Software Engineering Body of Knowledge (SWEBOK)

Software Engineering

Herstellung und Entwicklung von Software, der Organisation und Modellierung der zugehörigen Datenstrukturen und dem Betrieb von Softwaresystemen.

Die Drei Gesetze

1. **Humphrey's Law:** «Menschen wissen nicht, was sie wollen, bevor sie es sehen.»
2. **Ziv's Law:** «Softwareentwicklung ist unvorhersehbar und kann nie vollends verstanden werden.
3. **Conway's Law:** «Software ist ein Spiegel der Firma und der Menschen, die sie entwerfen.»

Gesetze der Softwareentwicklung

Conway's Law: *«Organisationen entwerfen Systeme, die ihre eigene Kommunikationsstruktur widerspiegeln».* Ein Softwareprodukt wird oft so modularisiert, wie das Team strukturiert ist.

Brooks's Law: *«Adding manpower to a late software project makes it later».* Neue Entwickler brauchen Einarbeitung und verursachen mehr Kommunikationsaufwand.

Parkinson's Law: *«Work expands so as to fill the time available for its completion»*. Wenn man einer Aufgabe mehr Zeit gibt, dauert sie auch länger, selbst wenn sie einfach wäre.

Sturgeon's Revelation: *«90 % von allem ist Müll»*. Kritisches Denken und gute Reviews sind nötig, um Qualität zu erkennen.

The Iceberg Fallacy: Das meiste der Software-Komplexität ist unsichtbar. Projektaufwand und Risiken werden oft unterschätzt.

The Peter Principle: Gute Entwickler:innen werden zu Teamleitern – obwohl sie dort vielleicht nicht glänzen. Beförderung ist nicht immer mit besserer Leistung gleichzusetzen.

Pareto-Prinzip (80/20 Regel): 80 % der Wirkung entstehen durch 20 % der Ursachen. Bsp:
- 20 % der Features liefern 80 % des Geschäftswerts.
- 20 % der Funktionen decken 80 % der Nutzung durch die Benutzer ab. → Feature-Priorisierung
- 20 % des Codes verursachen 80 % der Bugs → Refactoring and Bugfixing

Plangetriebene Methoden

Plangetriebene Methoden (z. B. das Wasserfallmodell, V-Modell) beruhen auf der Annahme, dass Anforderungen im Voraus bekannt sind und der Entwicklungsprozess systematisch in Phasen organisiert werden kann: Analyse → Design → Implementierung → Test → Wartung.

Vorteile: Hohe Planbarkeit und Nachverfolgbarkeit, Verlässlichkeit bei stabilen Anforderungen, Einfachere Risikoanalyse in frühen Phasen
Nachteile: Geringe Flexibilität bei Anforderungsänderungen, Wenig Kundenfeedback im Prozess

Geeignet für: Sicherheitskritische Systeme, Projekte mit fixen Anforderungen
Nicht geeignet für: Innovative oder experimentelle Projekte, UX-/Design-getriebene Produkte, Kurzzyklische Web- oder App-Entwicklung

Agile

Das *Ziel* eines Software-Projekts ist **flexibel** und bewegt sich
→ **Cone of Uncertainty:** viel Unschärfe am Anfang des Projekts.

Methodologies to make software development more predictive:
- More planning
- Requirements and scope
- Documentation before coding
- Strict change control

The Agile Manifesto (2001)

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentaion
- **Customer collaboration** over contact negotiation
- **Responding to change** over following a plan

Items on left are valued more than items on the right.

12 Principles: Customer Satisfaciton, Welcome Change, Continous Delivery, Working Together, Motivated Team, Face to Face, Working Software, Constant Pace, Good Design, Simplicity, Self Organisation, Reflect and adjust
Values: Transparency and Openess, Organizational Culture, Software Crafts
Technical Practices: Testing, Continuous Integration(CI), Clean Code

Risk (Risiko)

Risiko ist die Unsicherheit über zukünftige Ereignisse, die das Projekt negativ beeinflussen könnten. Agile Methoden wie Scrum, XP oder Kanban minimieren Risiken durch **kurze Iterationen.** Risiken werden **früh sichtbar.**

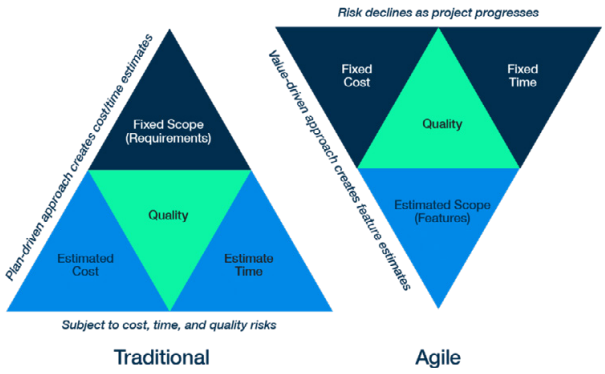
Cost of Change (Kosten der Änderung)

Beschreibt, wie teuer es ist, eine Anforderung oder ein Design nachträglich zu ändern.

Cost of Change steigt **exponentiell** über die Zeit für **plangetriebene** (klassische) Projekte. Späte Änderungen sind sehr teuer, weil sie viele Folgeänderungen auslösen

Agile Methoden (automatisierte Tests, Refactoring, inkrementelle Architektur) halten die Änderungskosten über die Zeit **niedrig.** Späte Änderungen sind erwartet und eingeplant (durch Product Backlog).

Iron Triangle and Paradigm Shift



Agile Estimation and Planning

Goals

- Reduce risk and uncertainty
- Establish trust and convey information
- Support better decision making

Planning becomes to a process of setting and revising goals that lead to a longer-term objective.

The Planning Onion

Metrics

Story Points are a relative measure of the complexity of a user story Represents the overall size of the story. Estimates the amount of effort involved in developing the feature.

Velocity is a measure of a team's rate of progress (Fortschrittsmessung). Calculated by summing each completed user story → Estimation of size can be turned into a schedule.

Estimations

Feature/Task (e.g. GitHub issue)
< **User Story** (set of features/tasks)
< **Epic** (large story)
< **Theme** (set of stories)

Let the team do the estimates (not the product owner).
Time estimates are not commitments.
Re-estimate only when we believe that a story's relative size has changed.
Plan for value.

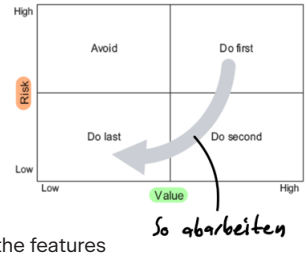
Cost, Risk and Prioritization

The **cost** of a feature is a huge determinant in the overall **priority** of a feature.

Risks: Schedule risk, Cost risk, Functionality risk.

There is rarely enough time to do everything, the **product owner** needs to **prioritize** what is worked on first.

Combine risk and value in **prioritizing** features:



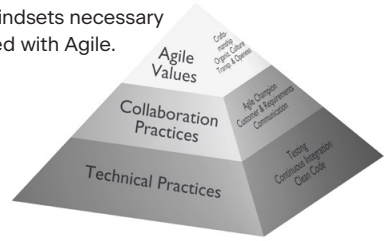
Primary factors
- Financial value of the features
- The cost of developing (and supporting) the features
- The significance of new knowledge created by developing the features
- The amount of risk removed by developing the features

Customer ≠ Enduser. Customer pays for the product, Enduser might not.

Tracking and Communication: Task Board (Kanban) or Burndown Chart

Pyramid of Agile Competencies

Illustrates the layers of skills and mindsets necessary for teams and individuals to succeed with Agile.



Technical Practices
Absolute musts: **Clean Code** and **Automated Testing** for providing quality software, **Continuous Integration** to deliver software with high frequency

eXtreme Programming (XP)

XP gives **principles** and **practices** in order to deal with risks (schedule, business change, false feature, staff turnover)

4 external Forces (Constraints)

Time, Ressources, Scope and **Quality**
Clients set set three contraints → The development team sets the fourth

Core Values

Communication, Simplicity, Feedback, Courage, Respect, Embrace change, Incremental Change (small Adjustments/Releases)

Principles

Rapid Feedback, Incremental Change, Assume Simplicity (Kompliziertheit vermeiden), Embrace Change, Quality Work

Practices

Test-Driven Development (TDD; Erst Tests, dann Code), Self-Organized Team, Continuous Integration (automatisiertes Testen), Pair Programming, Continous Refactoring, Simple Design (nur das notwendes implementieren).

Slack is a small, intentional buffer of time built into an iteration or sprint to handle unexpected work/issues and improvements.

Spike is a time-boxed (e.g. 1–2 days, or every friday afternoon) research task used for exploration and learning. Spikes typically do not produce shippable code, but rather **knowledge, prototypes, recommendation** or **decision.**

Planungspraktiken

- **Planning Game:** Gemeinsames Planen von Releases und Iterationen mit dem Kunden.
- **Small Releases:** Häufige, nutzbare Softwareversionen liefern.
- **Sustainable Pace:** Keine Überstunden – langfristig produktiv arbeiten.

Software Craft / Craftsmanship

Coding Dojo

A collaborative learning environment where developers come together (physically or virtually) to **practice coding problems** in a safe, non-competitive setting.
Purpose: Improve programming skills, team collaboration, share knowledge, build a culture of continuous improvement, practice clean code

Code Kata

A short, repeatable **coding exercise** designed to help developers improve skills through repetition and reflection. Examples: FizzBuzz, Game of Life, Roman Numerals Converter etc.

Tidy First

Das Buch «Tidy First?» von Kent Beck ist eine praxisnahe Abhandlung über Softwaredesign, Codequalität und den Umgang mit technischer Schulden.

Aufräumen vor oder nach der Änderung?

Zentrale Frage des Buches: Sollte man zuerst den Code aufräumen (tidy), bevor man ihn ändert, oder erst ändern und danach aufräumen?
Antwort: Es kommt darauf an. Der Autor beschreibt, wann welches Vorgehen sinnvoll ist – abhängig von Kontext, Risiko und Veränderungsart.

Tidyings vs. Design Changes

Tidyings: Kleine, strukturverbessernde Änderungen ohne funktionale Auswirkungen (z.B. Umbenennen, Extrahieren von Methoden). Tidyings sind inkrementelle, sichere Schritte, die helfen, Komplexität zu reduzieren.
Design Changes: Änderungen an Struktur und Verhalten, die das Design grundlegend verändern können.

Tidyings als iterative Vorbereitung

Beck schlägt vor, mit kleinen Tidyings zu beginnen, um den Code **lesbarer, verständlicher** und **leichter änderbar** zu machen. Tidyings sind **inkrementelle, sichere Schritte,** die helfen, **Komplexität zu reduzieren.**

Cognitive Load reduzieren

Aufgeräumter Code senkt die mentale Belastung (Cognitive Load) und hilft dabei, Probleme klarer zu erkennen.

Experimentieren mit Design

Design wird als ein Experimentierprozess verstanden. Durch häufiges Aufräumen kann man besser sehen, welche Designs sich „organisch ergeben“ und welche Anpassungen notwendig sind.

Mechanische vs. kreative Änderungen

Tidyings sind mechanisch (also gut automatisierbar oder reproduzierbar). Designänderungen sind kreativ (erfordern Nachdenken, Entscheidungen). Kent Beck empfiehlt, mechanische Änderungen zu isolieren, um kreative Änderungen klarer zu sehen.

Design als ökonomische Aktivität

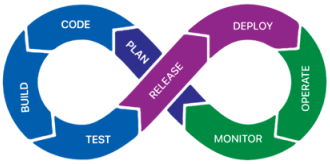
Designentscheidungen sind wirtschaftlich motiviert: Es geht nicht um "perfekten" Code, sondern um verhältnismäßige Investitionen in Verständlichkeit und Wartbarkeit.

DevOps

DevOps is about automation and enables development teams to continuously deliver value to their customers.

Definition: DevOps is a methodology integrating and automating the work of software development (Dev) and information technology operations (Ops).

The DevOps Livecycle – 4 Phases



1. **Idea and Planning:** Gather requirements and feedback
   Stack: e.g. GitHub Issues and Project Boards
2. **Development: Continuous integration (CI),** automated tools to turn code changes into builds, run tests against required checks, and merge and prepare code for deployment, Version Control (VCS, e.g. git), cloudbased development, virtual environments, ongoing changes and code review
   Stack: e.g. GitHub Codespaces, GitHub Actions
3. **Deployment: Continuous delivery (CD)** and **Continuous deployment,** Tools to automatically push code changes to a non-production testing or staging environment. Operations teams can immediately deploy the changes to production.
   Stack: e.g. GitHub Actions, GitHub Packages, Microsoft Azure
4. **Learn:** Operations teams keep an eye on releases with monitoring tools that measure performance, ensure stability and uptime, gather customer feedback, and stay in close contact with developers to push required fixes
   Stack: e.g. New Relic, Netdata, Lightstep, App Dynamics, Sentry, Raygun, Honeycomb, Splunk

4 Key Metrics (DORA)

1. **Deployment Frequency:** Average number of daily finished code deployments to any given environment
2. **Lead Time for Changes:** Time between acceptance and deployment
3. **Time to Restore Services:** How long it takes to restore a service or recover from a failure
4. **Change Failure Rate:** How frequently deployments fail

Software Automation

Ways to automate
- **On-Demand:** Run a script or press a button
- **Scheduled:** Run a script at certain times (nightly builds)
- **Triggered:** On certain events (commit/push → GitHub actions)

Types of Automation
- **Build:** Compiling source code into binary code, Software packaging
- **Test:** Automated Unit-, Integration-, Acceptance-Tests
- **Deployment:** Automated deployment to Test or Production Environments
- **Operation:** Infrastructure provisioning, Monitoring, Scaling

Goals
- **Improve Product Quality:** Automated testing, Automated Code auditing, Documented history of builds to verify issues)
- **Faster Time To Market (Cycle Time):** Accelerate the process, Immediate feedback
- **Minimize Risks:** Time to Restore Services, Change Failure Rate, find broken build fast and early. awareness of current software status. Reduce Risk by releasing often.

Scrum

Scrum is an agile team collaboration framework.

Goals

- Focus on delivering the highest business value in the **shortest time.**
- **Rapidly inspect** actual working software (e.g. every two weeks.)

Scrum in a Nutshell

- **Split** your **organization** into small, cross-functional, **self organizing teams.** The business sets the priorities. but teams self-organize to determine the best way to deliver the highest priority features.
- **Split** your **work** into a list of small, concrete **deliverables.** Sort the list by **priority** and estimate the relative effort of each item.
- **Split time** into short fixed-length iterations (usually 1–4 weeks), with potentially shippable code demonstrated after each iteration. Every two weeks to a month anyone can see real working software and **decide** to release it as is or continue to enhance it for another sprint.

Key Components

Sprints (= Iterations):
Breaks work into goals, completed within time-boxed iterations (usually 1-4 weeks long). Optimize the release plan and update priorities in collaboration with the customer, based on insights gained by inspecting the release after each iteration (sprint).

Scrum Team
Typically 5–9 people

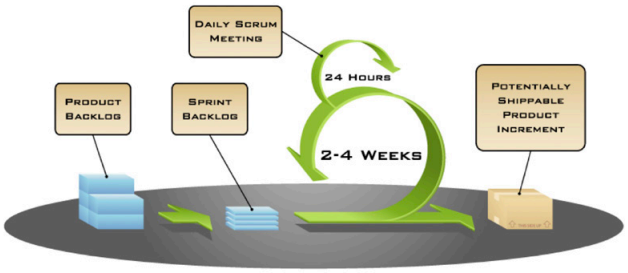
- **Product Owner:** Defines the product vision and features, prioritizes the backlog. Accepts or rejects the result.
- **Scrum Master:** A person in charge of a scrum team. Ensures team is functional and productive. Moderates in the team, and helps to organize. Optional for small teams.
- **Developers:** Build and deliver the product increment. Also create a plan for the sprint.

Artifacts

- **Product Backlog:** A prioritized list of features, enhancements, and fixes. The **Product Goal** describes which serve as a target.
- **Sprint Backlog:** A subset from the product backlog selected for the sprint. Product Backlog item meets **Definition of Done (DoD)** → new Increment
- **Increment:** The working product/feature delivered at the end of a sprint.

Ceremonies (Events)

- **Sprint Planning:** Define the sprint goal and backlog items for the sprint.
- **Daily Scrum (Stand-up Meeting):** A short daily meeting (15 minutes) to sync progress and remove blockers.
- **Sprint Review:** Every 2–4 weeks. Demonstrate completed work to stakeholders. Deciding whether to accept the release or to continue work on it. A collaborative working sesison. No slides, only working software. 1/2 to 1 hour preparation.
- **Sprint Retrospective:** Reflect on the sprint and identify improvements. Purpose is to find ways to increase quality and effectivness. Done after every sprint. 15–30 minutes. Whole team participates. Optimize the process by having a retrospective after each iteration.



Daily Scrum

Everyone answers three questions:
1. What did you do yesterday?
2. What will you do today?
3. Is anything in your way?

Tools

Kanban and **Burndown Chart/Increment.**

Definition of Done (DoD) for a User Story

Example:
- Unit tests pass and coverage met standard (85% or above)
- Non-functional tests pass (scalability, reliability, security, etc.)
- Code is reviewed and coding standards are met
- Continuous integration implemented (automated build, deployment testing)
- Necessary documentation is completed

Definition of Ready (DoR) for a User Story

Example:
- User Story defined
- User Story Acceptance Criteria defined and dependencies identified
- User Story sized by Delivery Team
- Performance criteria identified (where appropriate)
- Person who will accept the User Story is identified

Software Architecture / Applikationsarchitektur

An architecture is a set of significant **decisions** about the **organization of a software system,** the selection of individual structural elements and their interfaces, and their behavior → **It is about the Organisation of code**

Complexity

Software is probably the most **complex** thing ever created by human hands.
Complex ≠ Complicating
Complexity: Made up of many interconnected parts
Complicating: Emerging systems (e.g. flocking), hard to predict/understand

Conformity: The interfaces of the system to other systems varies
Changeability: Software is subject to perpetual change.
Invisibility: Software is invisible and cannot be visualized directly.

Architectural Drivers

Driver 1: Non-functional behavior, crosscutting concerns

- **Measurable at Runtime:** Performance, Security, Availability, Usability, Robustness
- **Incapable of Measurement:** Scalability, Integrability, Portability, Maintainability, Testability, Reusability

Driver 2:

Types of Architecture

Infrastructure, Security, Network Data, Hardware, Application, Database, System, etc.

Building Blocks: Components, Layers, Packages, Namespace

System Architecture: Its about understanding both **software** and **hardware** and their relationship.

Software Architecture
Defines the pieces that compose **an application:** Logging, Exception handling, security (authentication, authoristaion and confidentiality), Performance, scalability, availability, Audit and regulatory requirements, Interoperability/integration. Metaphor: Building architecture

Enterprise Architecture
Defines ways how an enterprise uses **many applications.**
Metaphor/Metapher: City planning

Measurements

Measurable at Runtime: Performance, Security, Availability, Usability, Robustness
Incapable of Measurement: Scalability, Integrability (Integrierbarkeit), Portability, Maintainability, Testability, Reusability

Principles

Modularity: Components of a design can easily exchanged
Portability: When software or parts of it can run in other environments
Changeability: Its easy to make changes to the system
Conceptual Integrity: Similar parts of a system should be designed similarly
Intellectual Control: Software should be understood by those responsible
Buildability: Software design must specify a target system so that it can be realized by any team in a given time
Low Coupling: components have minimal dependencies on each other.
High Cohesion: Elements within a unit perform a single, well-defined task.
Encapsulation: Dependent modules are combined into larger units
→ Design for Change

Business (Domain) Logic

Code that determines how **data** is **created, displayed, stored,** and **changed.**

Examples: Calculating discounts, Determining shipping cost, Preventing duplicate booking, etc.

Tiers ans Layers

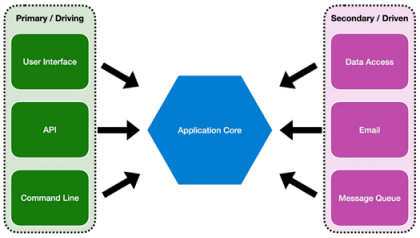
Tier: Physical Deployment across machines, processes, or servers.
The 3 tiers **presentation, logic** and **backend** contain the 7 logical layers.

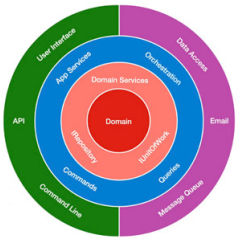
Layer: Logical Organization of responsibilities within a software system.
Layers: 1. Presentation Layer (Viers), 2. Application Layer (Authentication, etc.), 3. Communication Layer (Transfer Objekt), 4. Business Service Layer (Authentication Manager, User Manager, etc.), 5. Business Object Layer, 6. Integration Layer, 7. Backend Systems (Database).

Distribution of Work and Layering

Client-Server systems are divided into three **structurally categories:**
- **Presentatin**
- **Function**
- **Data**

Port-Adapter Architecture
Also called *Onion-* or *Hexagonal-Architecture*





Arrows depicts dependencies.

The Core logic has no dependencies and no references to the outside world.
The Logic Layer (inside the core) defines Interfaces (aka Ports).
Secondaries implement that Port (Interface) by using an Adapter.
Primaries make calls into the core. Secondaries are driven by the Core.
Dependencies inside the Core always point to the inside.

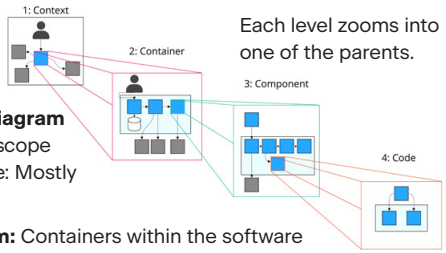
Nachteil: Viel Mapping ist nötig.

Architecture Styles

Architektur Stil	Anwendung	Vorteil	Nachteil
Independent Components			
Communicating Processes	Parallel processing	Simple modeling, scalability	Complexity of the individual elements
Event Systems	GUI's, Real Time Systems	Independence of elements, change-friendliness	Non-deterministic behavior of the elements
Call-and-Return			
Main Program & Subroutine	Structured Programming, Client-Server (RPC)	Defined control flow	Scalability, expandability
Object Oriented	General design, client-server	Universal	Complexity, freedom of application
Layered	SOA, Multi-Tier Architectures	Conceptual integrity, locality of changes	Performance, complexity
Virtual Machine			
Interpreter	Processor and operating system simulation	Portability, flexibility	Performance
Rule-Based Systems	Expert systems	Flexibility through rules and regulations	Complexity, performance
Data Flow			
Batch Sequential	Host Systeme	Data control	Flexibility, interaction
Pipes and Filters	Software Converter, Compiler	Flexibility, distribution	Complexity
Data Centered			
Repository	Master Data Management	Simple	Single Point of Failure
Blackboard	Data-driven control systems	Scalability	Application limited

Architecture Documentation

C4 Model



Each level zooms into one of the parents.

Level 1 – System Context Diagram
Shows the software system scope and who is using it. Audience: Mostly non-technical people

Level 2 – Container Diagram: Containers within the software scope. Audience: Technical people inside and outside the development team.

Level 3 – Component Diagram: Components within the container. Audience: Software architects and developers.

Level 4 – Code diagram: A single component (e.g. UML diagram) Audience: Software architects and developers.

Architecture Canvas

Dient, die Architektur einer Softwarelösung (Architekturplanung) strukturiert, übersichtlich und kollaborativ zu dokumentieren und zu kommunizieren.

Bereiche: Business goals, Stakeholder, Funktionale Anforderungen, Technologiestack (Frontend, Backend), Laufzeitumgebung, Infrastructur und Deployment, Schnittstellen, API und Integration, Security und Compliance, Risiken, Monitoring und Analytics, etc.

Architecture Patterns

Event Sourcing

Event Sourcing speichert **jede Zustandsänderung** als **unveränderliches Event** in zeitlicher Reihenfolge.
Der **aktuelle Zustand** ergibt sich aus der **Abfolge aller Events.**

Vorteile: Built-In Log/History, Nachvollziehbarkeit, Debugging, Fehlerkorrektur, Passt gut zu DDD (Domain-Driven-Design)

Performance, Skalierbarkeitund Concurrency: CRUD stösst bei hoher Last an Grenzen (Locks, Latenz). Event Sourcing ist atomar (insert-only)

Audit und Nachvollziehbarkeit: CRUD speichert nur den aktuellen Zustand → Historie geht verloren. Event Sourcing speichert Änderung mit Timestamp

Nachteile: Replay der Events notwendig → **Projections (Snapshots),** Joins auf andere Objekte nicht ganz trivial

Monolith, Self-Contained Systems, Microservices and Serverless

Monolith: Eine einzige, zusammenhängende Anwendung.
Keine Trennung zwischen Modulen zur Laufzeit.

Self-Contained Systems (SCS)
Mehrere kleine modulare und unabhängige Software-Systeme.
Funktionen sind in mehreren unabhängigen Systemen separiert.

Microservices: Bestehen aus vielen kleinen, lose gekoppelten Diensten/ Prozessen. **Kommunikation** zwischen den Diensten **via APIs über Netzwerk.**

Serverless: Kosten basierend nur auf tatsächlichem Ressourcenverbrauch. Skaliert automatisch basierend auf Auslastung. Verwaltung physischer Server wird an ein Cloudprovider (e.g. AWS) ausgelagert. Ausführung von Funktionen / Gebrauch von Ressourcen nur bei Bedarf (Pay-Per-Use).

Strangler and Online Migration

Strangler
Schrittweise Ablösung von Altsystemen. Paralleler Betrieb von alt und neu.

Online Migration
Bestehende Systeme (z.B. Cloud-Migration oder Datenbankmigration von MySQL zu PostgreSQL) im **laufenden Betrieb** ohne Downtime umstellen mit nathlosem Übergang für die Enduser.

Datenbanken werden parallel betrieben mit Synchronisation (Dual Writes und Shadow Traffic). Ermöglicht einen **sicheren Rollback.**

Circuit Breaker, Retry and Bulkhead

Ausgangslage: Kurzfristige Verbindungs-/Serviceunterbrüche von unabhängigen Microservices → **Resilience Patterns** zur Aufrechterhaltung des Systems.

User → Website → **Circuit Breaker/Retry/Bulkhead (= Proxy)** → Service

Circuit Breaker: Wenn Grenze von Fails bei Serviceaufruf überschritten wurde, werden Calls direkt abgelehnt → kein Warten auf Service Timeouts. Nach Timeout werden Requests langsam durchgelassen.

Retry: Strategien: Retry-Immediately, Retry-After-Delay oder Cancel

Bulkhead: Isoliert einzelne Services und schützt das Gesamtsystem vor Überlastung.Isoliert Nutzer von Service-Überlastungen anderer Nutzer.

Command Query Responsibility Segregation (CQRS)

Separate Modelle für Lese- und Schreibeoperationen:

- **Command:** Veränderung eines internen Zustandes, ohne Rückgabewert.
- **Query:** Etwas zurück geben, ohne Veränderung eines internen Zustandes.

Jedes Modell kann unabhängig optimiert und skaliert werden.
Separation of Concerns.

Data Access Patterns

Transaction Script: Organizes and divides the business (domain) logic into individual procedures so that each procedure covers a single request from the Presentation Layer.

Domain Model: Describes an object model of the problem domain that includes behavior and data.

Table Module: Describes a single instance (singleton) that encapsulates the business logic for all rows in a database table or view.

The Cynefin Framework

Das Cynefin Framework (pronounced *kuh nev in*) ist ein Modell, das helfen soll, zu verstehen, welches Verhalten in bestimmten Kontexten zum Erfolg führt. Es basiert auf der Komplexitätstheorie.

If you want people to change, change the system they work in, not the people themselves.

Concepts and Terms

Cynefin is a Welsh word that signifies the multiple factors in our environment and our experience that influence us in ways we can never understand.

Constructive irritants people see a problem and courageously bring it into the light and **offer concrete solutions.** Contrary of a complainer. They are finger pointers and focus on everything that isn't working.

Non linear: Describes something that does not progress or develop smoothly from one stage to the next in a logical way Instead. Kleine Änderungen der Eingaben, physikalischen Interaktionen oder Stimuli können grosse Effekte oder sehr signifikante Änderungen der Ausgaben verursachen.

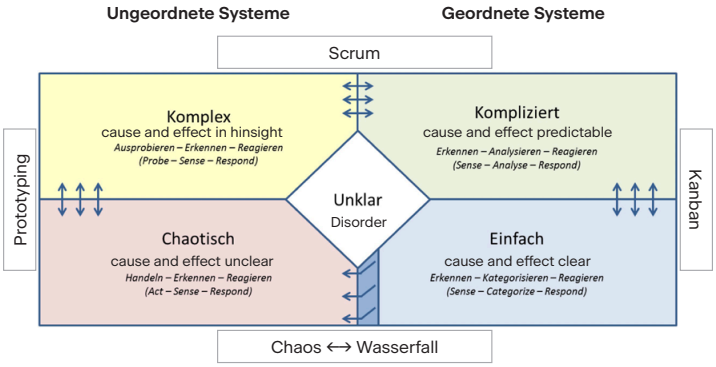
Complexity Theory

Exaptation: The process by which features acquire functions for which they were not originally adapted or selected.

Coherence: Sits between certainty and random speculation. It's about plausibility, not proof. It helps us act wisely without needing complete certainty. Most of the time we have to make decisions based on evidence which is coherent bu tnot absolute.

Coevolution: As one thing interacts with another thing, patterns form (coevolve) from the interaction. Once a pattern is formed, you can't go back.

The 5 Cynefin Domains



Codefin

Einfach: Checkliste. It's possible to fall from simple to chaotic, but not the other way around. Wasserfall-Approach (v-model)
Kompliziert: Prozess ist klar definiert
Komplex: Prozess muss erlernt werden
Chaotisch: Neuer Prozess muss gelernt/entwickelt werden

Komplexes Adaptives System (CAS)

Definition: Ein komplexes adaptives System ist ein dynamischen Netzwerk mit vielen Akteuren (Zellen, Individuen, Firmen, Nationen, soziale Netzwerke, Stromnetze, Tierschwärme), die parallel agieren, und ständig agieren und reagieren auf das was die anderen Akteure machen. Die Kontrolle eines komplexen adaptiven Systems tendiert dazu, verstreut und dezentralisiert zu sein. Wenn es ein zusammenhängendes Verhalten im System geben soll, muss dies aus dem Wettbewerb und der Kooperation der Akteure kommen. Das Verhalten des gesamten Systems ist das Resultat einer großen Anzahl von Entscheidungen, die von vielen einzelnen Agenten getroffen werden. *John H. Holland*

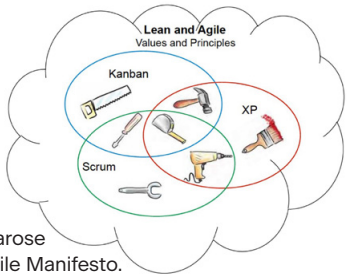
Beispiele: Klima, Städte, Firmen, Märkte, Regierungen, Industrien, soziale Netzwerke, Stromnetze, Tierschwärme, Verkehrsströme, etc.

CAS ist ein System, das insofern komplex ist, als es ein dynamisches Netzwerk von Interaktionen darstellt, aber das Verhalten des Ganzen möglicherweise nicht entsprechend dem Verhalten der einzelnen Komponenten vorhersehbar ist.

Adaptiv bedeutet in diesem Kontext, das das System auf Mirko Ereignisse reagiert, in dem es sein individuelles und kollektives Verhalten anpasst sich selbst neu organisiert. Die Interaktionen in einem CAS sind **nichtlinear.**

Kanban and Lean Software Development

Overview



Lean and **Agile** are two compatible **principles** that outline how to succeed with **product development.**

Scrum, XP, Lean and **Kanban** are **concrete ways (process tools)** of putting these principles into practice.

Lean arose from manufacturing. **Agile** arose from software development and the Agile Manifesto.

Adaptive vs. Prescriptive (less/more rules to follow):
adaptive ← Kanban (3 rules), Scrum (9), XP (13), RUP (120+) **→ prescriptive**

Agile

We know what Agile meant in 2001
(through the **Agile Manifesto**). Today it is less clear.

Lean

Lean is the western term for what the Japanese call TPS (Toyota Production System)—an approach to manufacturing that has helped make Toyota a very successful car manufacturer.

Lean means **Elimination of waste** (minimize time, capital investment, cost) and **Continuous Improvement** to (1) create **process speed** and (2) improve **efficiency** and **quality.** Lean is an implementation of the Agile Manifesto.

Principles

- **Eliminate Waste:** Remove all wastes that add no value to project
  1. Code Development (Partially completed work, Defects)
  2. Project Management (Unnecessary documentation, loss of knowledge)
  3. Workforce Potential (Task Switching, multi-tasking, don't let devs wait)
- **Amplify Learning** (through feedback)
- **Decide as late as possible** (until they can be made based on facts)
- **Deliver as fast as possible** (when the customer asks for it)
- **Empower the team** (let them decide on things)
- **Build integrity** (den eigenen Werten treu bleiben und danach handeln)

Kanban

A delivery **flow system** that limits the amount of work in progress (WIP) by using **visual signals (cards / boards).**

Practices

- **Visualize the Workflow:** Seeing the work and the policies
- **Split the work into pieces:** Write each item on a card and put on the wall (named columns where each item is in the workflow)
- **Limit Work In Progress (WIP):** Assign explicit limits to how many items may be in progress at each workflow state.
- **Measure the Lead Time (Cycle Time)** = avg. time to complete one item. Optimize the process to make lead time as small as possible.
- Weitere: Make Policies Explicit, Evaluate Improvement Opportunities

Kanban Values

Transparency (sharing information), Balance, Collaboration, Customer Focus (Knowing the goal), Flow, Leadership (at all levels), Understanding (self-knowledge), Agreement (to move together toward goals), Respect and Understanding

Scrum vs. Kanban

Scrum	Kanban
Timeboxed iterations prescribed.	Timeboxed iterations optional.
Team **commits to a specific amount of work** for this iteration.	Commitment optional.
Items must be broken down so they can be **completed within one sprint.**	No particular item size.
Estimation prescribed	Estimation optional.
Burndown chart prescribed.	No particular type of diagram. → Cumulative Flow Diagrams (CFD)
Prescribes 3 **roles** (PO/SM/Team).	No roles
Prioritized product backlog	Prioritization is optional.
Cannot add items to ongoing iteration.	Can add new items whenever capacity is available.