and optionally belongs to zero or more categories of similar units.



How to find Components (or Modules)?

DCA Domain Concept Abstraction

Model domain concepts as entity components and then group at higher levels.



SOC Separation of Concerns

Build components for clearly distinct concerns.



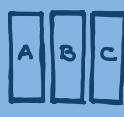
USE Reusability Potential

Decide on components based on their reusability potential.



UCC Use-Case Clustering

Build domain components for each use-case or each logical use-case cluster.



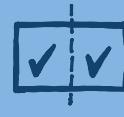
SRP Single Responsibility Principle

Build components for clearly distinct responsibilities.



DCC Divide & Conquer Complexity

Master overall complexity by splitting larger things into smaller things.



DDD Domain-Driven Design

Model domain "Bounded Contexts" through DDD and derive components from them.



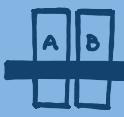
CNC Coupling and Cohesion

Decide on components based on their loose coupling and strong cohesion.



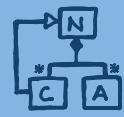
CCC Cross-Cutting Concerns

Build common cross-cutting concerns as cross-cutting components.



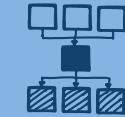
OOD Object-Oriented Design

Model Object-Oriented Design entities (and/or OOP constructs) as components.



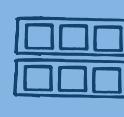
DEP Dependency Encapsulation

Decide on components based on their encapsulation of dependencies.



PAT Architecture Patterns

Build inner components to comply to outer structure, slicing and clustering architectures.



Interface Design

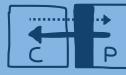
Definition of an Interface

A well-defined shielding and abstracting **boundary** of a passive, providing component, consisting of one or more distinguished, **outside-in** designed **interaction endpoints**, each accessed and controlled by active, consuming components through the **exchange of input/output information** and operating under a certain **syntactical and semantical contract**.

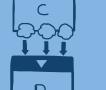
Endpoint: Name, Directive, Command, Function, Method, Procedure, Address, Port, URL, Dialog, ...

Exchange: Option, Argument, ... Return Value, Result, Request/Response Message, Error/Exception, Interaction, ...

Contract: Syntax, Pre-Condition, Invariant, Post-Condition, Side-Effect, Idempotence, Determinism, Functionality, ...

Types of Software Interfaces	
API Application Programming Interface	
Example: foo("bar" , 42) (call and use)	
SPI Service Provider Interface	
Example: register("foo", (x, k) => ...) (extend and implement)	
SCI Startup Configuration Interface	
Examples: INI, Java Properties, TOML, YAML, JSON, XML, etc.	
BPI Batch Processing Interface	
Examples: Unix at(1), Unix ts(1), GNU Batch, Spring Batch, Java Batch, SAP LO-BM, etc.	
CLI Command-Line Interface	
Example: foo -x --bar=baz quux	
GUI Graphical User Interface	
Examples: Windows/WPF, macOS/Cocoa, KDE/Qt, GNOME/GTK	
RNI Remote Network Interface	
Examples: GraphQL-IO, HTTP/REST, SOAP, RMI, WebSockets, AMQP, MQTT, etc.	

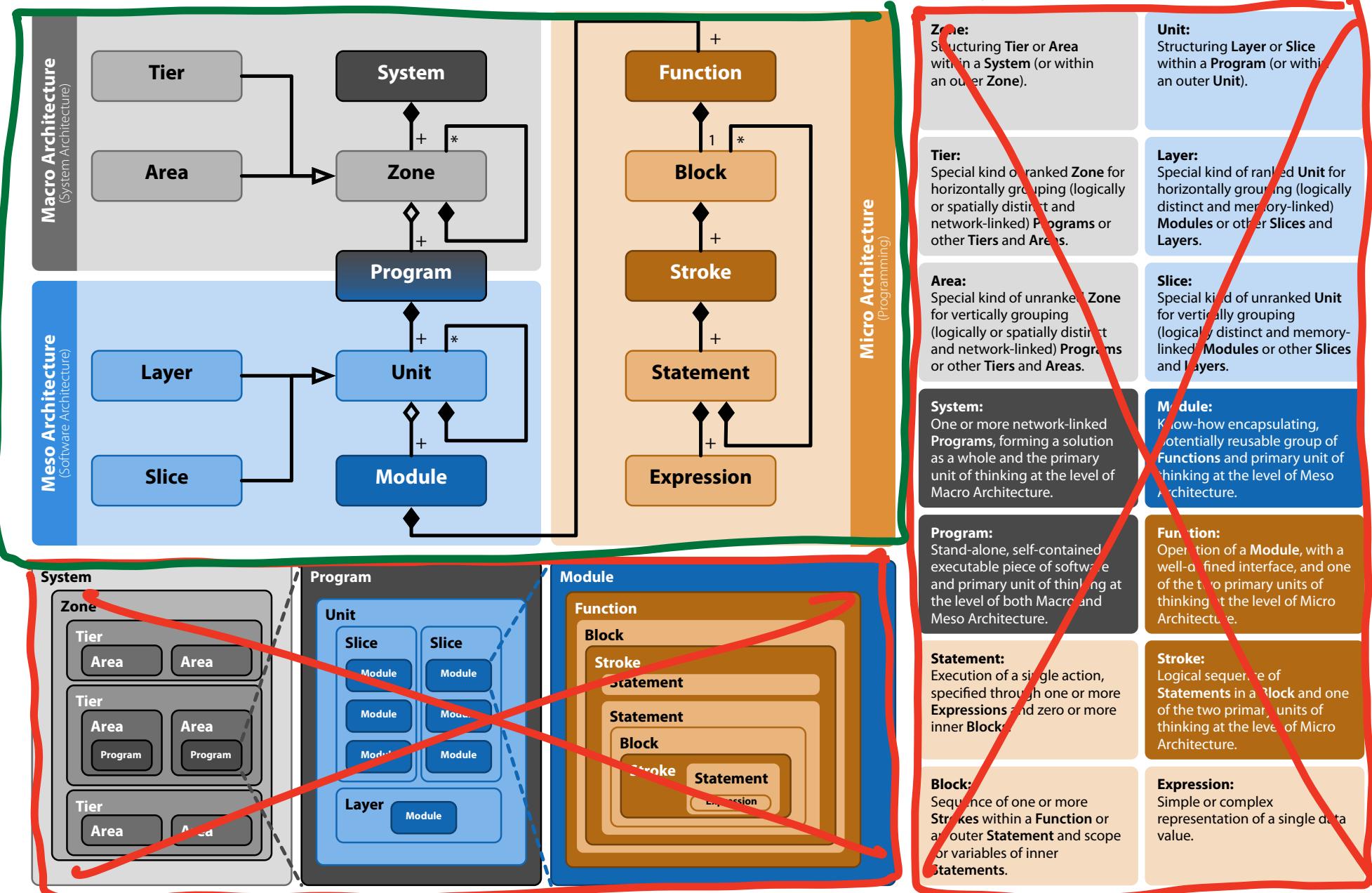
Characteristics of Good Interfaces	
AP Appropriate & Proportional	
Appropriate to consumer requirements, proportional to provider functionality.	
SA Shielding & Abstracting	
Shields from direct access, abstracts and hides implementation details.	
IE Inviting & Expressive	
Invites through "outside-in" design, powerful in expressiveness.	
IF Intuitive & Foolproof	
Intuitive to grasp and use, hard to misuse.	
OC Orthogonal & Concise	
Supports combinatorial use-cases, causes minimum boilerplate.	
TP Tolerant & Predictable	
Tolerant on input, predictable on output.	
EC Extensible & Compatible	
Easy to extend for providers, backward/forward-compatible for consumers.	

Selected Interface Design Patterns	
IVF Interface Version & Features	
Provide version and feature information for algebraic comparison and feature detection.	
2LF Leaky Two-Layer Facade	
Provide higher-level convenient use-case and lower-level orthogonal feature interface.	
EVE Event Emitter	
Emit events to previously registered, interested consumers.	
CTX Multi-Context	
Use contexts to distinguish between different usage scenarios and to carry common info.	
CEF Configure-Execute Flow	
Spread use-cases onto a flow of configuration exchanges and a final executional exchange.	
IOC Inversion Of Control	
Invert control of asynchronous operations via callbacks, promises or async. mechanisms.	
HMR Human/Machine Responses	
Support humans and machines in outputs through both description and parsing-free info.	

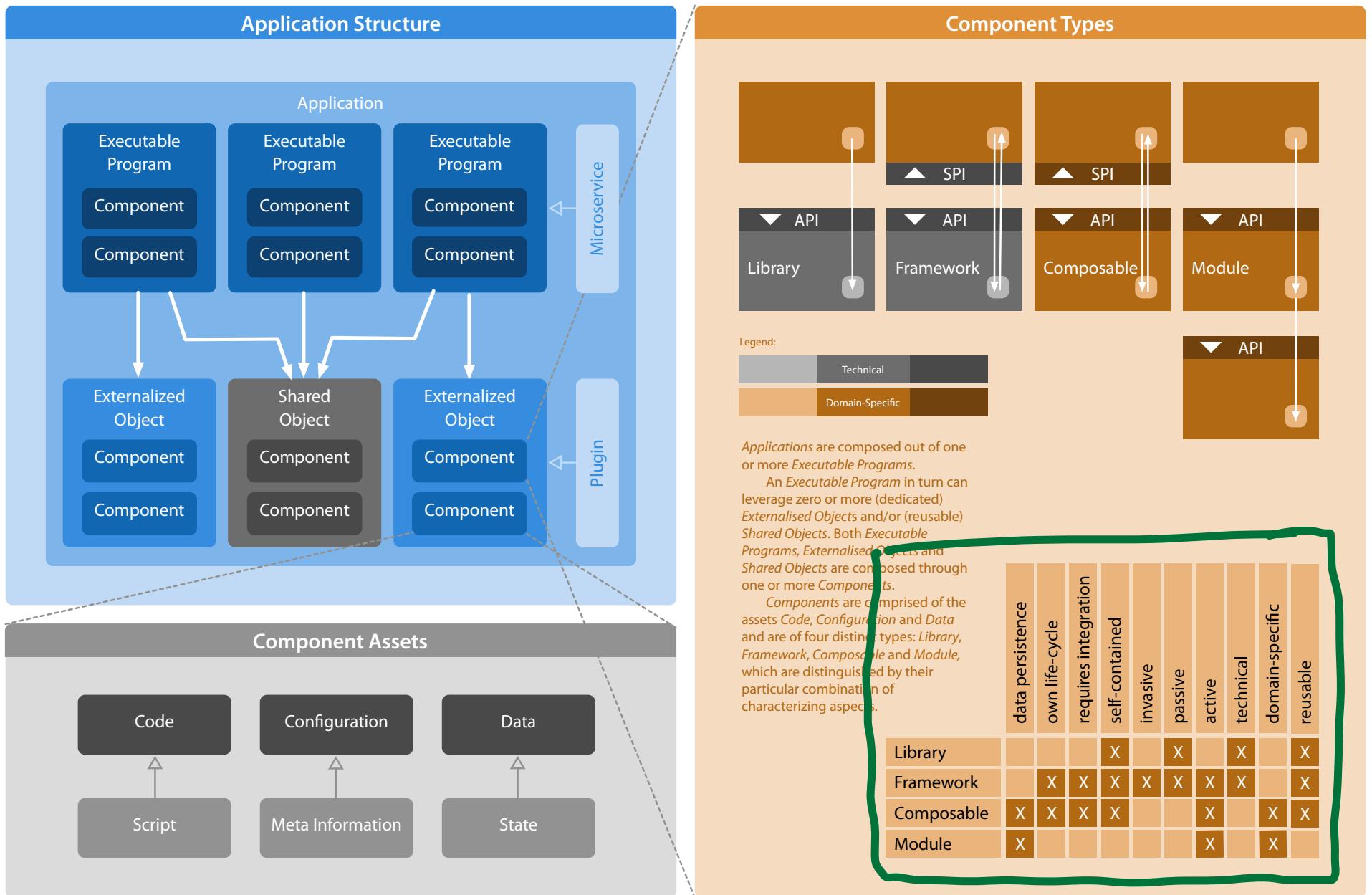
AF 05.2

Intellectual Content: Version 1.05 (2022-1-95). Authored 20-5-2022 by Dr. Ralf S. Engelschall, <http://engelschall.com>. All Rights Reserved. Unpublished. Reproduction is prohibited without written permission from the Technische Universität München (TUM) for educational purposes only.

Component Hierarchy

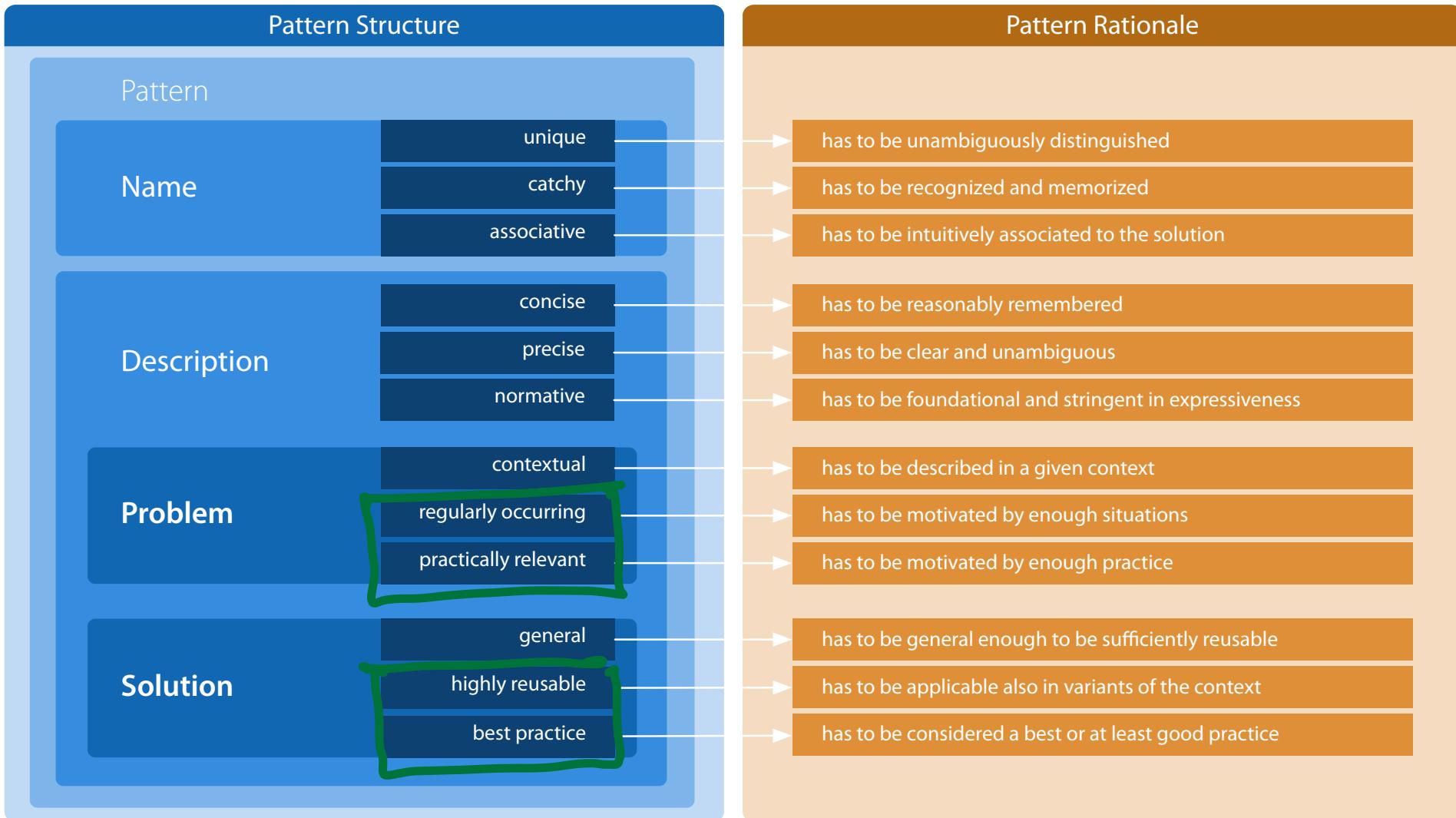


Application Composition



Pattern Definition

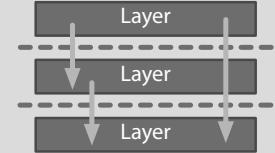
Pattern: unique, catchy, and associative **Name** and concise, precise, and normative **Description** of a contextual, regularly occurring, and practically relevant **Problem** and a general, highly reusable, and best practice **Solution** for it.


AF 06.1


Layering Principle

Horizontally split code or data into two or more logically, optionally also spatially, clearly distinct, isolating, named, and ranked layers.

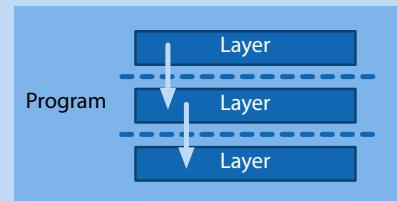
A Layer is not allowed to have relationships to or knowledge about any upper Layers. Additionally, for *Closed Layering*, each Layer is allowed to have relationships to and knowledge about any directly lower Layer only. In contrast to *Open Layering* or *Leaky Abstraction*, where each Layer is allowed to have relationships to and knowledge about any lower Layer.



LR Layer

Split related code or data of a Program into two or more logically distinct domain- or technology-induced Layers.

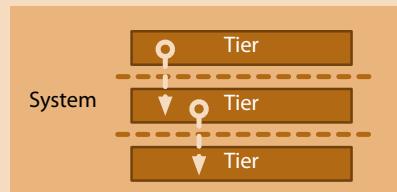
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction.



TR Tier

Split related code or data of a System into two, three or more logically and spatially distinct, network-connected, domain- or technology-induced Tiers.

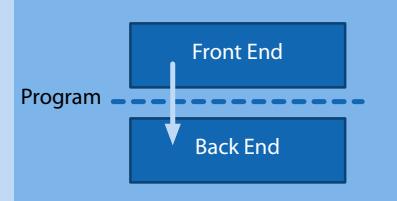
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Deployment Partitioning.



FB Front End / Back End

Split the code of a Program into exactly two logical Layers: a user-facing Front End and a data-facing Back End.

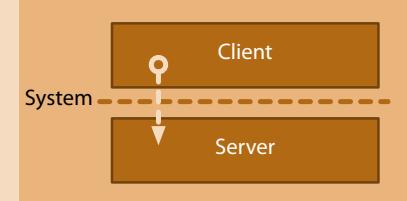
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Organisational Alignment.



CS Client / Server

Split the code of a System into two spatially distinct, network-connected Layers, each forming a stand-alone Program: a user-facing and multi-instantiated (Rich) Client and a data-facing (and logically) single-instantiated (Thin) Server. Both contain a Front/Back End.

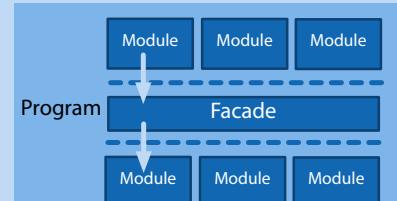
Rationale: Multi-User, User Computing, Resource Leverage, Distributed Computing.



FD Facade

Splice a domain-specific Facade Layer into two Layers of two or more Modules. The extra Facade Layer acts as a broker between the Modules.

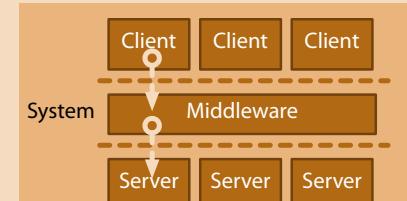
Rationale: Information Hiding, Cross-Cutting Concern Centralization, Functionality Orchestration, Authorization, Validation, Conversion.



MW Middleware

Splice a domain-unspecific Middleware Layer into a Client/Server communication. The extra Layer is a stand-alone Program Tier and acts as a broker between Client and Server.

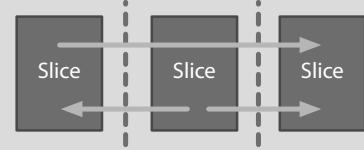
Rationale: Communication Peer Discovery Simplification, Transport Protocol Conversions, Network Topology Flexibility.



Slicing Principle

Vertically split code or data into two or more logically, optionally also spatially, clearly distinct, named, and unranked slices.

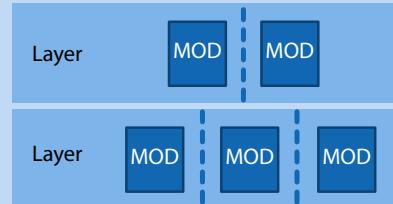
The particular slicing should minimize the total amount of individual relationships between the resulting slices. Per type of relationship, there should be no cycle in the transitive relationships.



MOD Concerned Module

Split related code or data (usually across a single Layer) into two or more logically distinct domain- or technology-induced Modules.

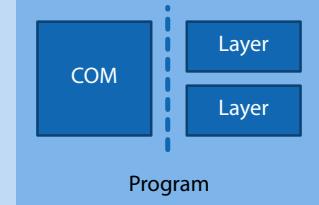
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity.



COM Common Slice

Factor out common or cross concern code or data of a Program (across all Layers) into a single spatially distinct, separate slice.

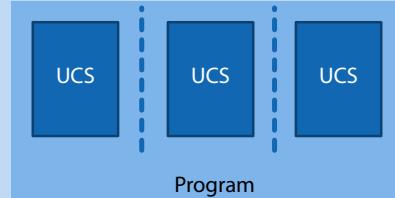
Rationale: Lack of Redundancy, Single Point of Truth, Reusability.



UCS Use-Case Slice

Split the code and data of a Program (across all Layers) into two or more purely logical slices, one for each distinct, domain-specific Use-Case.

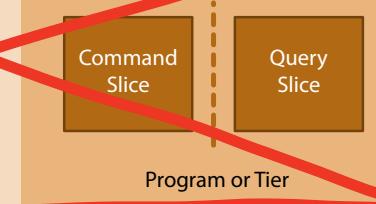
Rationale: Comprehensibility, Domain Alignment, Mastering Complexity.



CQRS Command-Query Responsibility Segregation

Split code and data of a Program (across all Layers) or a Tier into exactly two slices to segregate operations that read data (queries) from the operations that update data (commands).

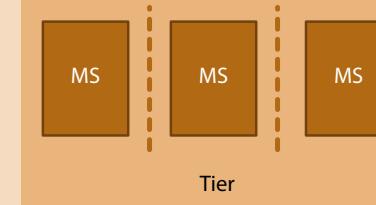
Rationale: Separated Scalability, Separated Data Access Patterns, Event Sourcing Approach.



MS Microservice

Split code and data of a Tier (across all Layers) into two or more distinct, loosely-coupled, domain-enclosed, functional services, each forming a stand-alone Program.

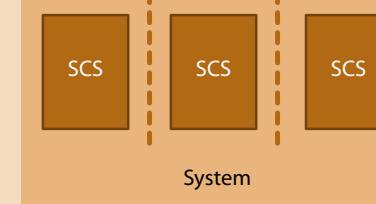
Rationale: Heterogeneity, Long-Term Large-Scale Maintenance, Replaceability, Resilience, (Scalability), (Easy Deployment), (Organizational Alignment), (Composability), (Reusability).



SCS Self-Contained System

Split code and data of a System (across all Layers and Tiers) into two or more distinct, loosely-coupled, domain-enclosed, functional systems, each forming a stand-alone sub-System.

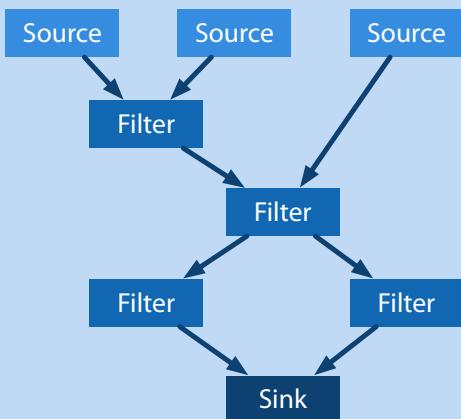
Rationale: Mastering Complexity, Heterogeneity, Resilience, Scalability, Easy Deployment, Organizational Alignment, Reusability, Replaceability.



Pipes & Filters

Pass data through a directed graph of **Components** and connecting **Pipes**. The components can be **Sources**, where data is produced, **Filters**, where data is processed, or **Sinks**, where data is captured. Source and Filter components can have one or more output Pipes. Filter and Sink components can have one or more input Pipes. Components are independent processing units and operate fully asynchronously.

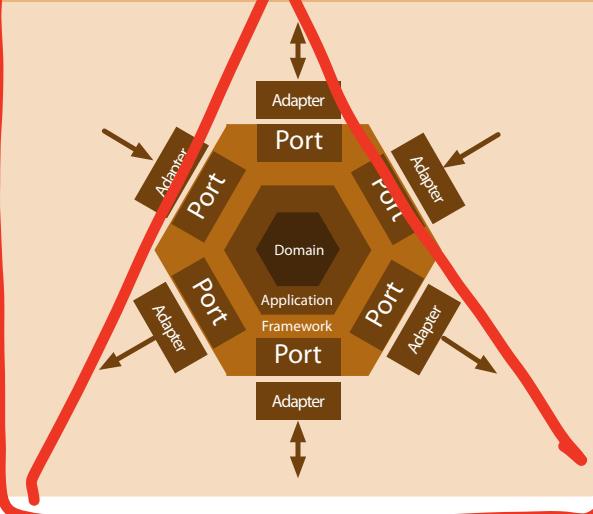
Examples: Unix commands with stdin/stdout/stderr and the Unix shell connecting them with pipes; Apache Spark or Apache Camel data stream processing pipelines.



Ports & Adapters (Hexagonal)

Perform communication in a Hub & Spoke fashion by structuring a solution into the three "Layers" **Domain**, **Application** and **Framework** and use the Framework layer to connect with the outside world through **Ports** (general Interfaces) and **Adapters** (particular Implementations). Often some Ports & Adapters are user-facing sources and some are data-facing sinks, although the motivation for the Ports & Adapters architecture is to remove this distinction between user and data sides of a solution.

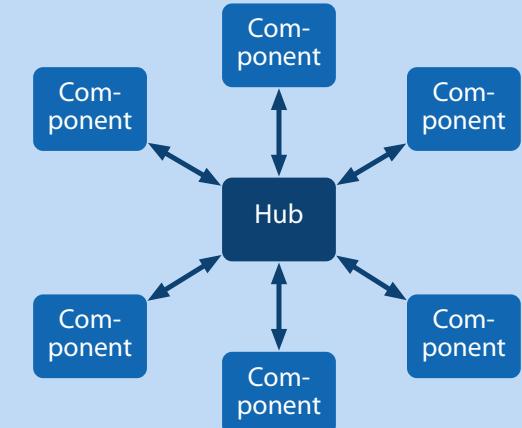
Examples: Message Queue, Enterprise Service Bus or Media Streaming Service internal realization.



Hub & Spoke

Perform communication (the **Spoke**) between multiple Components through a central **Hub** Component. Instead of having to communicate with $N \times (N-1) / 2$ bi-directional interconnects between N Components, use the intermediate Hub to communicate with just N interconnects only. Sometimes one distinguishes between K ($0 < K < N$) source and $N - K$ target Components and then $K \times (N - K)$ uni-directional interconnects are reduced to just N interconnects, too.

Examples: Message Queue, Enterprise Service Bus, Module Group Facade, GNU Compiler Collection, ImageMagick, etc.

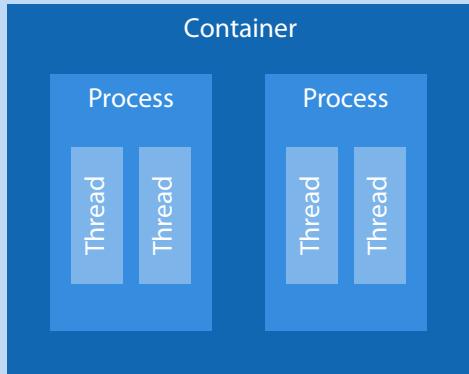


Container, Process, Thread

The Operating System manages and orchestrates the run-time execution of applications in **Containers**, programs in **Processes** and control flows in **Threads**.

Containers are the ultimate enclosures, separating and controlling both the computing resources processor, memory, storage and network. Processes are the primary enclosures, still separating and controlling at least the computing resources processor and memory. Threads are the light-weight enclosures, just separating and controlling the computing resource processor. Containers can contain one or more Processes, and Processes can contain one or more Threads.

Examples: Docker Container, Unix Processes, POSIX Threads.

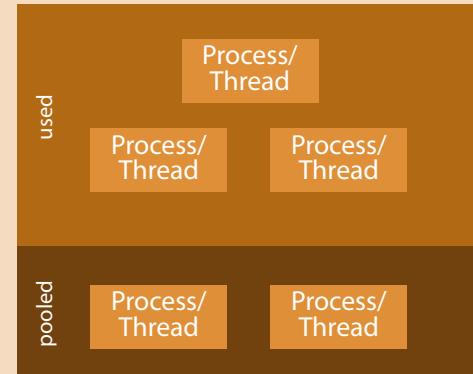


Process/Thread Pool

Instead of creating a Process/Thread for handling each incoming I/O request, pick a pre-created Process/Thread out of a resource **Pool** in order to increase performance and decouple I/O traffic (leading to threads of execution) from the actual computing resource usage and utilization.

The Process/Thread Pool usually has a lower and upper bound of processes/threads. The lower bound keeps the system "hot" between I/O requests. The upper bound limits the computing resource usage and avoids over-utilization.

Examples: Apache HTTP Daemon

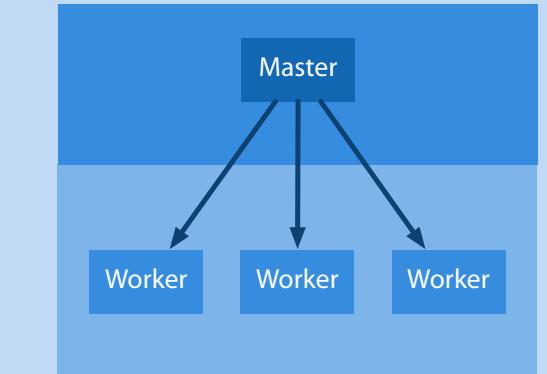


Master-Worker

The system has a single permanent **Master** container/process/thread and a Pool of many ephemeral **Worker** containers/processes/threads. The Master starts, restarts, pauses, resumes and stops the Workers and usually also delegates incoming I/O requests to them. The Workers process the I/O requests and deliver the responses.

Starting the Master usually implicitly starts an initial set of Workers (the initial Pool), stopping the Master implicitly stops all still pending Workers.

Examples: Unix init(8) daemon, Apache HTTP Daemon, SupervisorD, Node.js Cluster module

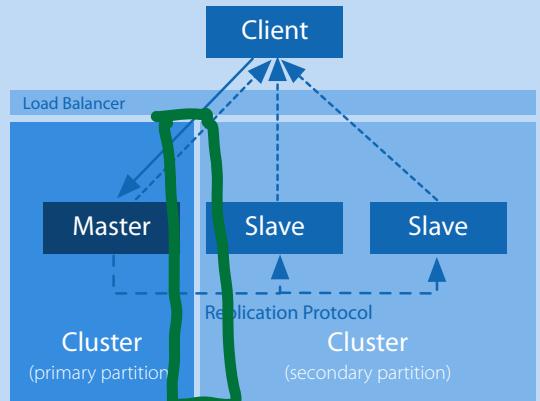


Master-Slave (Static Replication)

Cluster of a single **Master** and multiple **Slave** nodes, where data is continuously copied from the Master to the Slave nodes in order to support high-availability (where a Slave will take over the Master role) in case of a Master outage and increased read performance (where regular read requests are also served by the Slaves).

In this static replication scenario the Master is usually assigned statically and in case of outages has to be reassigned usually semi-manually. Especially, the full reestablishment of the original Master assignment after a Master recovery usually is a manual process.

Examples: OpenLDAP Replication, PostgreSQL WAL Replication.

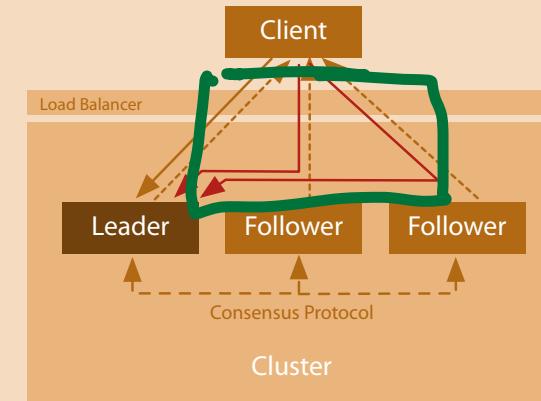


Leader-Follower (Dynamic Replication)

Cluster of a single **Leader** and multiple **Follower** nodes, where data is written on the current Leader node and data read on both the current Leader and all Follower nodes. For writing data to the cluster, the Leader node performs a consensus protocol (e.g. RAFT, Paxos or at least Two-Phase-Commit) with the Followers and this way automatically and consistently replicates the data to the Followers.

In this dynamic replication scenario the Leader is usually automatically assigned by the cluster nodes through an election protocol and in case of outages is automatically re-assigned. There is usually no re-establishment of the original Leader assignment.

Examples: Apache Zookeeper, Consul, EtcD, CockroachDB, InfluxDB.

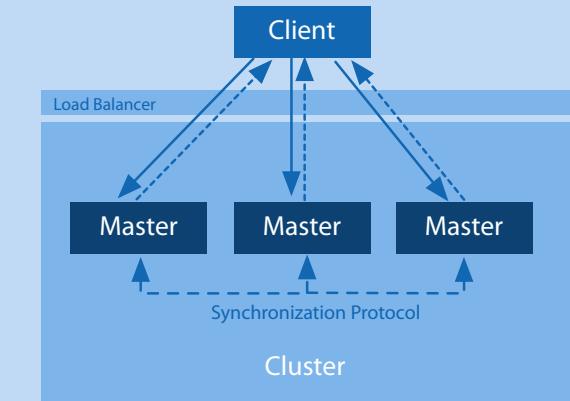


Master-Master (Synchronization)

Cluster of multiple **Master** nodes, where data is read and written on any Master node concurrently. The Master nodes either use Strict Consistency through writing to a mutual-exclusion-locked shared storage concurrently or use Eventual Consistency in a Shared Nothing storage scenario where they continuously synchronize their local data state to all other nodes with the help of a synchronization protocol.

The synchronization protocol usually is based on either Conflict-Free Replicated Data Types (CRDT) or at least Operational Transformation (OT). In any scenario, data update conflicts are explicitly avoided.

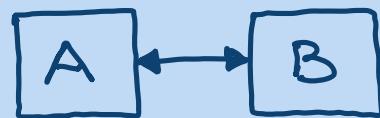
Examples: ORACLE RAC, MySQL/MariaDB Galera Cluster, Riak, Automerge/Hypermerge.



PTP Point-to-Point

Communicate between two network nodes in a point-to-point fashion, usually through a direct link.

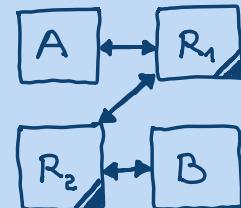
Rationale: simple communication where both nodes know about each other and can directly reach each other.



RTG Routing

Communicate between two network nodes in a point-to-point fashion, but by routing the network packets over intermediate forwarding nodes (routers).

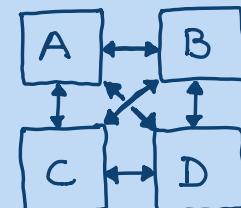
Rationale: simple communication where both nodes know about each other, but cannot directly reach each other.



P2P Peer-to-Peer

Communicate between multiple network nodes (usually all in the client and server role at the same time) without involving a central hub node (in the role of a server) — except for the initial network entry discovery.

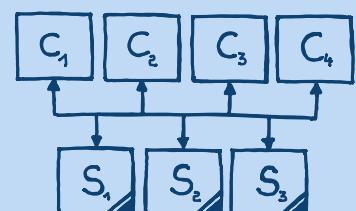
Rationale: communication without central control (although a seed peer is required).



C/S Client/Server

Communicate between multiple nodes in the client role (making requests, and usually with ephemeral addresses) and multiple nodes in the server role (serving responses, and usually with fixed addresses).

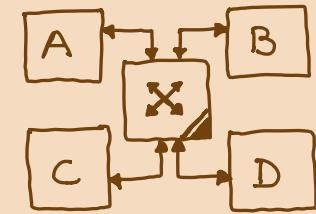
Rationale: communication with central orchestration, control and data storage.



BUS Bus/Broker/Relay

Communicate between multiple nodes with the help of a central packet forwarding hub node in a star network topology.

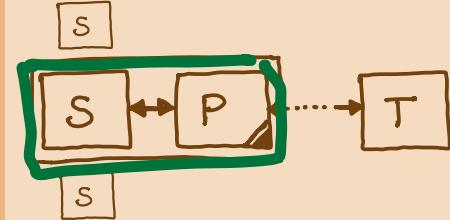
Rationale: decouple communication nodes: instead of Point-to-Point (PTP) communications between all nodes, there are just PTP communications with the hub.



FPR (Forward) Proxy

Communicate between two nodes by using an intermediate forwarding proxy node in front of the source node.

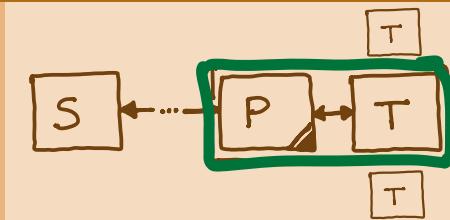
Rationale: bridge network topology constraints (segmented networks); caching at source side; auditing of communication.



RPR Reverse Proxy

Communicate between a source and a target node by using a masquerading proxy node directly in front of the target node.

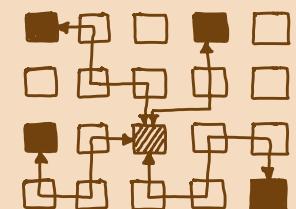
Rationale: load balancing for multiple target nodes; caching at target side; auditing of communication; security shielding of target nodes; protocol conversions.



VPN Virtual (Private) Network

Communicate between nodes in a logical star network topology on top of an arbitrary physical routed network topology.

Rationale: secure private network overlaying an unsecure public network; simplify network topology.



UCT
Unicast (one-to-one)

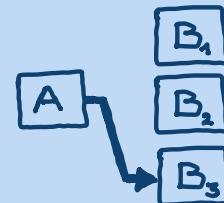
Communicate messages from one source to exactly one destination node. The destination node is explicitly and individually addressed.

Rationale: private communication between exactly two nodes which both know each other beforehand.


ACT
Anycast (one-to-any)

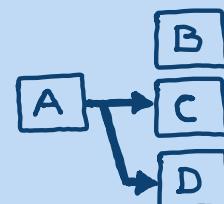
Communicate messages from one source to one of many destination nodes. The picked destination node usually is the network-topology-wise “nearest” or least utilized node in a group of nodes.

Rationale: Unicast, optimized for network failover scenarios, load balancing and CDNs.


MCT
Multicast (one-to-many)

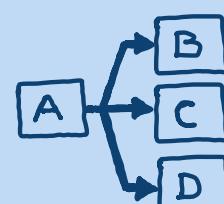
Communicate messages from one source to many destination nodes. The destination nodes usually form a group and are usually not individually addressed.

Rationale: node communication where destination nodes dynamically change or where total traffic should be reduced.


BCT
Broadcast (one-to-all)

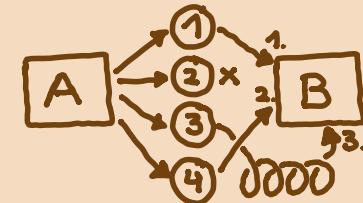
Communicate messages from one source to all available destination nodes. The destination nodes usually are implicitly defined by the extend of the local communication network segment.

Rationale: spreading out messages to all available nodes for potential responses.


DGR
Datagram (Single Packet)

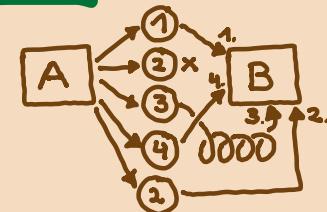
Communicate messages as an unordered set of single packets, usually without any network congestion control, retries or other delivery guarantees.

Rationale: simple low-overhead communication without prior communication establishment (handshake).


STR
Stream (Sequence of Packets)

Communicate messages as an ordered sequence (stream) of packets, usually with network congestion control, retries and delivery guarantees (at-most-once, exactly-once, at-least-once).

Rationale: reliable communication between nodes.


PLL
Pull (Request/Response, RPC)

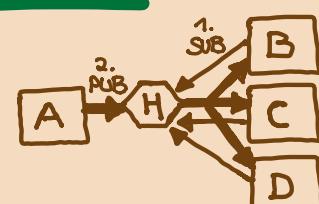
Communicate by performing a request (from the client node) and pulling a corresponding response (from the server node).

Rationale: Remote Procedure Call (RPC) like Unicast or Anycast communication.


PSH
Push (Publish/Subscribe, Events)

Communicate by “subscribing” to “channels” of messages (on one or more receiver nodes or on an intermediate hub) once and then publishing events to those “channels” (on the sender node) multiple times.

Rationale: event-based Multicast or Broadcast communication.



Data Structure Types

Scalar, Atom, Primitive Type
Plain integer or real number, single character or character string, not indexed and (for string only) accessed in O(1) by character position.

Tuple, Object, Structural Type, Record
Ordered, fixed-size sequence of scalar elements, each of individual type, indexed by name and accessed in O(1) by element name.

Sequence, Array, List
Ordered sequence of elements, each of same type, indexed by position and accessed in O(1) or O(n) by element position.

Set, Bag, Bucket
Unordered set of elements, each of same type, not indexed and accessed in O(1) or O(n) by element reference.

Map, Hash, Associative Array
Unordered sequence of elements, each of same type, indexed by (scalar) key and accessed in O(1) by key.

Graph, Nodes & Edges
Unordered set of linked elements (nodes), each of individual type, indexed by (scalar) key and accessed in O(1) by key or by following a directed link (edge).

Data Evolution Approaches

In-Place Editing
Modify data through direct in-place editing, overwriting the previous revision.

Stacking Revisions
(highlighted) Modify data through stacking revisions, preserving all previous revisions. Latest revision is always on top of stack.

Structural Difference
Modify data through merging, journaled domain-unspecific structural differences.

Operational Transformation (OT)
Modify data through applying journaled, domain-specific operational transformations.

Data Sharing Approaches
 Event Sourcing & CRDT
Share data as a chronological sequence of data change events from which the data states can be (re)constructed. Optionally, use a Conflict-Free Replicated Data-Type (CRDT) protocol for the change events.

Ref.-Counting & Copy-on-Write
Share data between resources by using reference-counted data chunks, duplicating a chunk and resetting its reference count to one on write operations only and destroying a chunk only if the reference count drops to zero.

Data Store Types

Key-Value Store
Storage of values in an unordered manner, indexed and queried by key.
Redis, Riak, Memcached, RocksDB, LevelDB

Large-Object Store
Storage of unstructured binary-large object (BLOB) data and its associated meta-data, indexed and queried by unique id.
Minio, SeaweedFS, AWS S3

Triple Store
Storage of subject-predicate-object triples, indexed and queried by subject/predicate/object values and example triples.
Redshift, Virtuoso

File-Tree Store
Storage of unstructured data as named files in a directory tree, indexed and queried by name path from root directory to leave file.
ZFS, XFS, UFS2, APFS

Graph Store
Storage of values as vertices and edges in a graph, both optionally referencing associated key/value pairs. Indexed and queried by key/value pairs and traversed by following edges.
Neo4J, OrientDB, ArangoDB

Document Store
Storage of structured "documents", indexed by id and key/value fields and queried by id and example documents.
MongoDB, CouchDB, RethinkDB

Relational/Table Store
Storage rows of fixed-size, typed value columns, indexed and queried by column values.
PostgreSQL, MariaDB, SQLite, H2, ORACLE DB, IBM DB2

Full-Text Store
Storage of unstructured text, indexed and queried by content words.
ElasticSearch, Solr, Groonga

Wide-Column Store
Distributed storage of rows of sparse (often untyped) value columns, indexed and queried by column values.
Cassandra, Memcached, HBase, ScyllaDB

Time-Series Store
Storage of integer or real values (y-axis) of a time-series (x-axis) into a fixed-size storage format in a round-robin manner where older values are increasingly aggregated (leading to lower resolutions at older times) and finally overwritten.
InfluxDB, MetricTank, RRDTool

DataVault Store
Long-term historical storage of foreign, arbitrary relational data in a fixed schema of hubs, links and satellites, indexed and queried for analysis and reporting purposes.
DataVault 2.0

BlockChain Store
Storage of values in an unordered manner within information blocks which are cryptographically chained through their hash values and distributed in a peer-to-peer way.
Ethereum, Quorum, Tendermint, Hyperledger

Data Guarantees

CAP (Trade-In)

A distributed data store cannot provide more than two out of three guarantees: Consistency (C), Availability (A), Partition-Tolerance (P). So, it has to choose between Consistency (CP) and Availability (AP) when a network partition or failure happens.



BASE (NoSQL)

The semantics (usually of NoSQL systems) of (B)asically (A)vailable, (S)oft state, and (E)ventual consistency. BASE systems favor Availability over Consistency in the CAP-context.



ACID (RDBMS, NewSQL)

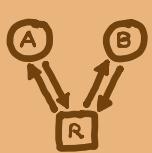
The four guarantees provided in parallel (usually by RDBMS and NewSQL systems): Atomicity, Consistency, Isolation and Durability. ACID systems usually favor Consistency over Availability in the CAP-context.



Data Access

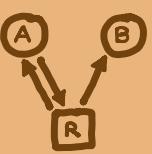
Shared Read/Write

Shared access to data for both read and write operations. Example: Multiple threads on heap or Master-Master database setup.



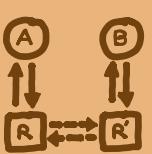
Shared Read / Exclusive Write

Shared access to data for read operations and exclusive access (via a single "owning" component) to data for write operations. Example: RDBMS Master-Slave cluster with shared storage.



Shared Nothing

No shared access to data at all for both read and write operations. Example: Leader-Follower setup with RAFT consensus where Leader writes data only.



Data Access Grouping

Transaction

Protect a sequence of operations from interim exceptions by bracketing the operations in a technical transaction (ensuring that either all or none of the operations succeed).



Compensation

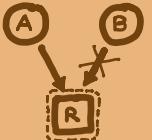
Protect a sequence of operations from interim exceptions by undoing the already succeeded operations through domain-specific compensating (reverse) operations.



Data Consistency

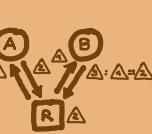
Exclusive Locking (Mutex)

Protect data from concurrent access and resulting inconsistencies with a mutual exclusion lock (mutex) which allows just a single peer to access the data at a time.



Optimistic Locking

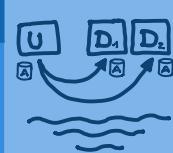
Protect data from concurrent access and resulting inconsistencies by taking note of a revision number or content hash during read operations and checking that this information has not changed before writing the data.



Data Spreading & Aggregation

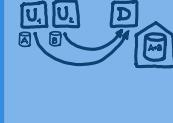
Data River (1-to-N)

A real-time fan-out replication of data from a single upstream/source data repository to multiple downstream/target data repositories.



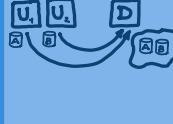
Data Mart (N-to-1), ODS

A massive sized, easily accessible data repository for storing "big data" from many upstream sources in a (real-time) and structured way and with knowing the actual subsequent analysis usage.



Data Lake (N-to-1), Cache

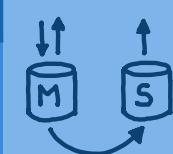
A massive sized, easily accessible data repository for storing "big data" from many upstream sources in a (real-time) semi-structured way and without knowing the actual subsequent usage.



Data Transfer

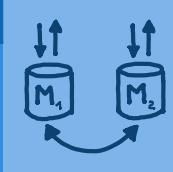
Replication

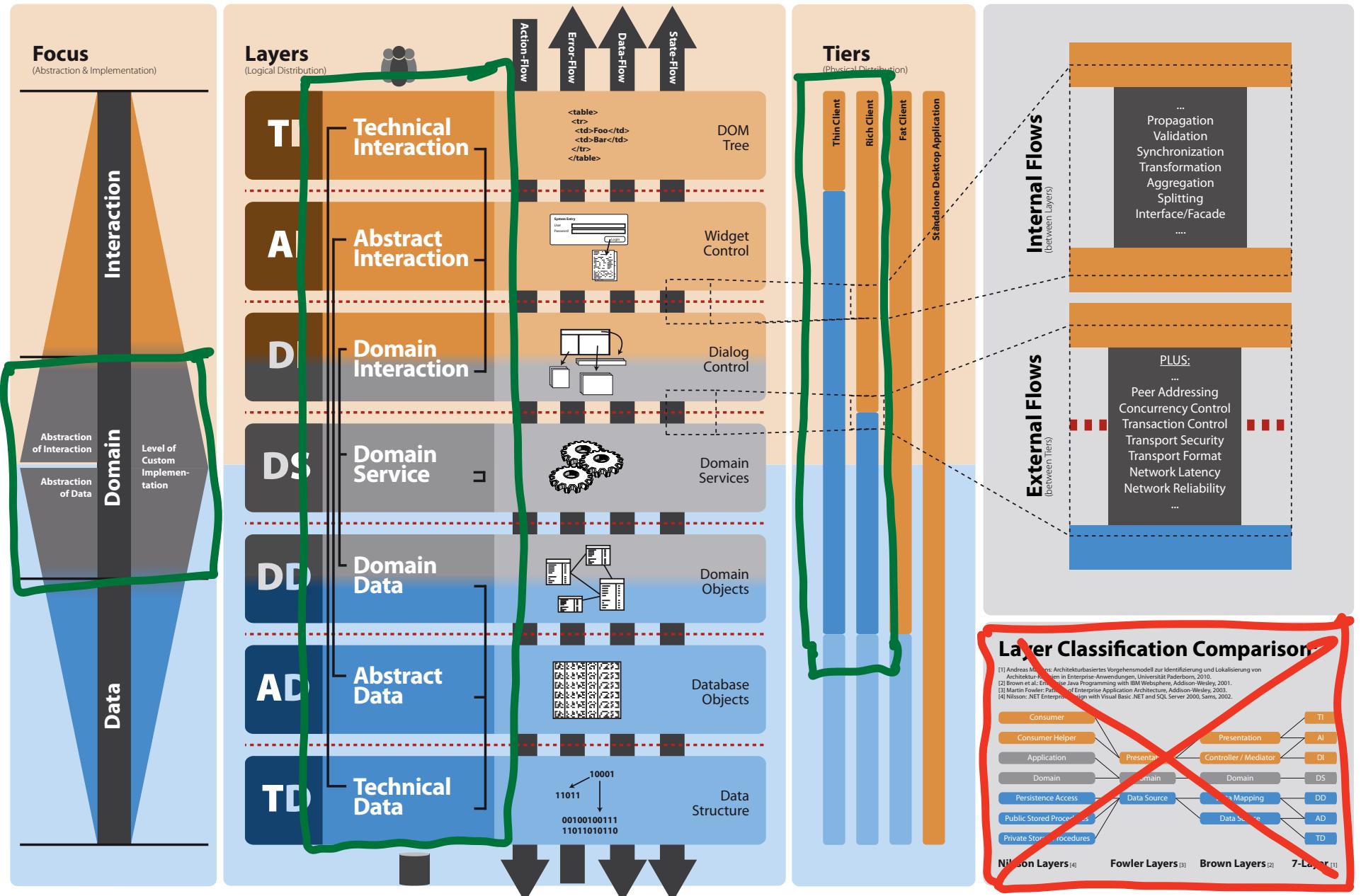
Continuously stream or regularly copy data from a master system to one or more slave systems in order to either read the data from slave systems faster or have slave systems available as a fallback/back-up in case of a failure of the master system.

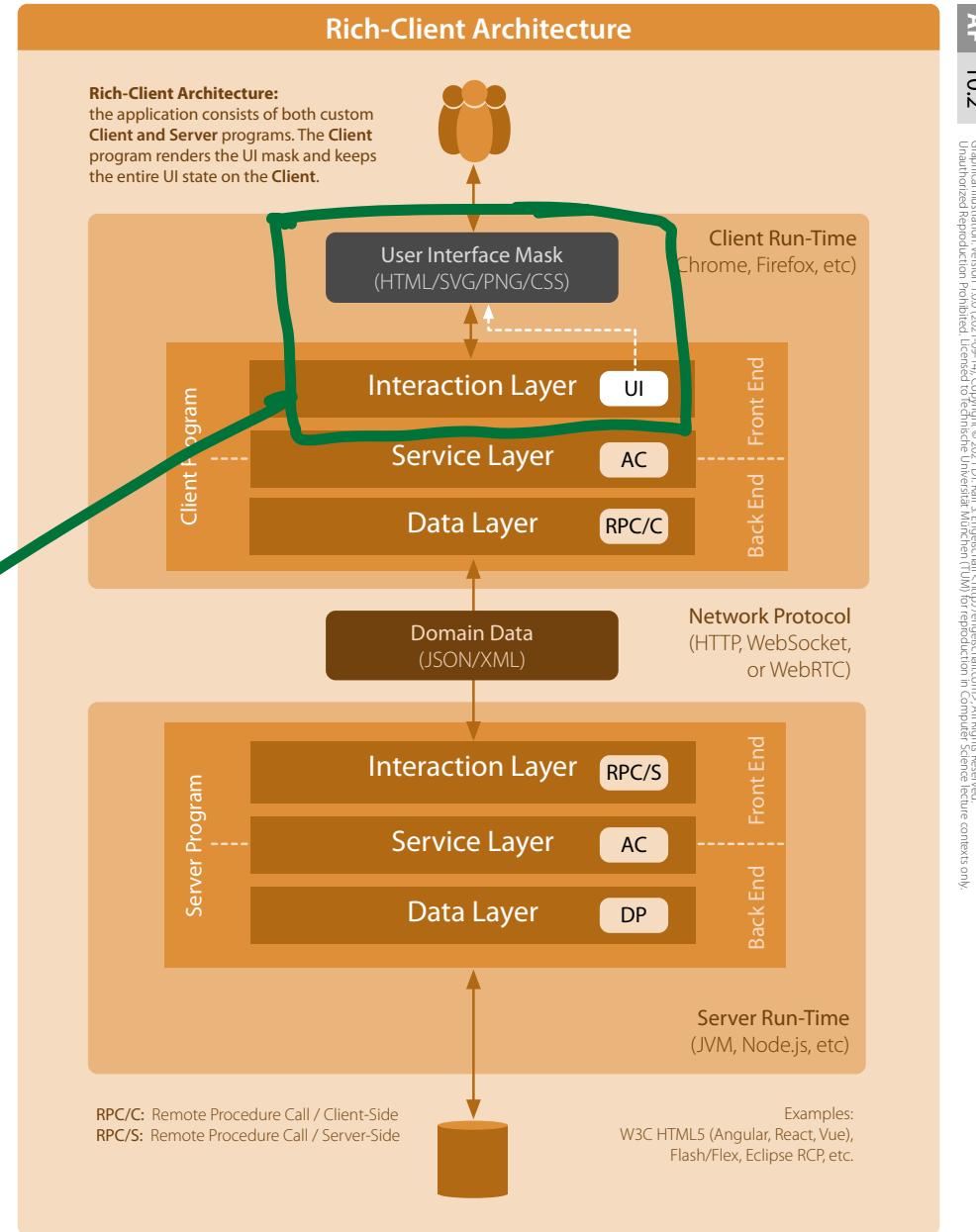
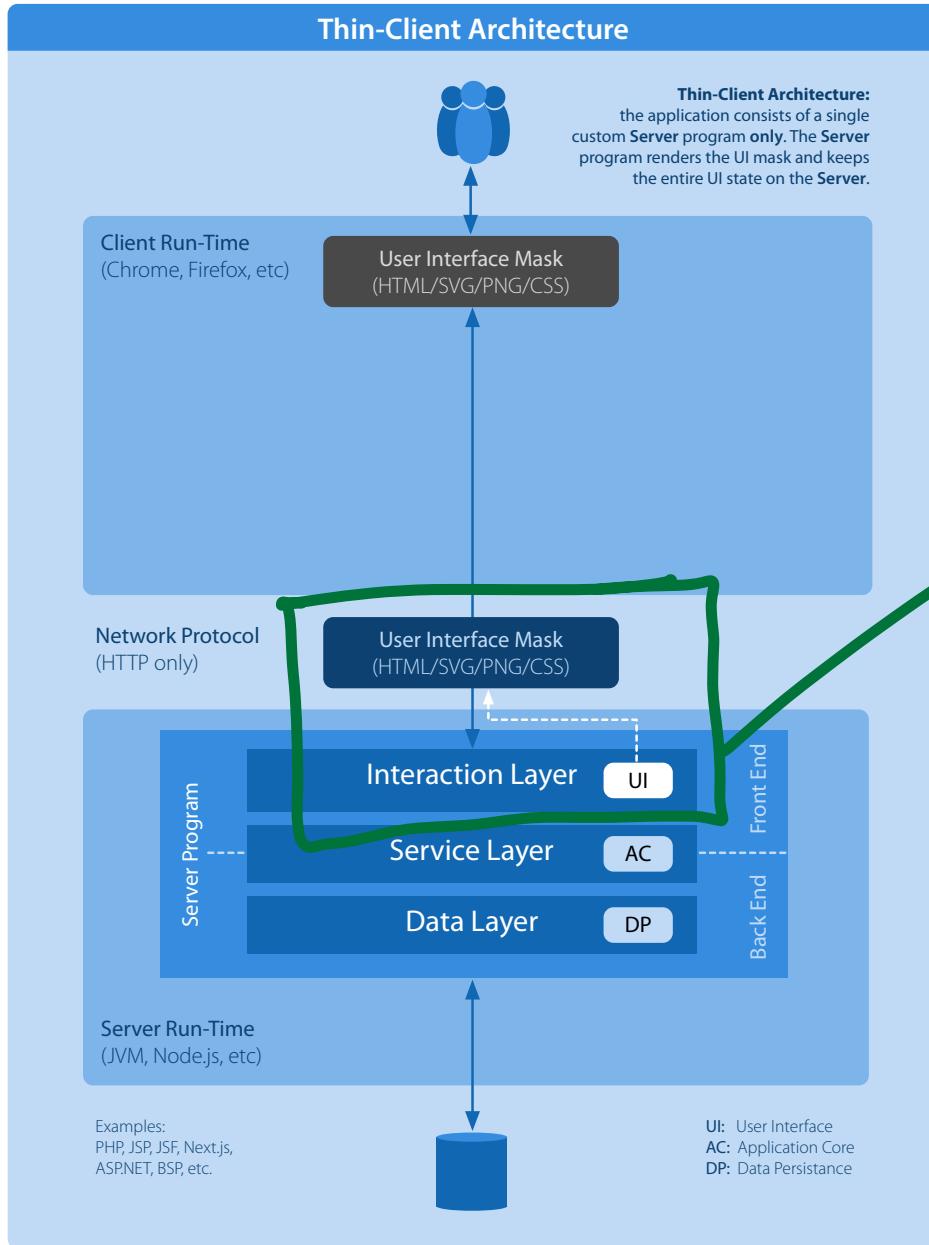


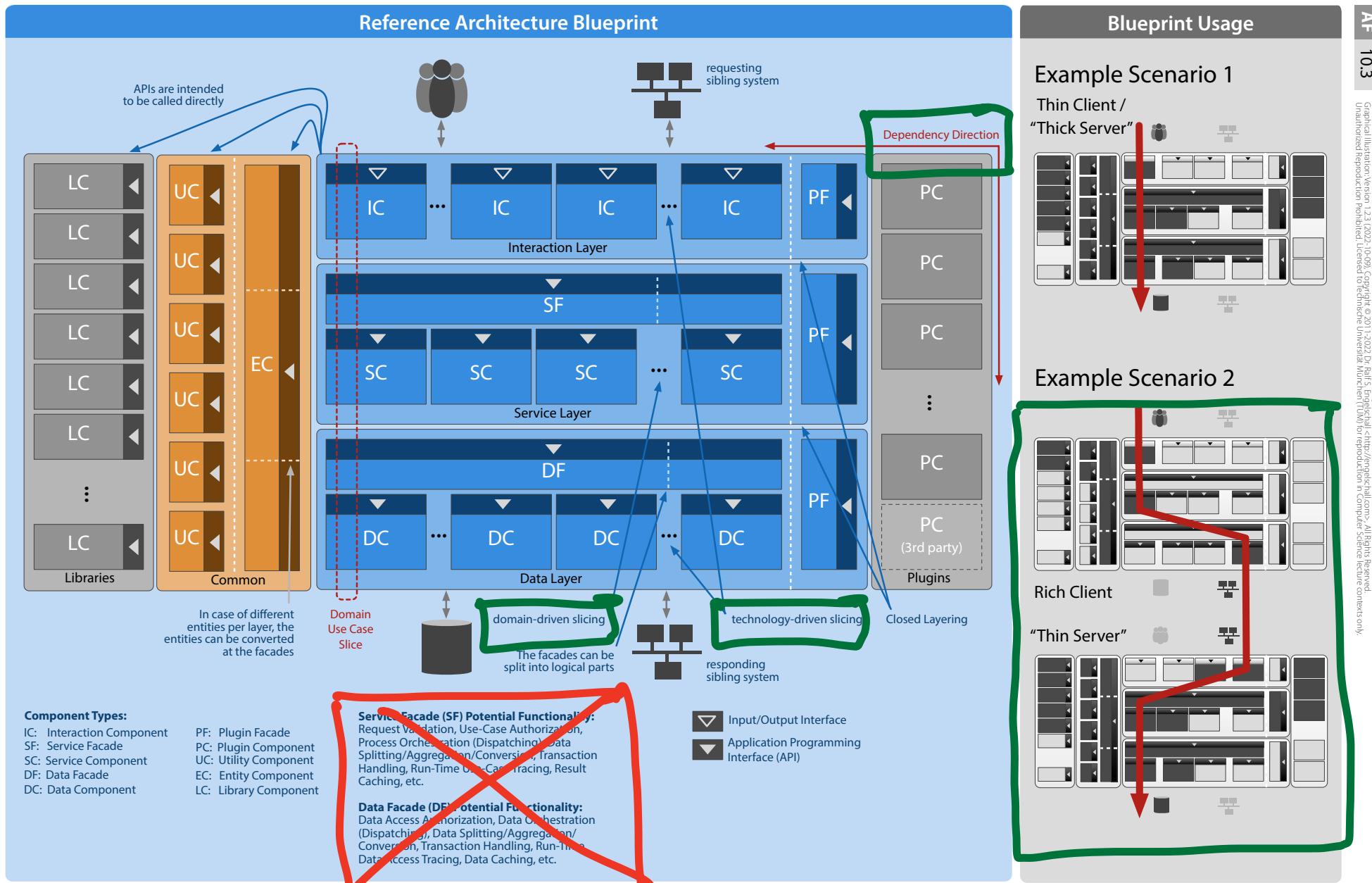
Synchronization

Continuously stream or regularly copy data between multiple master systems and resolve potential concurrent data modification conflicts. This way allow distributed and even disconnected computing.

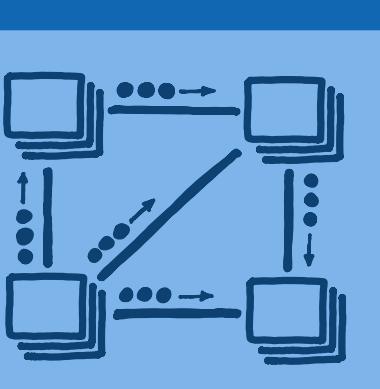




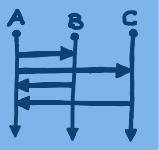
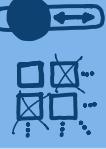
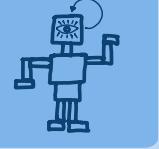




Architecture & Systems

DEF	Definition
<p><i>Reactive System Architecture enables the realization of Reactive Systems.</i></p> <p><i>Reactive Systems are in subordinated interaction with their dominating environment. They continuously process endless data streams as small messages, react at any time and respond within tight time limits. For this, they continuously observe their environment and adapt their behaviour to the current situation.</i></p>	
	

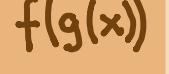
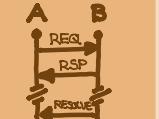
Demand & Deliverables

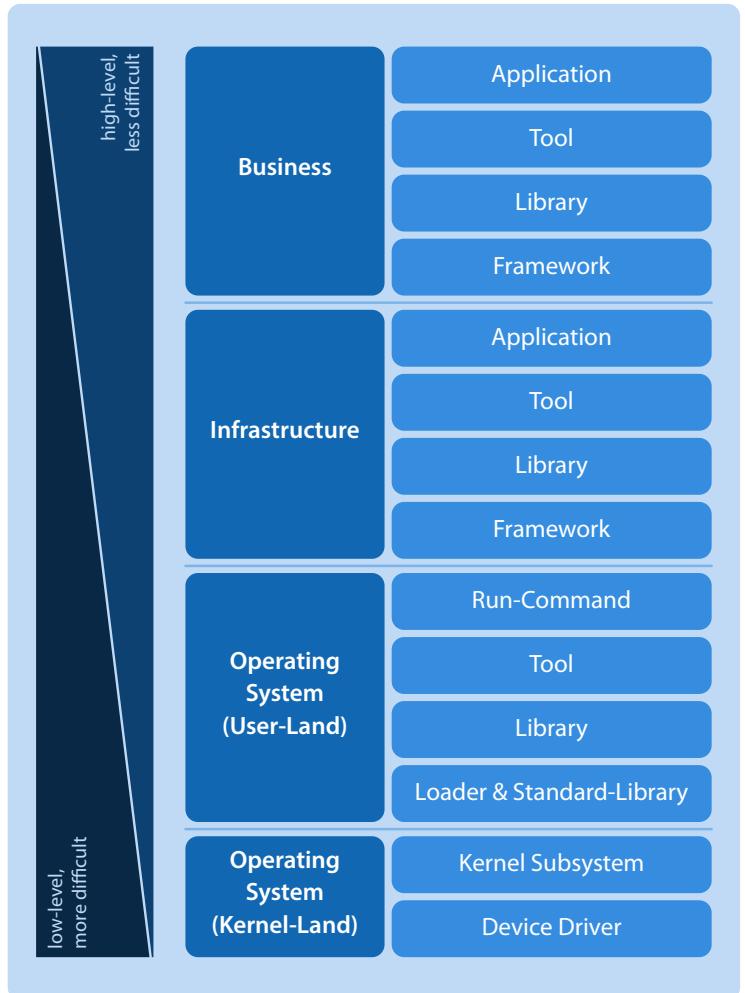
CTX	Context
<p><i>Real-time communication in the context of Digitization, Internet, Internet of Things (IoT), Systems of Engagement, Media and Analytics.</i></p> 	
VAL	Values
<p><i>Non-blocking input/output data processing, fast responses within tight time limits, and continuous availability of the provided services.</i></p> 	
REQ	Requirements
<p><i>Services are elastic and provide high scalability, and are resilient and provide high fault tolerance.</i></p> 	
PRP	Properties
<p><i>Services run fully autonomously, monitor themselves, and automatically adapt to changes in the environment.</i></p> 	

Principles

Stay Responsive Always respond in a timely manner.	Accept Uncertainty Build reliability despite unreliable foundations.	Embrace Failure Expect things to go wrong and design for resilience.	Assert Autonomy Design components that act independently and interact collaboratively.	Tailor Consistency Individualize consistency per component to balance availability and performance.	Decouple Time Process asynchronously to avoid coordination and waiting.	Decouple Space Create flexibility by embracing the network.	Handle Dynamics Continuously adapt to varying demand and resources.
--	--	--	--	---	---	---	---

Patterns & Paradigms

ARC	Architecture	COM	Communication	DAT	Data	STY	Style
<p>Microservices, Cloud-Native Architecture (CNA), Event-Driven Architecture (EDA).</p> 		<p>Asynchronous Communication, Non-Blocking I/O, Sequence, Push, Backpressure, Quality of Service (QoS).</p> 		<p>Semantical Event, Small Message, Endless Stream.</p> 		<p>Functional Programming, Asynchronous Programming.</p> 	
EXE	Execution	INF	Infrastructure	PRC	Processing	ASY	Asynchronism
<p>Parallelization, Concurrency, Actors, Threads, Thread-Pool, Event-Loop.</p> 		<p>Message Queue (MQ), Load Balancer, Reverse Proxy, Service Mesh, Virtual Private Network (VPN).</p> 		<p>Complex Event Processing (CEP), EAI Patterns, Stream Processing (map, flatMap, filter, reduce), Event Sourcing.</p> 		<p>Callback, Promise/Future, Observable, Publish & Subscribe.</p> 	



Remember:

A Technology Platform is less about choosing a particular programming language and more about choosing a particular ecosystem for targeting a particular level of software!

Opinionated Recommendation (as of 2022):

Business: Scala, Kotlin, TypeScript, AssemblyScript

Infrastructure: Go, Rust, Scala, Kotlin, TypeScript

Operating System (UL): Rust, Go

Operating System (KL): C, C++, Rust

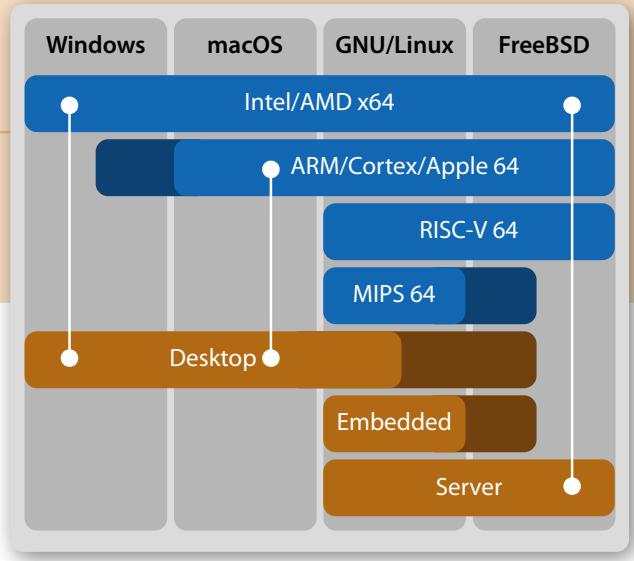
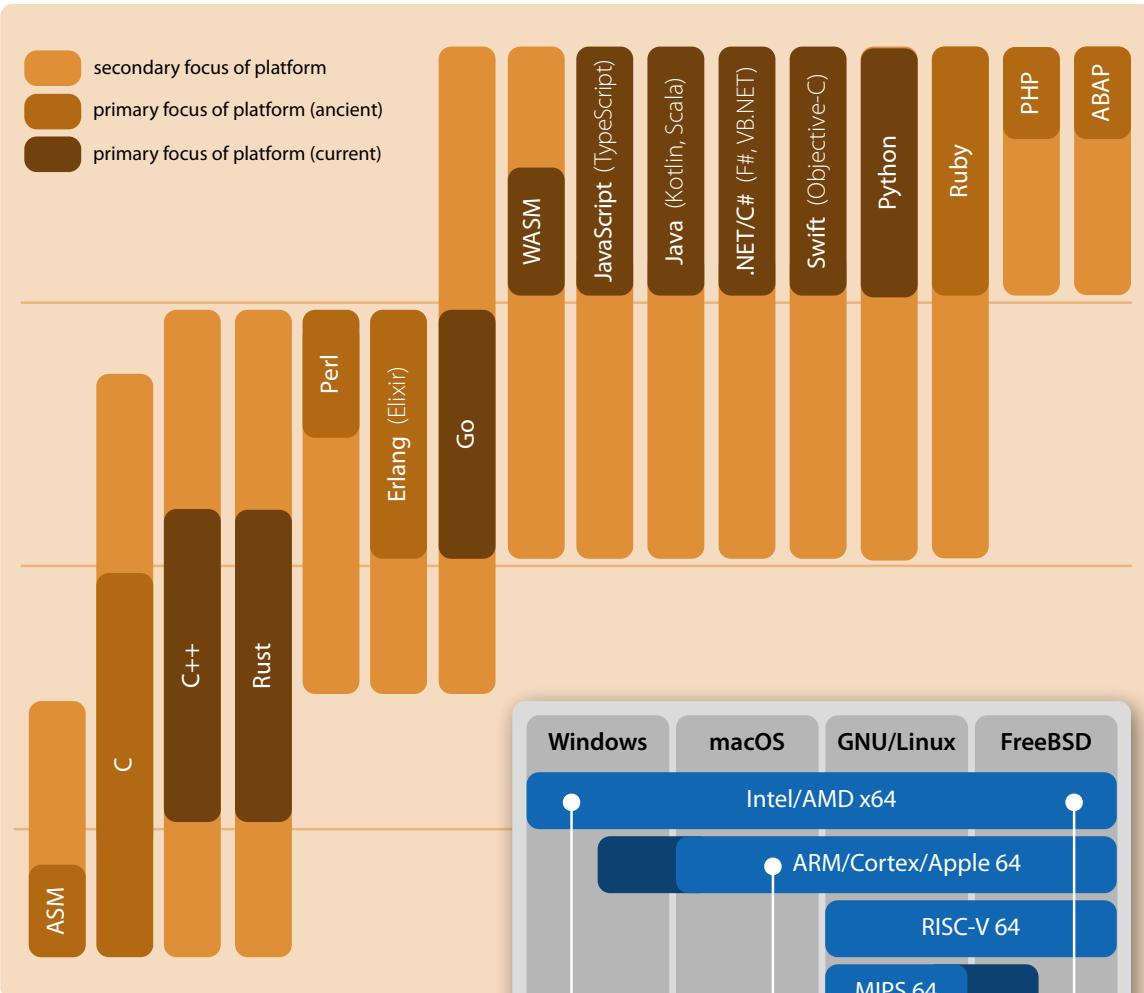
Typical Computing Devices (as of 2022)

Intel/AMD x64: Personal Computer (PC)

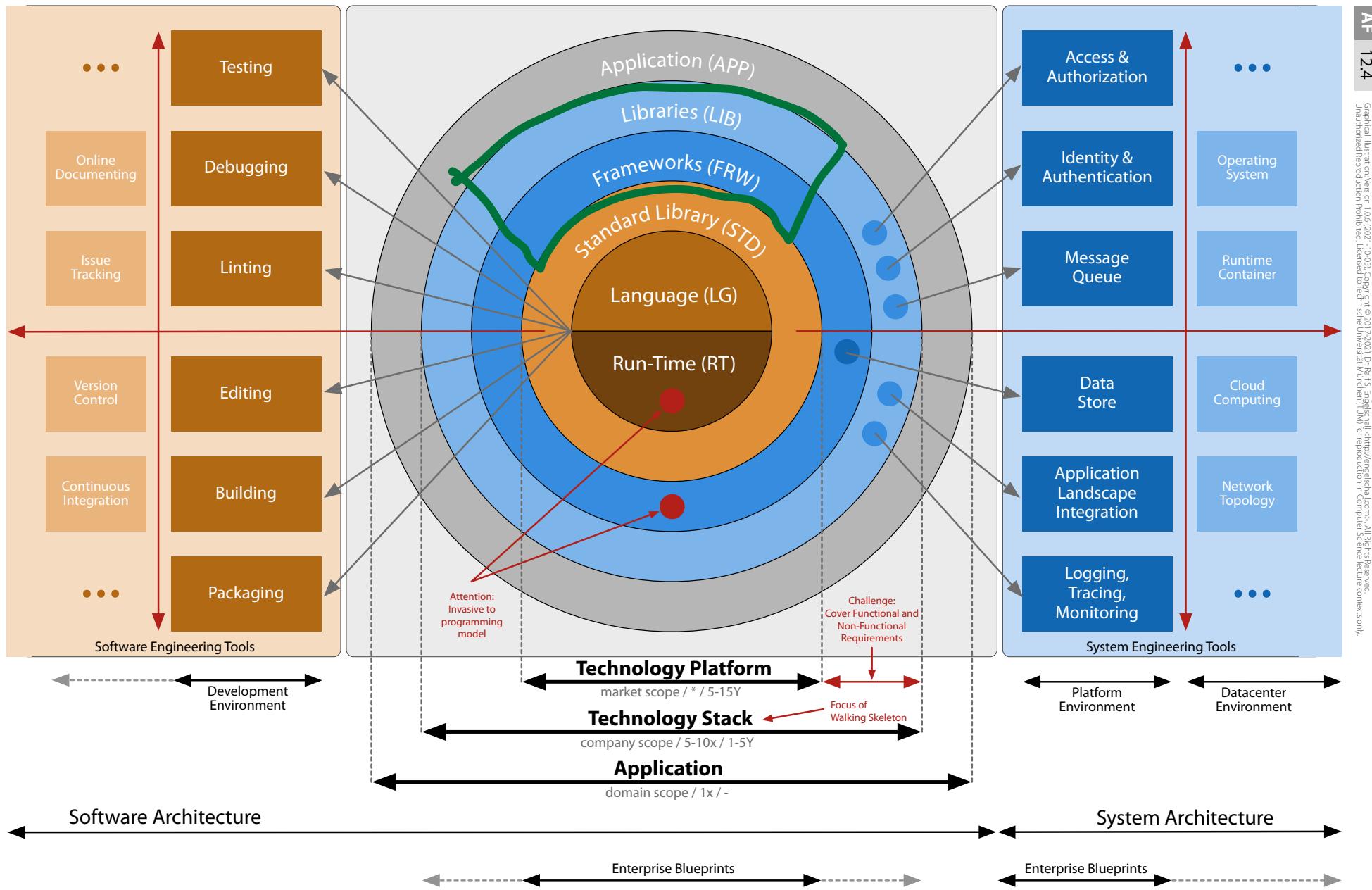
© Cortex/Apple 64: Raspberry PI, BeagleBone, ROCKSPro64, iMac

RISC-V 64: Single-V HiFive Unmatched

MIPS 64: Compex WPJ344



Technology Stack



Rich-Client Aspects

IT Interface Theme	18 Interface Internationalization	DL Dialog Life-Cycle
Style Reset, Shape, Color, Gradient, Shadow, Font, Icon Bootstrap, TypoPRO, FontAwesome, Normalize	VueJS vue-i18next, i18Next Text Internationalization (I18N).	ComponentJS (none) Component States, Component State Transitions.
IW Interface Widgets	DC Data Conversion	DS Dialog Structure
Icon, Label, Text Paragraph, Image, Form, Text-Field, Input Area, Date Picker, Toggle, Radio Button, Checkbox, Select List, Slider, Progress Bar, Hyperlink, Popup Menu, Dropdown Menu, Toolbar, Tooltip, Tab Pill, Breadcrumb, Pagination, Badge, Alert, Panel, Modal, Table, Scrollbar, Carousel Bootstrap, Select2, SlickGrid, ...	VueJS Moment, Numeral, Accounting, ... Value Formatting, Value Parsing, Localization (L10N).	ComponentJS ComponentJS-MVC Component, Model/View/Controller Roles, Hierarchical Composition
IL Interface Layouting	DB Data Binding	SP State Persistence
Responsive Design, Media Query, Frame, Grid, Padding, Border, Margin, Alignment, Force, Magnetism Bootstrap, Swiper, jQuery Page, ...	VueJS (none) Reactive, Observer, Unidirectional, Bidirectional, Incremental	ComponentJS (none) Local Storage, Cookies, Caching
IE Interface Effects	PM Presentation Model	BM Business Model
Transition, Transformation, Keyframes, Easing Function, Sound Effect, Physics VueJS Animate.css, DynamicJS, Howler, ...	ComponentJS (none) Parameter Value, Command Value, State Value, Data Value, Event Value, Value Validation, Presentation Logic	DataModelJS , Pure-UUID Entity, Field, Relationship, Universally Unique Identifiers (UUID)
II Interface Interactions	DN Dialog Navigation	UA Use-Case Authorization
Mouse, Keyboard, Touchscreen, Gesture, Clipboard, Drag & Drop VueJS Hammer, Mousetrap, Dragula, ...	ComponentJS Director, URI.js Deep Linking, Routing, Dialog Flow	ComponentJS (none) User Experience, Dialog Restriction, User Group, Role, Use-Case, Data, Access.
IS Interface States	DA Dialog Automation	CN Client Networking
Rendered, Enabled, Visible, Focused, Warning, Error, Floating VueJS (none)	ComponentJS ComponentJS-Testdrive Dialog Macros, Click-Through, Smoke Testing.	Axios , Apollo Client Request/Response, Synchronization, Push, Pull, Pulled-Push, REST, GraphQL, Authentication, Session.
IM Interface Mask	DC Dialog Communication	ED Environment Detection
Markup Loading, Markup Generation, Virtual DOM, Text, Bitmaps, Vectors, 2D/3D Canvas, Accessibility VueJS jQuery-Markup, D3, Snap.svg, FabricJS, ...	ComponentJS Latching Service, Event, Model, Socket, Hooks	Modernizr , FeatureJS, jQuery-Stage Runtime Detection, Feature Detection.

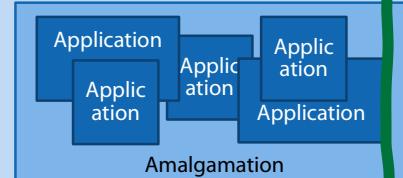
Thin-Server Aspects

ED Environment Detection	SN Server Networking	CN Client Networking
Detect the run-time environment, like underlying operating system, execution platform, network topology, feature toggles, etc.	Listen to network sockets, accept connections and manage request/response and message communication.	Provide mechanisms to connect to peers over the network and perform request/response and/or publish/subscribe communication.
Node process, syspath	HAPI hapi-plugin-websocket, ws	(none) Axios, MQL/TJs, ws
AP Argument Parsing	PI Peer Information	TS Task Scheduling
Parse options and arguments of the Command-Line Interface (CLI) to bootstrap application parameters.	Determine unique identification and add-on information about the client peer.	Schedule and execute recurring tasks independent of regular I/O operations.
(none) yargs	HAPI hapi-plugin-peer, geoip	(none) node-scheduler
CP Configuration Parsing	SH Session Handling	ET Execution Tracing
Load and parse directives from configuration file to bootstrap application parameters.	Manage secured per-connection sessions to keep state between communication requests and client sessions.	Provide mechanisms for tracing the execution by logging event and measurement information at certain points of interest.
(none) js-YAML	HAPI YAR	Microkernel Winston
PD Process Daemonizing	UA User Authentication	DA Database Access
Detach from the startup terminal and host process in order to run fully independently.	Determine and validate the unique identity of the user communicating over the current network connection.	Map in-memory domain entities onto data store dependent persistent data structure.
(none) daemonize2	HAPI JWT, Passport	Sequelize GraphQL-Tools-Sequelize
PM Process Management	RV Request Validation	DC Database Connectivity
(Pre-)fork child processes and/or threads of execution and monitor and control them during the life-cycle of the application.	Validate the syntactical and semantical compliance of the requests and sanitize the requests.	Locally or remotely connect the database access layer to the underlying data store.
(none) cluster, nodemon	HAPI Joi, DuckyJS	Sequelize sqlite3, pg
CM Component Management	RP Request Processing	DS Database Schema
Structure the code into components, instance them under run-time and manage them in a stateful component life-cycle.	Process the request by dispatching execution according to the provided request and determined context information.	Create, update or downgrade the data schema inside the underlying data store.
Microkernel (none)	HAPI GraphQLjs	Sequelize (none)
CC Component Communication	RA Role Authorization	DB Database Bootstrapping
Provide inter-component communication mechanisms like events, hooks, registry, etc.	Determine whether the role of the current user is allowed to execute the current request.	Create, update or downgrade both mandatory bootstrapping and optional domain-specific data inside the underlying data store.
Microkernel Latching	(none) GraphQL-Tools-Auth	Sequelize ini

AMA Bare Amalgamation

Manually deploy all applications into a single, shared, and unmanaged filesystem location. Dependencies are resolved manually. Examples: Windows Fonts, Unix 1990th /usr/local.

Pro: simple deployment
Con: incompatibilities, hard uninstallation



UHP Unmanaged Heap

Manually deploy all applications into multiple, distinct, and unmanaged filesystem locations. Dependencies are resolved manually. Examples: macOS *.app, OpenPKG LSYNC.

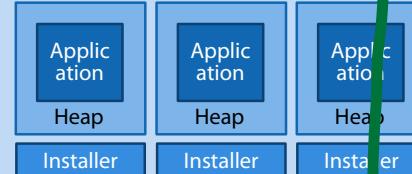
Pro: simple deployment, easy uninstallation
Con: no repair mechanism



MHP Managed Heap

Let individual installers deploy applications into multiple, distinct, and managed filesystem locations. Dependencies are manually resolved or bundled. Examples: macOS *.pkg, Windows MSI, InnoSetup.

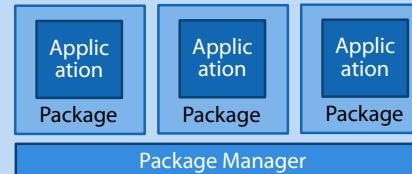
Pro: easy uninstallation, repairable
Con: requires installer, diversity, no dep.



PKG Managed Package

Let a central package manager deploy all applications into a single, shared, and managed filesystem location. Dependencies are automatically resolved. Examples: APT, RPM, FreeBSD pkg, MacPorts, Gradle, NPM.

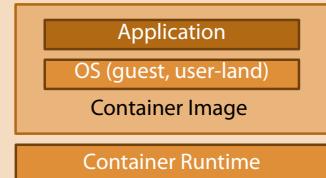
Pro: easy uninstall., repairable, dependencies
Con: P.M. pre-installation, P.M. single instance



CON Container Image

Bundle an application with its stripped-down OS dependencies and run-time environment into a container image. Examples: Docker/ContainerD, Kubernetes/CRI-O, Windows Portable Apps.

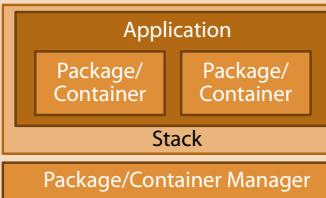
Pro: independent, simple deployment
Con: fewer variations, no dependencies



STK Package/Container Stack

Establish an application out of multiple Managed Packages. Examples: OpenPKG Stack, Docker Compose, Kubernetes/Kompose, Kubernetes/Helm.

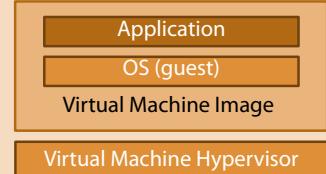
Pro: independent, flexible
Con: overhead



VMI Virtual Machine Image

Bundle an application with its full OS dependencies and run-time environment into a virtual machine image and deploy and execute this on a hypervisor. Examples: VirtualBox, VMWare, HyperV, Parallels, QEMU.

Pro: all-in-one, independent
Con: overhead, sealed, inflexible



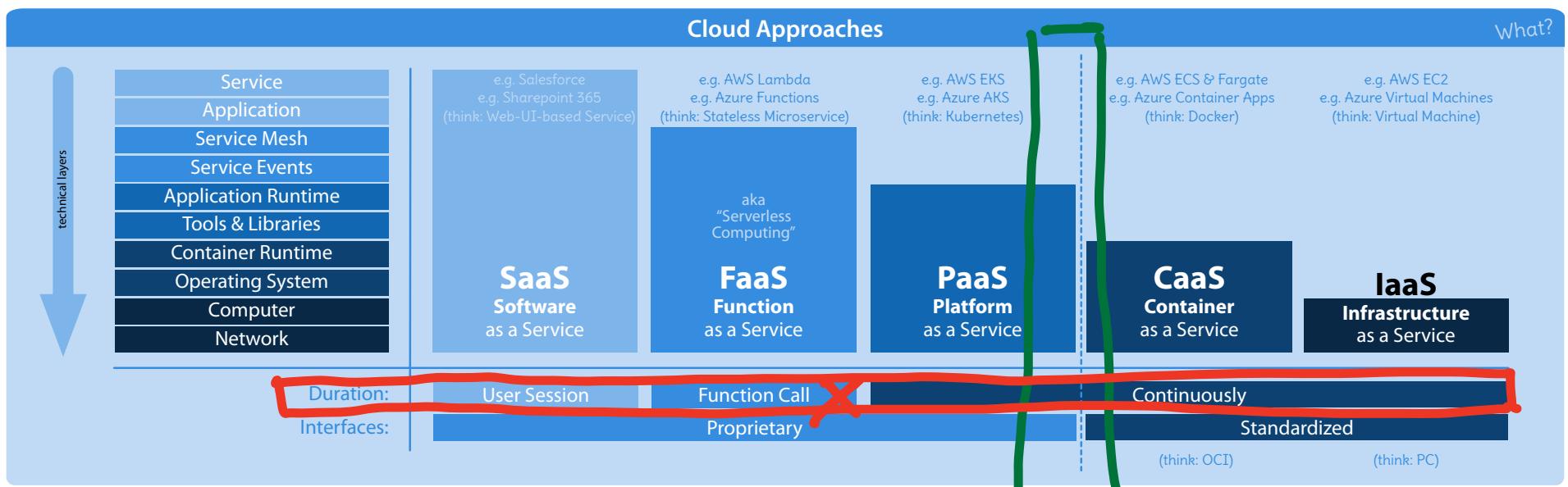
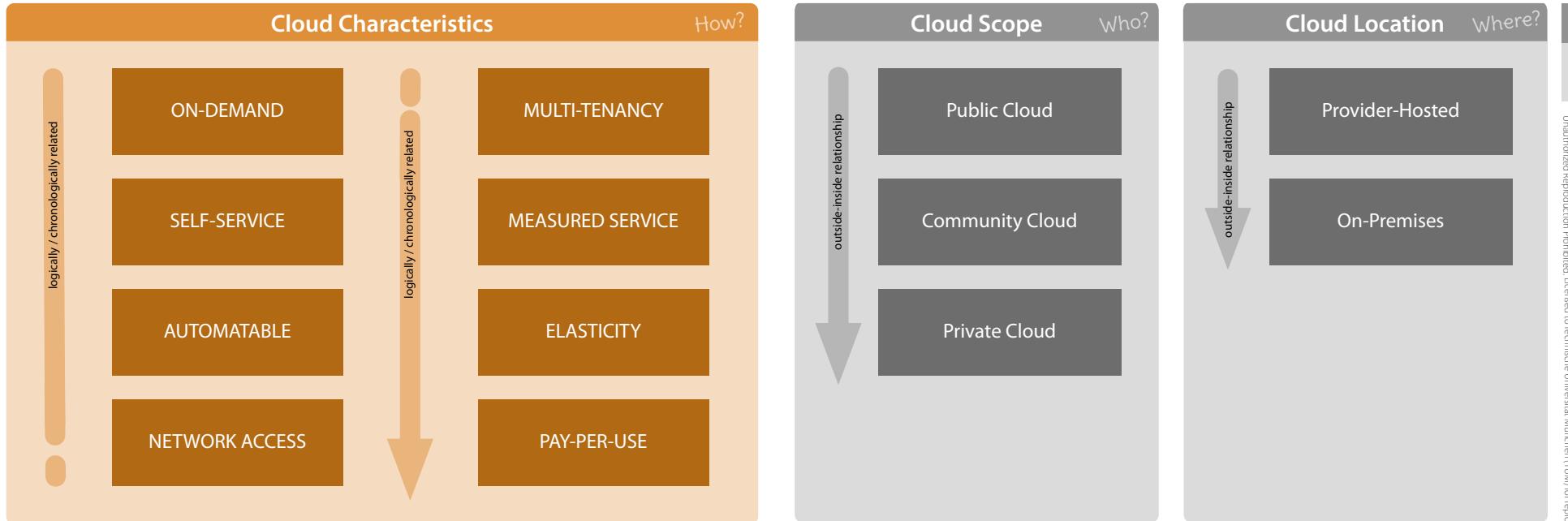
APP Solution Appliance

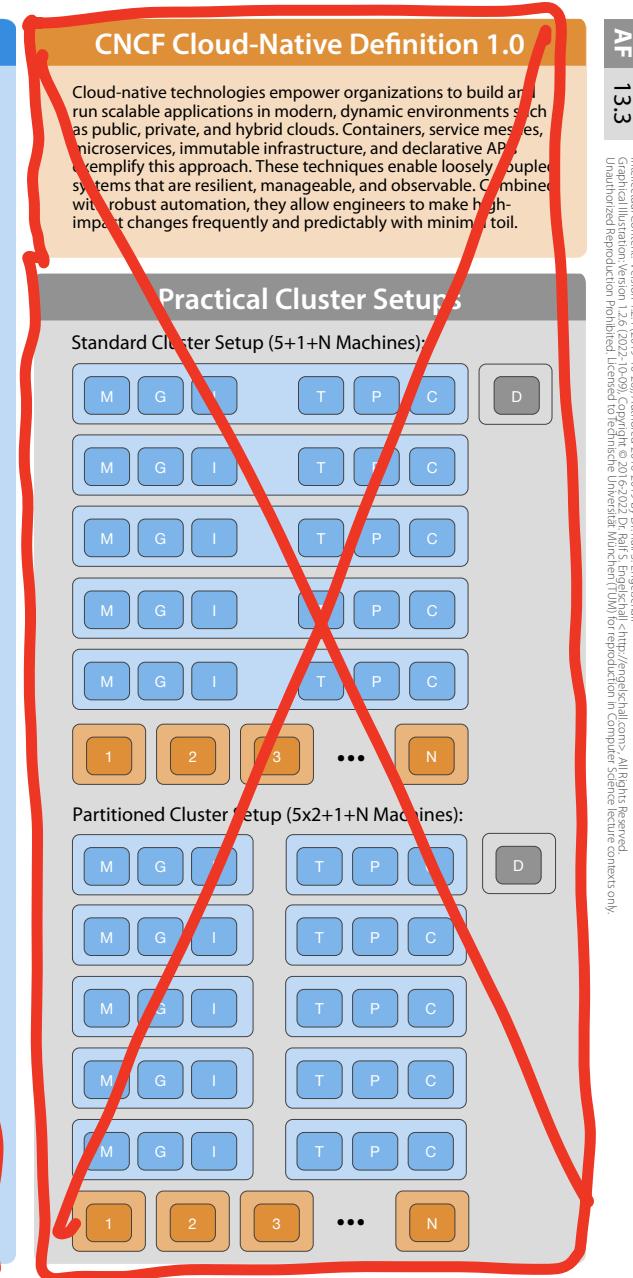
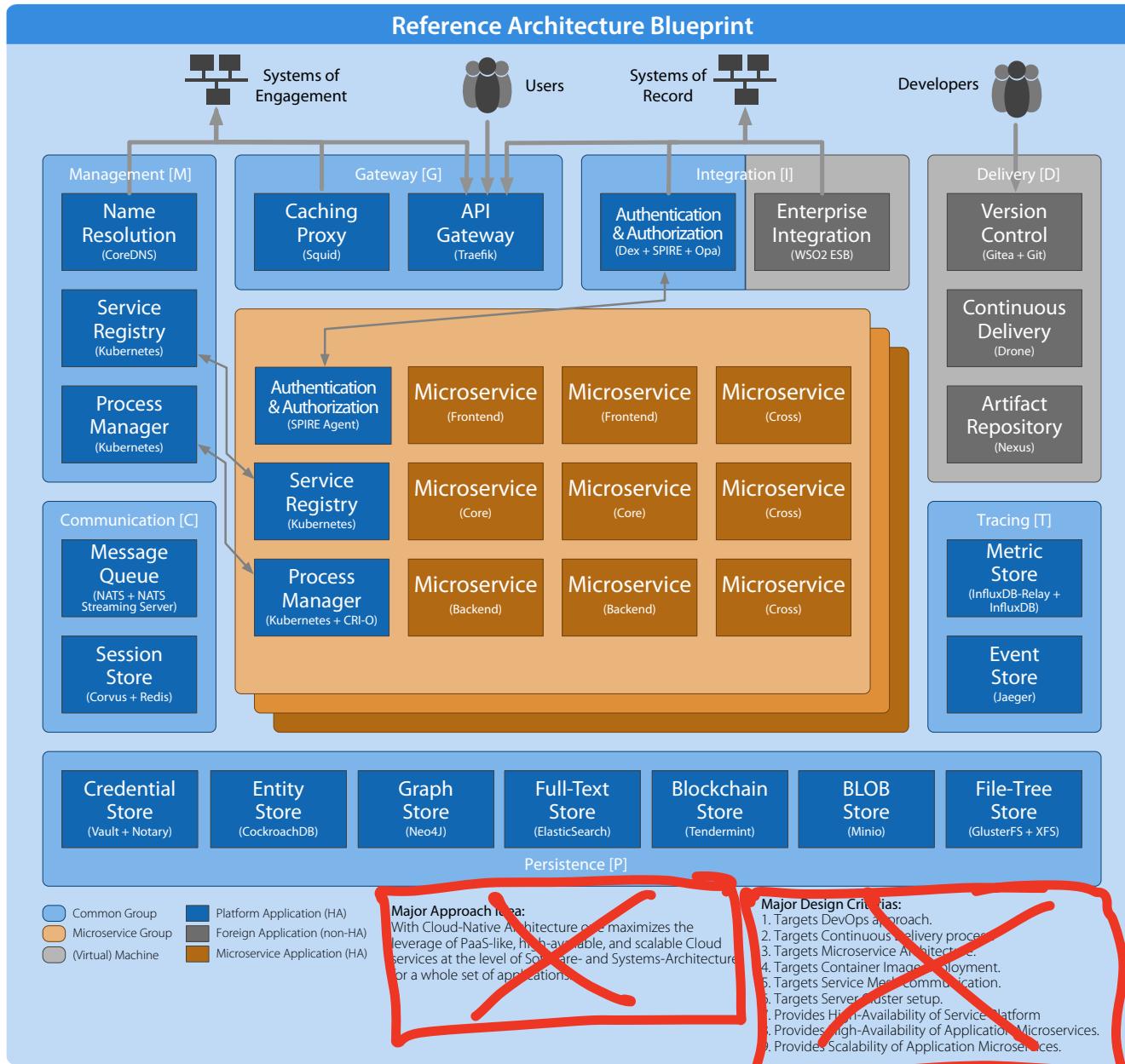
Bundle an application with its full OS dependencies, run-time environment and underlying hardware. Examples: AVM Fritz!Box, SAP HANA.

Pro: all-in-one, independent
Con: expensive, sealed, inflexible



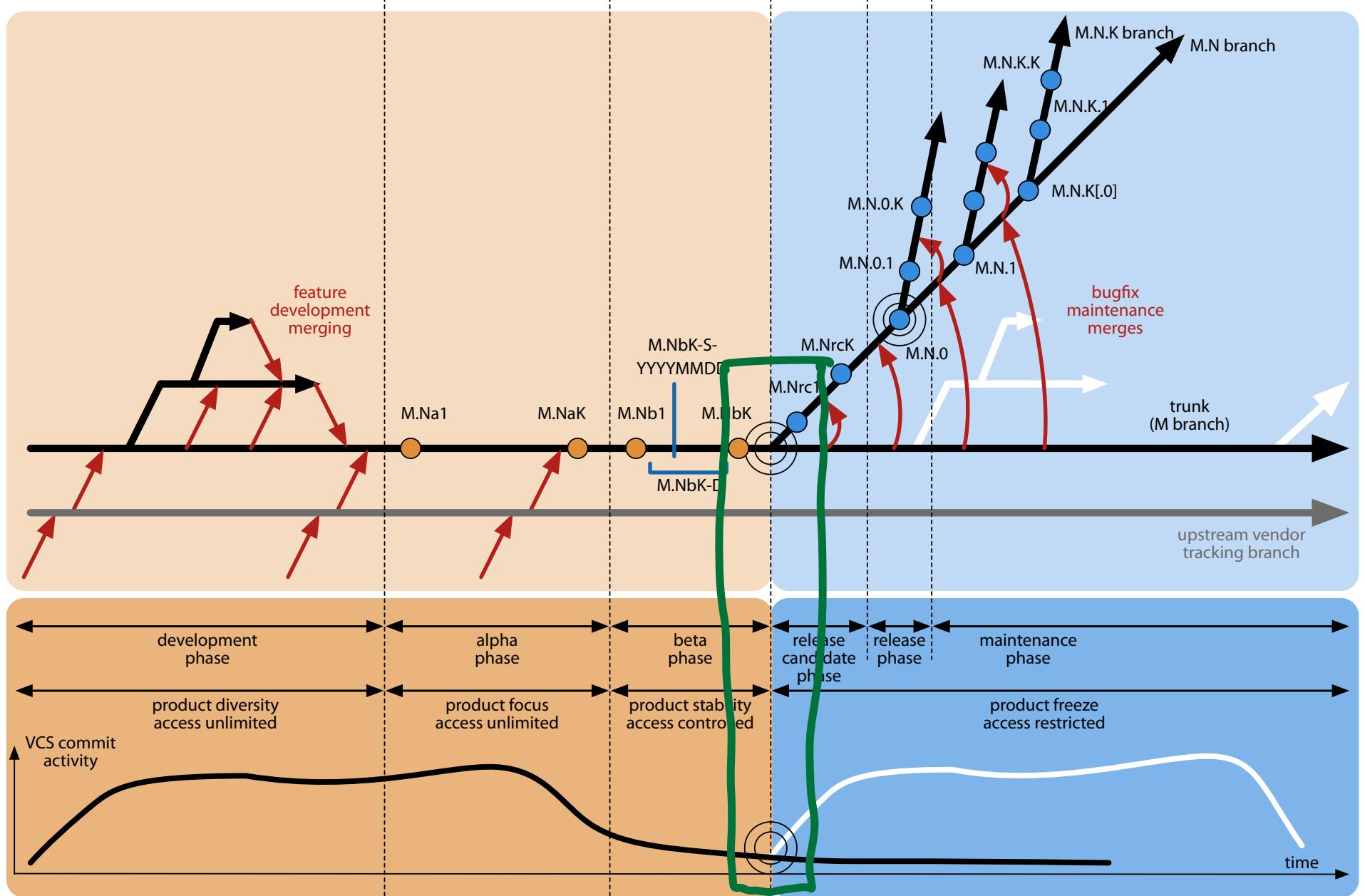
Cloud Computing Resources



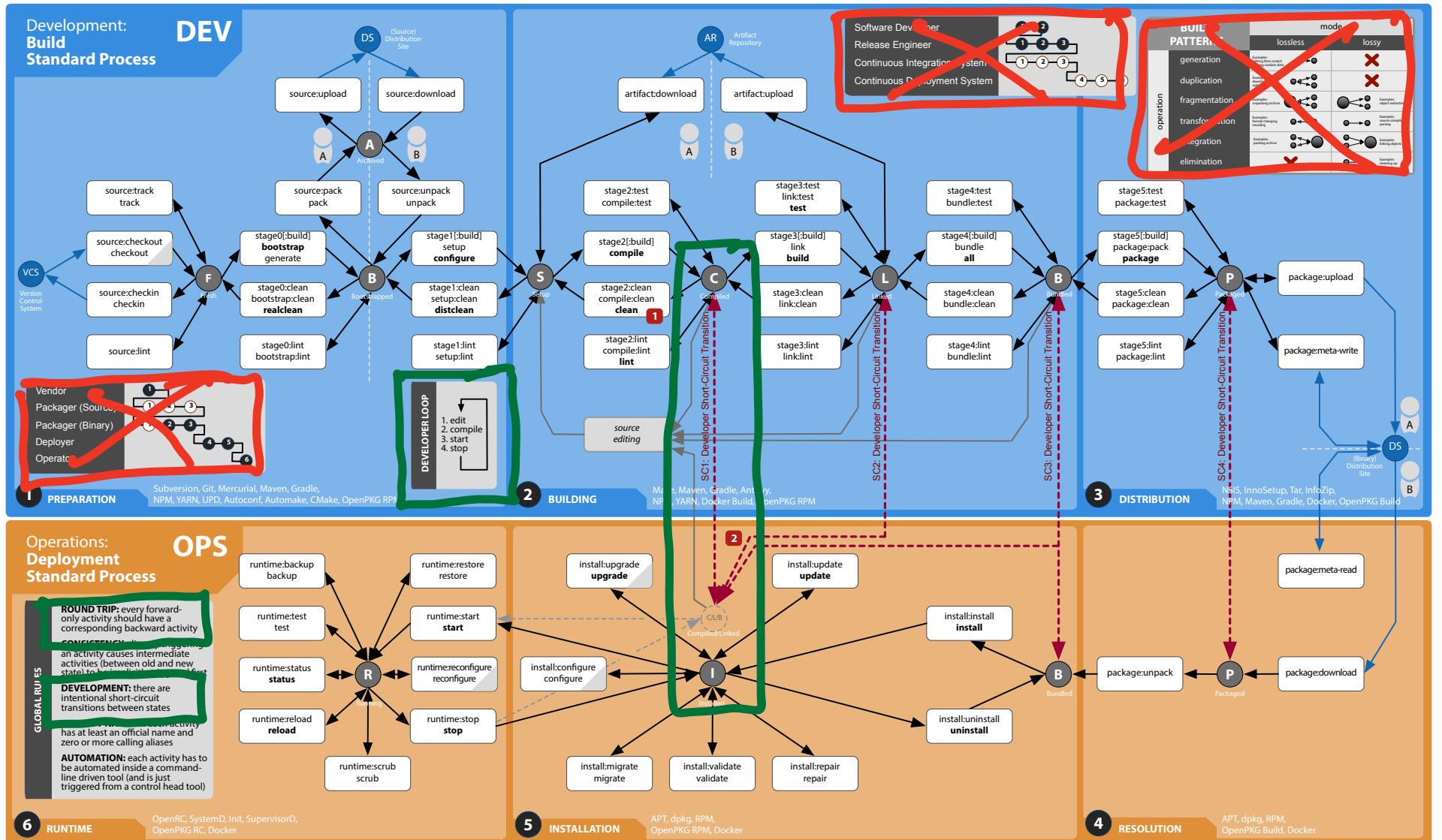


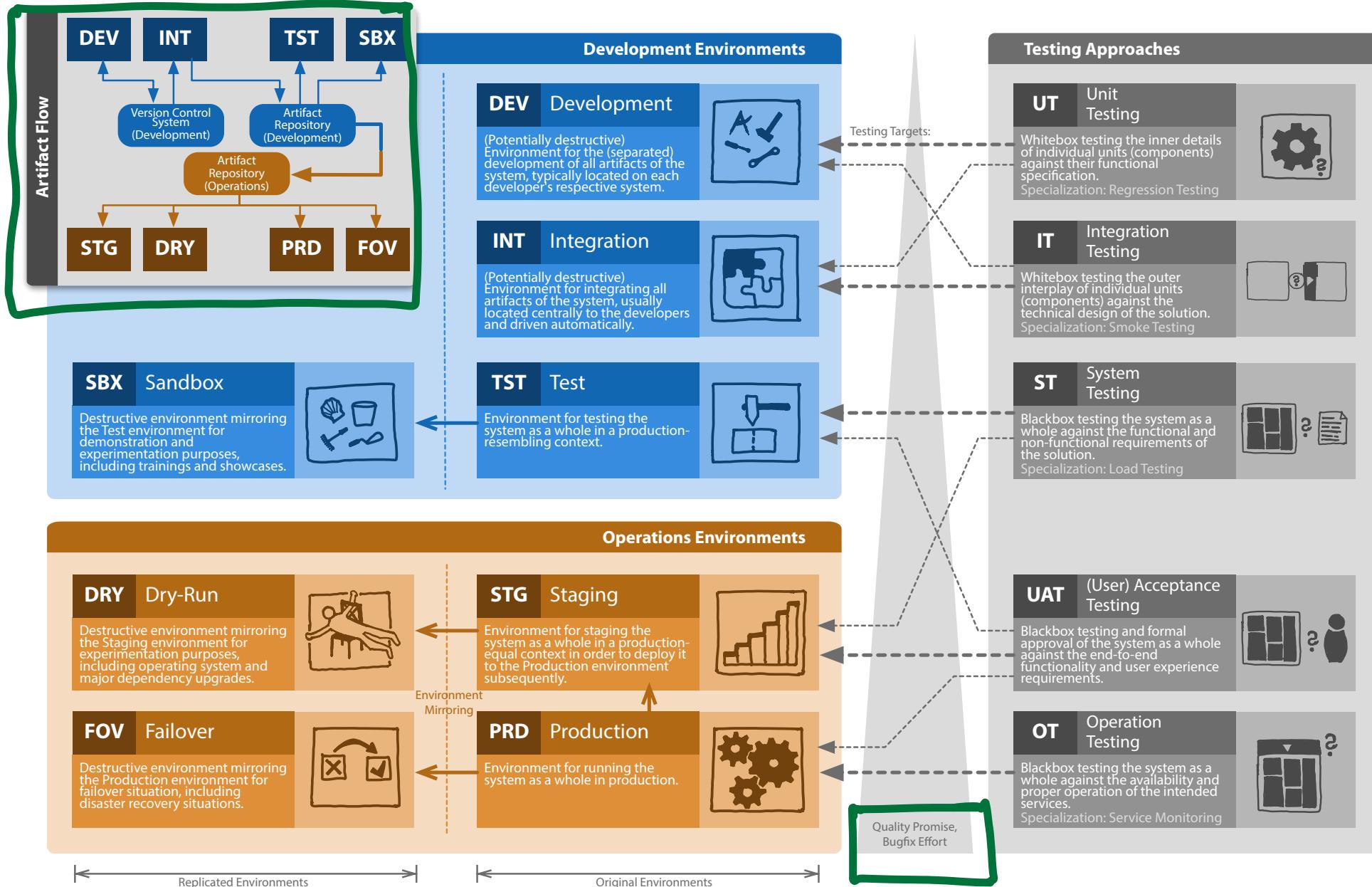
▪ Offline Capability

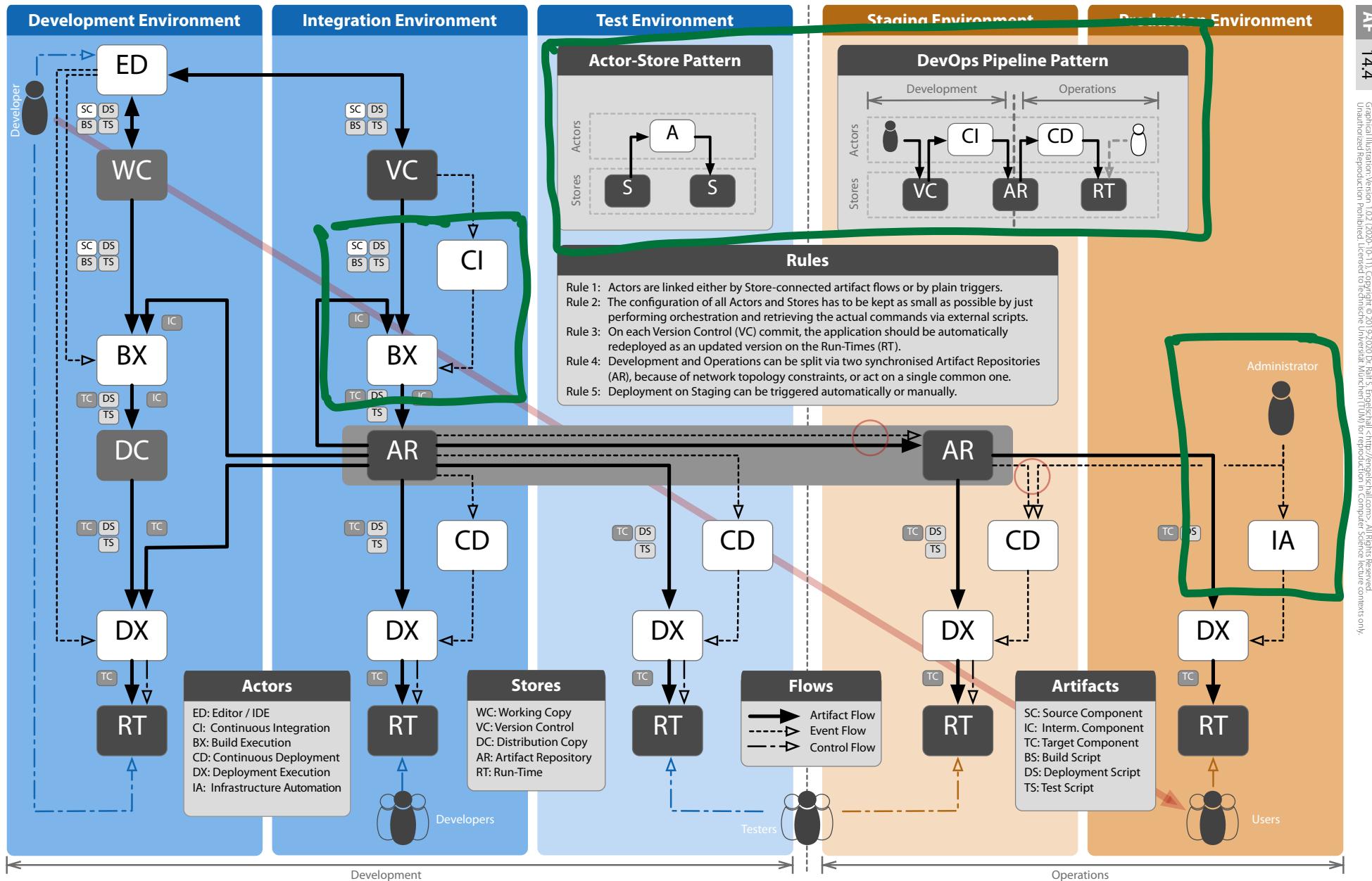




Assembly Process Architecture

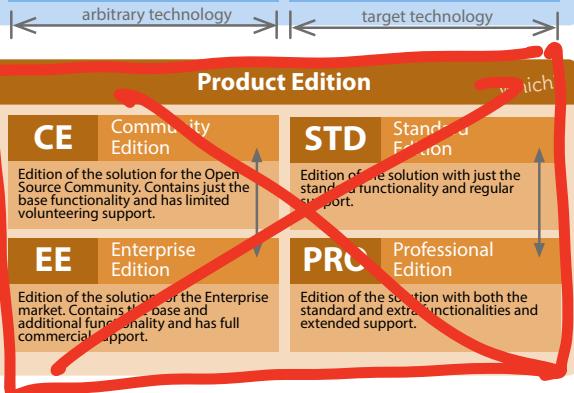






Software Release Management

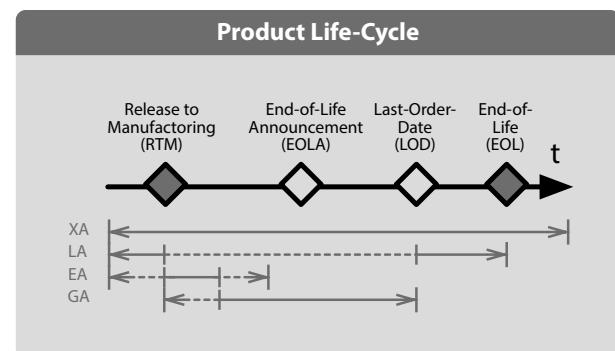
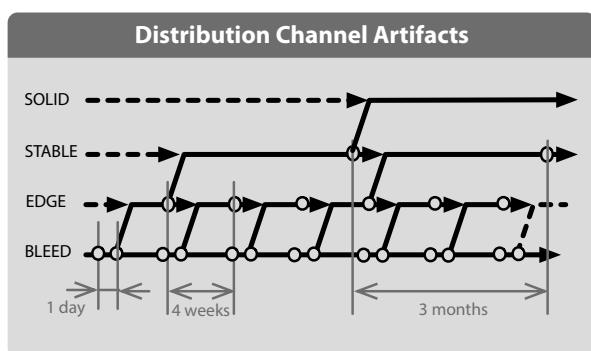
Evolution Stage		What?
WF	Wireframe	Distraction-free low-fidelity illustration of the solution and its base features, displaying its pure structure and core functionality only.
WS	Walking Skeleton	Realization of all technical, fundamental aspects of the solution, ensuring the domain-specific aspects can be realized later on top of it without problems.
PT	Prototype	High-fidelity mostly interactive sample, mockup, model or simulation of the solution and its base features, showcasing its structure and functionality.
MVP	Minimum-Viable Product	Early version of solution with just enough functionality to enable full turn of Build-Measure-Learn loop with minimal amount of effort and time.
PoC	Proof of Concept	Pure realization of most-risky aspects of the solution, proving their feasibilities. Might still be based on a different technology than WS, MVP and FP.
FP	Full Product	Final version of the solution with all intended functionality and targeting the mainstream market.



Version Number		When?
M	Major Version	Major Version of solution. Usually bumped on major technical or domain-specific changes only. A bump resets the Minor Version and the Revision, too.
N	Minor Version	Minor Version of the solution within the Major Version. Usually bumped on new features. A bump resets the Revision, too.
R	Revision	The Revision of the Release Phase within Major and Minor Version. Bumped for every A/B/C/R Release Phase.

Availability Scope (S)		Who?
XA	No Availability	No public availability of solution at all. The scope for all Development and sometimes Snapshot point-in-times.
EA	Early Availability	Early public availability of solution for early market. Usually for Beta or Release Candidate levels or for Release and initial Release Update levels.
LA	Limited Availability	Limited public availability of solution. Usually for releases after the End-of-Life-Announcement (EOLA) or for releases with specific customer features.
GA	General Availability	Late public availability of solution for mainstream market. Usually for Release and sometimes just for Release Update levels.

Distribution Channel		Where?
BLEED	Bleed Channel	Distribution channel for all daily snapshots ("YYYY.MM.DD") with experimental features turned on. Intended for testing purposes only.
STABLE	Stable Channel	Distribution channel for all quarterly releases ("YYYY.QN") with experimental features turned off. Intended for fast mainstream market and production use.
EDGE	Edge Channel	Distribution channel for all monthly releases ("YYYY.MM") with experimental features turned on. Intended for early market or testing purposes.
SOLID	Solid Channel	Distribution channel for all (half-)year releases ("YYYY[N]") with experimental features turned off. Intended for slow mainstream market and production use.



Sustainability

HI Minimize **HARDWARE Idleness**

Minimize the idleness and maximize the utilization of existing hardware resources.

Rationale: Unused or under-utilized hardware are an unnecessary waste of already available resources.

Keywords: Virtualization, Utilization.



DE Minimize **DESIGN Excessiveness**

Minimize the excessiveness and maximize the adequacy of solution designs.

Rationale: Non-adequate designs cause unnecessary complexity and waste resources.

Keywords: Reduced Libraries, Immutability.



HE Minimize **HUMAN Effort**

Minimize the efforts of humans and maximize the efforts of machines in all production and operation processes.

Rationale: Delegating tasks to machines gives humans the possibility to concentrate on more important tasks.

Keywords: Computer, Robot, Automation.



SI Minimize **SOFTWARE Inefficiency**

Minimize the inefficiency and maximize the efficiency of software applications and their development processes.

Rationale: Efficient software and development processes consume less resources.

Keywords: Caching, Monolith.



SE Minimize **SOLUTION Ephemerality**

Minimize the ephemerality and maximize the life-span of any type of solutions.

Rationale: Short life-spans of solutions cause unnecessary short renewals and this way wastes resources.

Keywords: High Quality, Best Practice.



EC Minimize **ENERGY Consumption**

Minimize the consumption and maximize the saving of energy in all production and operation processes.

Rationale: Electric energy still has to be partially generated from non-renewable resources.

Keywords: Eco Mode, Reduced CI/CD.



IA Minimize **INFORMATION Amount**

Minimize the total amount of gathered, transmitted, stored and spreaded information.

Rationale: Reduced amount of information means less data transmission, less data storage, less GDPR issues, etc.

Keywords: Compression, No Big Data.



EE Minimize **ECOSYSTEM Exploitation**

Minimize the exploitation and maximize the back-contribution in any type of ecosystems.

Rationale: The consumer and provider behaviour have to be in balance for every long-lasting ecosystem.

Keywords: Open Source Software.



CE Minimize **CARBON Emission**

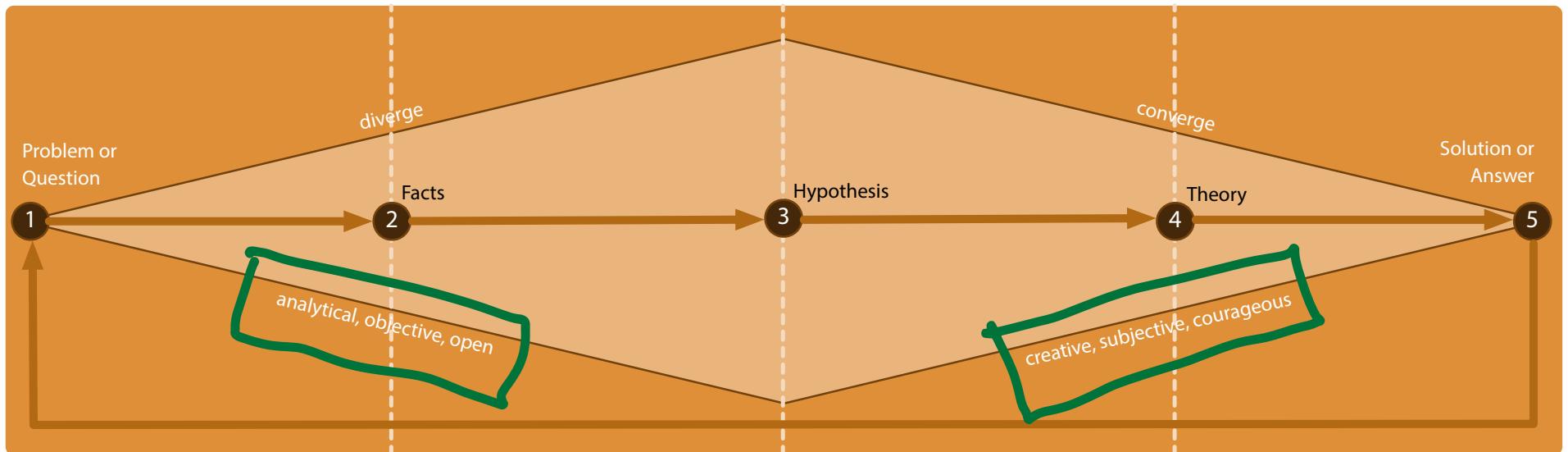
Minimize the carbon emission and hence the footprint during any type of production and operation processes.

Rationale: Climate change and global warming is partially caused or at least accelerated by carbon emissions.

Keywords: Reduced CO₂ Footprint.

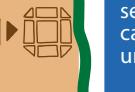
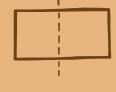


Think Clearly

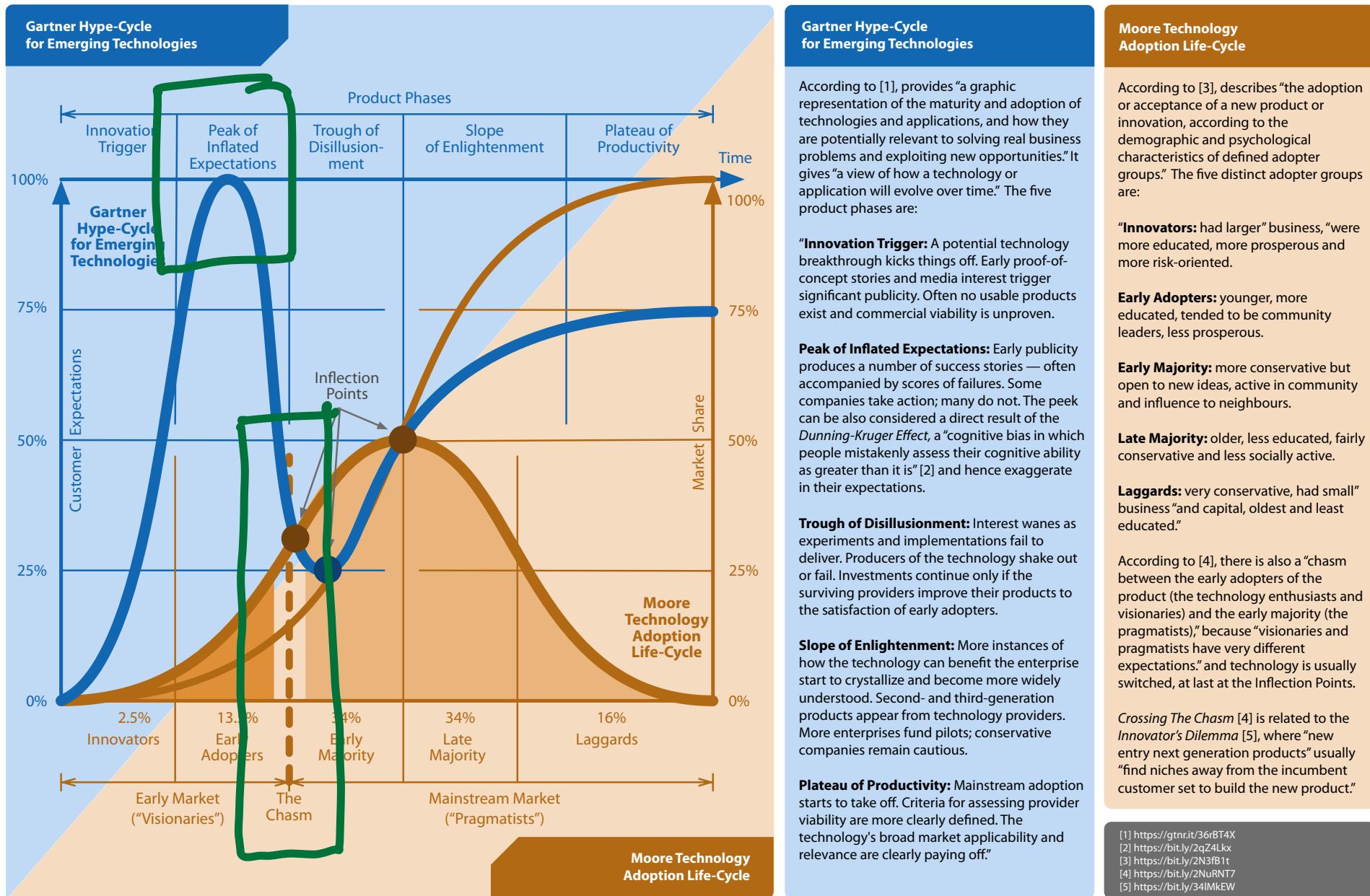


(primary step)

(secondary step)

Research	RE	Abstraction	AB	Lateral Thinking	LT	Backward Search	BS
Crawling the problem domain's body of knowledge to find starting points.		Solving the problem in a model of the problem before applying it to the real problem to get a better understanding.		Approaching the problem indirectly and creatively to find a not obvious solving lever.		Looking at the expected results and determine which operations could bring you to them.	
Brainstorming	BR	Generalization	GE	Hypothesis Proof	HP	Backtracking	BT
Suggesting larger number of solution ideas for further combination and development.		Thinking about the problem more abstract to get rid of special cases.		Assuming a possible solution and trying to prove (or disprove) the assumption to find starting points.		Remembering path towards the solution and on failure tracking back and choosing a new path.	
Analogy	AN	Specialization	SP	Root Cause	RC	Divide & Conquer	DC
Thinking in terms of similar problems for which solutions are known to get inspired.		Solving a special case first to get an impression towards the full solution.		Asking "Why?" five times in sequence to explore the cause-and-effect relationships underlying the problem.		Breaking down the large complex problem into smaller, easier solvable partial problems.	
Reduction	RD	Variation	VA	Means End	ME	Trial & Error	TE
Transform the problem into another one for which a solutions already exists to reduce solving efforts.		Changing the problem context or expressing the problem differently to find a not obvious solving lever.		Choosing an action from scratch just at each step to move closer and closer to the solution.		As a last resort brute-force testing all potential solutions in case of a small enough total solution space.	

Definition: **Heuristic** — fallible experience-based technique or strategy for problem solving in case *Rule of Thumb Guessing, Intuitive Judgement, Common Sense* and *Stereotyping* are either not sufficient or not appropriate.



Open Source Definition

Distribution terms (license) of Open Source Software must be compliant with the following criterias:

- Free Redistribution (Original) Source Code Availability
- Derived Works Allowance
- Integrity of the Author's Source Code
- No Discrimination Against Persons or Groups
- No Discrimination Against Fields of Endeavor
- Distribution of (Non-Exclusive) License
- License Must Not Be Specific to a Product
- License Must Not Restrict Other Software
- License Must Be Technology-Neutral

Open Source Personality Streams

§ Software Sharing

Dogmatism
Social Equity
Politics

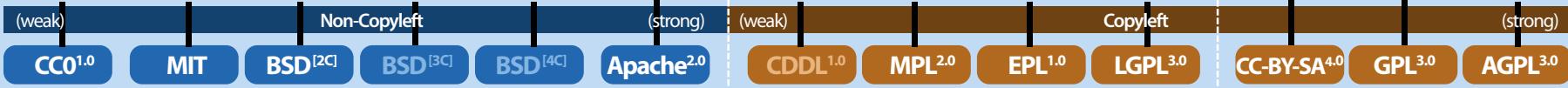
@ Software Hacking

Fundamentalism
Art
Hacking

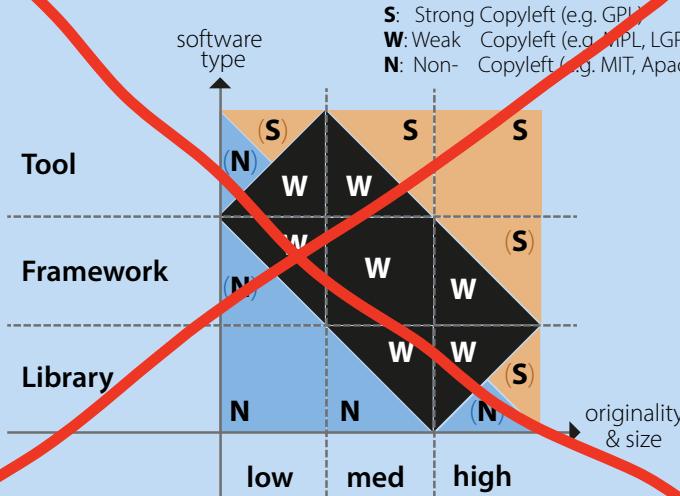
€ Software Engineering

Pragmatism
Business
Engineering

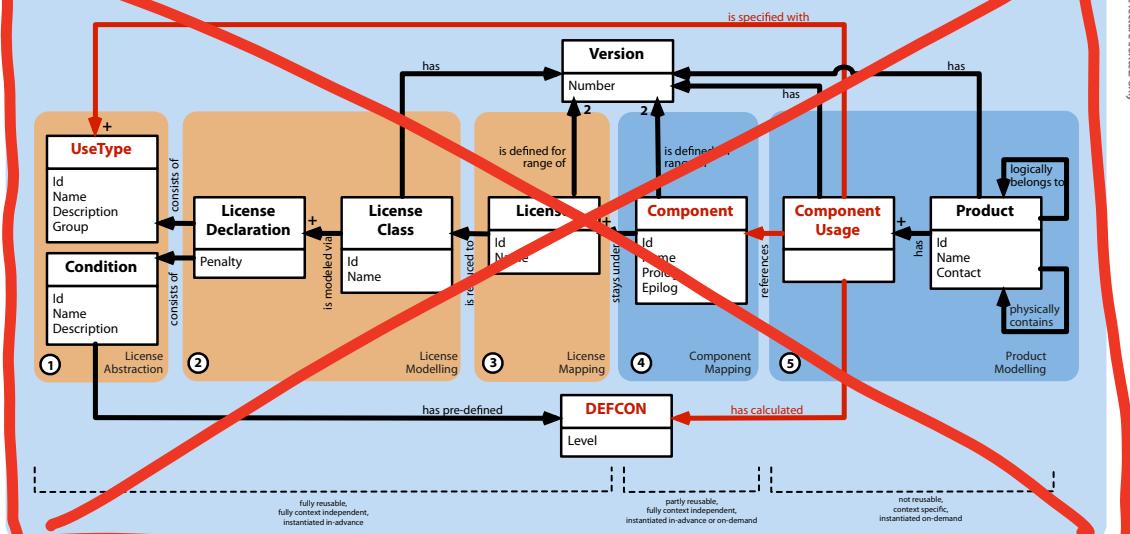
Most Popular Open Source Licenses



Choosing an Open Source License



License Compliance Checking Meta-Model



Specification (Example)

Customer: Twitter Inc.
Business: MicroBlogging

Use-Cases 1/3 (profile):

- user can register an account
- user can "follow" other users
- user can create lists of users he follows

Use Cases 2/3 (send):

- user can send tweets
- tweets are based on words, each either a text "example", tag "#example", user reference "@example" or URL <http://example.com>
- tweets are either public broadcast or personal/direct messages
- user can re-tweet a message of others

Use Cases 3/3 (query):

- user can view timeline (chronological tweets of others he follows)
- user can search for tweets (by keyword "foo", tag "#foo", or user "@foo")
- user can view tag cloud

Frontends/Clients:

- mobile app (iOS, Android)
- desktop app (Windows, Mac OS X)
- web app
- embedded web widget (query use cases only)

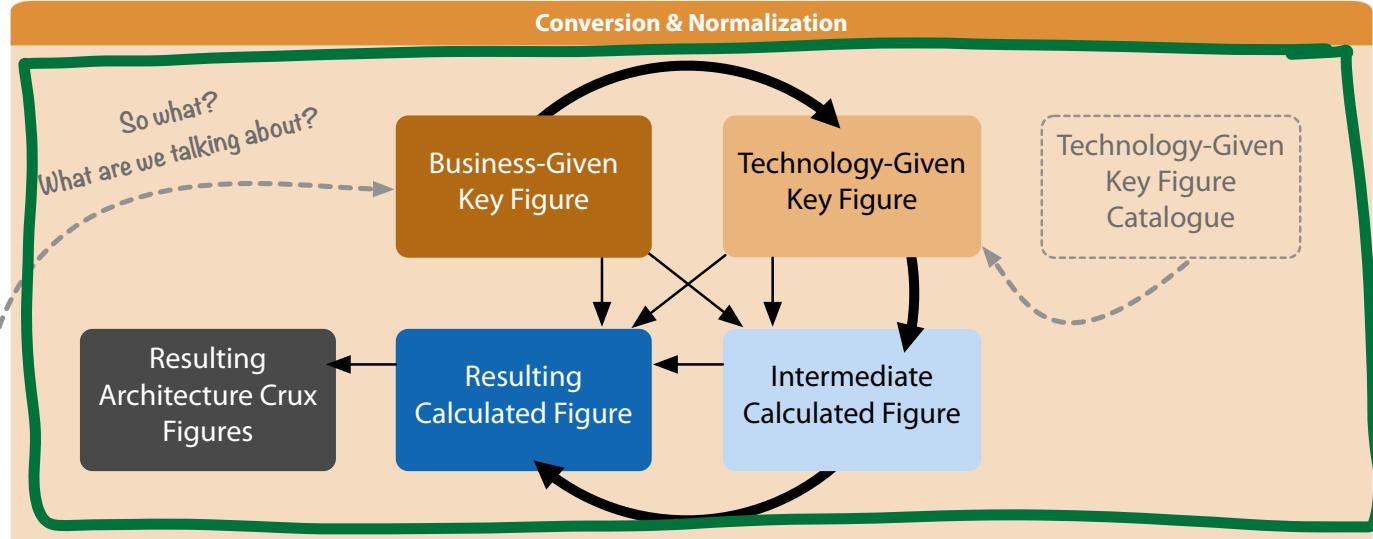
Current Demand (as of 2012):

- 140M user profiles
- 400M tweets/day
- 6393 tweets/second peak
- 140 characters/tweet
- 30K queries/second
- 300 GB/hour data in total
- 4,4 tweets/day/user on average
- 103,4 follower/user
- < 5s tweet-write-to-read-delay

Future Demand:

- quadratic user and traffic growth

Conversion & Normalization



```

graph TD
    A[Business-Given Key Figure] --> C[Resulting Calculated Figure]
    A --> B[Technology-Given Key Figure]
    B --> C
    B --> D[Intermediate Calculated Figure]
    D --> C
    D --> E[Resulting Architecture Crux Figures]
    E --> C
    
```

*So what?
What are we talking about?*

Calculation (Example)

Twitter Information		Traffic Bandwidth	
6.393 tweets/second peak	140 chars/tweet	350% overhead HTTP+TCP+IP+Ethernet	1250 MB/s
400.000.000 tweets/day (write)	4.630 tweets/second (write)	2,2 MB/s (write)	125 MB/s
2.592.000.000 queries/day (read)	30.000 queries/second (read)	140,2 MB/s (read)	12,5 MB/s
6,5 factor read/writes	10 tweets/query	142,4 MB/s	1,25 MB/s
621.880.000 tweets/day (average) total		64,3% users are active at all	
621.880.000 tweets/day (average) total		277.778 users/minute active	
Storage Requirements (static)		Storage Requirements (dynamic)	
140.000.000 user profiles	300 GB/hour data in total	2000 requests/sec (read) AS performance	1250 MB/s
52.000 chars/users for profile	214 TB/month data in total	100 requests/sec (write) AS performance	125 MB/s
6,62 TB profile (total)	100% overhead storage	15,0 servers for writes	12,5 MB/s
103,4 follower/user		1000 Mbps	
14.474.600.000 user follow links		1256,9 KB/s tweets	1000 Mbps
32 bytes/link		104,3 GB/day tweets	100 Mbps
0,42 TB links (total)		3,1 TB/month tweets	10 Mbps
Storage Hardware Requirements			
0,3 TB/disk (15K rpm)	200 chars/log entry	2000 requests/sec (read) AS performance	1250 MB/s
8,0 disks/server	6763,6 KB/s log	100 requests/sec (write) AS performance	125 MB/s
2,4 TB/server	557,3 GB/day log	15,0 servers for writes	12,5 MB/s
8,2 server/month (new)	16,6 TB/month log	46,3 servers for reads	1,25 MB/s
Computing Hardware Requirements			
9,2% ratio business data			
90,8% ratio infrastructure data			

AF 165

Intellectual Content: Version 11.01.2021-11-23, Authored 2011-2021 by Dr. Ralf Engelhardt. Graphical User Interface: Version 13.01.2021-10-20, Authored 2012-2021 by Dr. Ralf Engelhardt. Copyright © 2012-2021 Chair for Computer Systems and Networks, Institute for Microheterogeneous Devices, TUM. All Rights Reserved. All trademarks and registered trademarks are the property of their respective owners.

Weighted Decision Matrix

		A ₁	A ₂	...	A _n
C ₁	w ₁	E _{1,1}	E _{1,2}	...	E _{1,n}
C ₂	w ₂	E _{2,1}	E _{2,2}	...	E _{2,n}
...
C _m	w _m	E _{m,1}	E _{m,2}	...	E _{m,n}
		R ₁	R ₂	...	R _n

Light-Weight Alternative:
qualitatively cherry-picking major positive/negative backing criterias

		A ₁	A ₂	...	A _n						
+		C ₁	C ₄	C ₈	C ₁	C ₂	C ₇	...	C ₂	C ₉	C ₈
-		C ₂	C ₃	C ₄				...	C ₇	C ₅	
Decision for A _{best}											

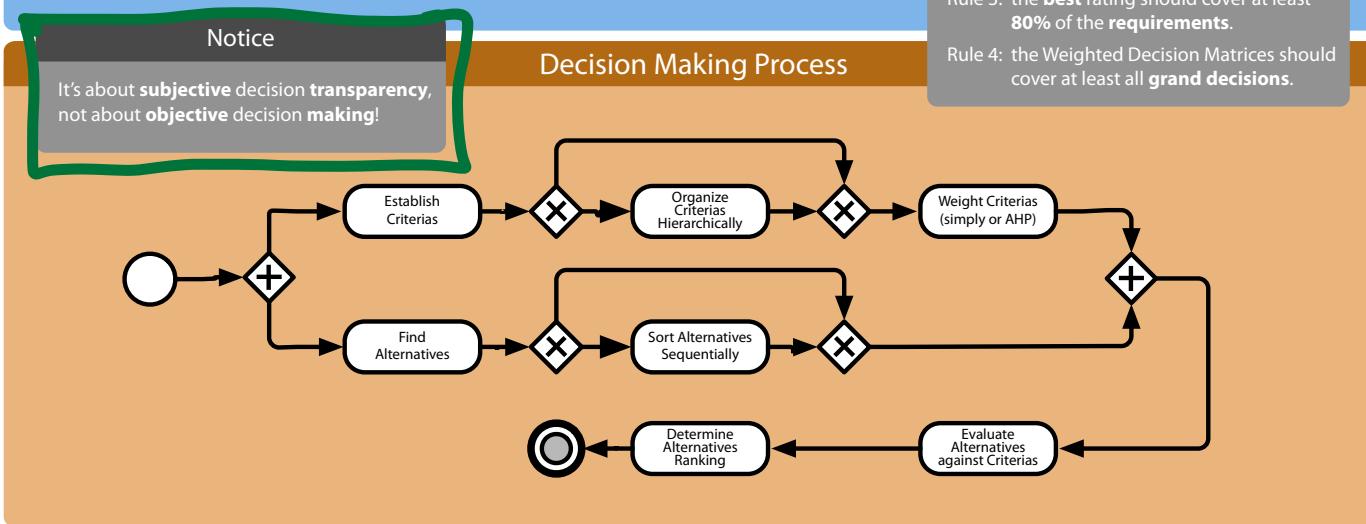
Notice

It's about **subjective decision transparency**,
not about **objective decision making!**

Decision Making Process

Best Practice Rules

- Rule 1: the alternatives have to be really **reasonably comparable**.
- Rule 2: the **best** rating should be a least **10% above** the **second** best rating.
- Rule 3: the **best** rating should cover at least **80%** of the **requirements**.
- Rule 4: the Weighted Decision Matrices should cover at least all **grand decisions**.



Standard Criteria Catalogs

Software Selection:

- Suitable Functionality
- Available Usage Examples
- Reasonable Documentation
- Reasonable Support
- Permissive License
- Long-Term Release Track Record
- Current Market Momentum

Software Selection (Open Source):

- + Clean Source Code
- + Clean Build Process
- + Open Source License

Software Selection (Library):

- + Non-Invasive Programming Model
- + Orthogonal Application programming Interface
- + Minimum/No Dependencies
- + Non-Copyleft Open Source License

Software Selection (Framework):

- + Orthogonal Application programming Interface
- + Adequate Dependencies
- + Non-Overlapping Scope
- + Non-Copyleft Open Source License

Software Selection (Tool):

- + Clean Deployment Procedure
- + Pleasant Command-Line Interface

Software Selection (Application):

- + Clean Deployment Procedure
- + Pleasant Graphical User Interface

Software Architecture Evaluation:

- Meets Functional Requirements
- Meets Non-functional Requirements
- Adequate Technology Overhead
- Single Dependency Direction
- Distance to State of the Art ("modern")
- Distance to Most Simple Approach ("adequate")
- Distance to Mainstream Approach ("mainstream")
- Documented Architecture Decisions ("rationales")
- Documented Architecture Views
- Documented Architecture Perspectives (NFR)

Focus Area Maturity Model

