SWEN 2 Summary (beliebig lang)

21. Februar, 2024; rev. 16. Juni 2025

Linda Riesen (rieselin)

1 Vorlesung 01: Software Manifesto

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity - the art of maximizing the amount of work not done - is essential.
- The best architectures, requirements, and designs emerge from selforganizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

2 Vorlesung 02: eXtreme Programming (XP)

Goal: Keep cost of change low

XP Variables and Constants

- Scope => Variable (actively managed by Customer)
- Quality => non-negotiable
- Budget (Resources) => usually fix
- Time => usually fix (Number of fix-length iterations)

XP Values

- Communication
- Simplicity
- Feedback
- Courage
- Respect

XP Practices

- The Planning Game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming

- Collective ownership
- Continuous integration
- 40 hours week
- On-site customer
- Coding standards

2.1 Story Points

Used to estimate Aufwand as estimation in relation to other Aufwand instead of exact.

- Story points estimates the amount of effort involved in developing the feature, the complexity of developing it, the risk inherent in it, and so on
- Story points are a relative measure of the complexity of a user story

2.1.1 Velocity

- Velocity is a measure of a team's rate of progress
- Velocity is calculated by summing the number of story points assigned to each user story that the team completed during the iteration
- Best guess is, that the team will complete similar amounts of story points per iteration

3 Vorlesung 03

3.1 User Story

consists of Card, Conversation and Confirmation (details, satisfaction conditions)
As a [user role], I want to [goal] so that [benefit] (there is more than 1 user!)

3.2 Requirements

- Written Requirements (time consuming, must be maintained, easier shared / reviewed / edited / misinterpreted)
- Verbal Requirements (quick, easily adapted, clarified => not good documented)

Agile Requirements

- Active user involvement is imperative
- Agile teams must be empowered to make decisions
- Requirements emerge and evolve as software is developed
- Agile requirements are 'barely sufficient'
- Requirements are developed in small, bite-sized pieces
- Enough's enough – apply the 80/20 rule
- Cooperation, collaboration and communication between all team members is essential

		Dysfunctional Question				
		Like	Expect	Neutral	Live with	Dislike
Functional Question	Like	Q	E	E	E	L
	Expect	R	I	I	I	M
	Neutral	R	I	I	I	M
	Live with	R	I	I	I	M
	Dislike	R	R	R	R	Q

M	Must-have	R	Reverse
L	Linear	Q	Questionable
E	Exciter	I	Indifferent

Abbildung 1: Categorizing a feature from answers to a pair of questions

3.3 Planning

- Estimating and planning are critical, yet are difficult and error prone
- Overplanning and doing no planning are equally dangerous
- A good plan is one that is sufficently reliable that it can be used as the basis for making decisions about the product and the project
- Agile planning is focused more on the planning than on the creation of the plan, encourages change, results in plans that are easily changed, and is spread throughout the project
- should be done by team, not product owner
- User Stories should be prioritized by value (financial, dev cost, risk, new knowledge learnt)

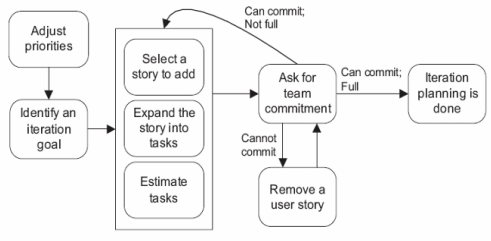

Abbildung 2: Commitment-Driven Iteration Planning

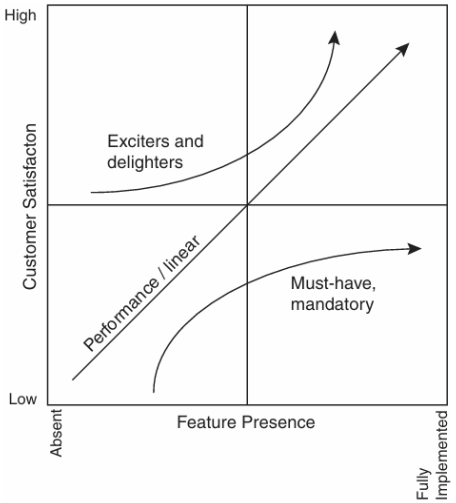

Abbildung 3: The Kano model of customer satisfaction

4 Vorlesung 04: DevOps

DevOps is a methodology integrating and automating the work of software development (Dev) and information technology operations (Ops).
Created to help the agile instead of the linear approach

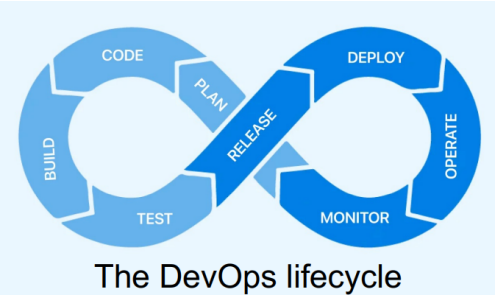

Abbildung 4: DevOps Lifecycle

4.1 DevOps Metrics

- Deployment Frequency: average number of daily finished code deployments to any given environment
- Lead Time for Changes: The amount of time between acceptance and deployment
- Time to Restore Services: How long it takes to restore service — or recover from a failure
- Change Failure Rate: How frequently deployments fail

4.2 Software Automation

- **On-Demand** run a script or press a button
- **Scheduled** at certain times => nightly builds
- **Triggered** on certain events => commit/push to VCS

Types of Automation

- **Build Automation** compile to binary, package, create documentation / release notes
- **Test Automation** automated unit/integration/acceptance tests
- **Deployment Automation** automated deployment to test/production env.
- **Operation Automation** infrastructure provisioning, monitoring, health management, scaling

Goals of Automation

- Improve Product Quality
- Faster Time To Market (Cycle Time)
- Minimize Risks (Time to Restore Services, Change Failure Rate)

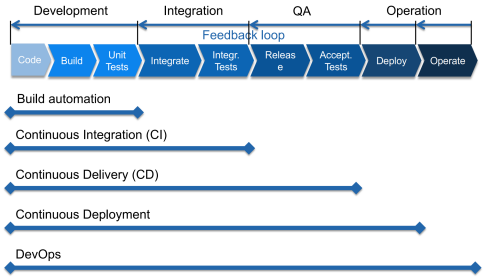

Abbildung 5: Software Automation Pipeline

4.3 Continuous Integration (CI)

- Everyone Pushes Commits To the Mainline Every Day!
  - «Integration is primarily about communication. Integration allows developers to tell other developers about the changes they have made. Frequent communication allows people to know quickly as changes develop.»
  - General rule of thumb
- Fix Broken Builds Immediately (Continuous Integration can only work if the mainline is kept in a healthy state)
- Keep the Build Fast (< 10 min)
- Hide Work-in-Progress (Latent code may never be executed in production, but that doesn't stop it from being exercised in tests.)
- Test in a Clone of the Production Environment

4.3.1 Patterns for Continuous Integration

Keystone Interface A central component that other services rely on, ensuring stability and backwards compatibility during integrations. It acts as a stable API or service that abstracts underlying changes, minimizing disruptions.

Dark Launching Deploying new features to production without making them visible to users, allowing for performance testing and gradual activation. This technique helps identify issues before a full-scale release and ensures minimal user impact.

Feature Flags Using toggles to enable or disable features dynamically, facilitating controlled rollouts and quick rollbacks. This approach allows developers to test new functionalities in production and gradually expose them to users without redeploying code.

Blue-Green Deployment Running two environments (Blue and Green) to enable zero-downtime deployments by switching traffic between them. The Blue environment runs the live version, while the Green environment hosts the updated version. After successful testing, traffic is switched from Blue to Green.

Canary Releases Releasing new versions to a small subset of users before full deployment, allowing for monitoring and minimizing risks. This method helps detect potential issues early by gradually increasing the exposure of the new version while keeping rollback options open.

4.4 Virtual Environments

- Virtualization is a set of technologies that create virtual versions of physical computing resources.
- It uses software to simulate hardware functions, allowing users to run multiple virtual machines (VMs) on a single physical machine.
- Containerization
  - A container is an isolated, resource controlled, and portable operating environment, where an application can run without touching the resources of other containers, or the host.
  - A container looks and acts like a newly installed physical computer or a virtual machine.
  - A container runs an operating system, has a file system, and can be accessed over a network as if it were a physical or virtual machine
- Infrastructure as Code (IaC)
  - managing and provisioning IT infrastructure using code (script) instead of manual configuration.
  - Store the infrastructure files in a VCS
- Microservices Architecture: A microservices architecture breaks monolithic application into a collection of loosely coupled services

5 Vorlesung 05: Scrum

- Scrum is an agile process that allows us to focus on delivering the highest business value in the shortest time.
- It allows us to rapidly and repeatedly inspect actual working software (every two weeks to one month).
- The business sets the priorities. Teams self-organize to determine the best way to deliver the highest priority features.
- 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
- Scrum teams do a little of everything all the time (Requirements, Design, Code, Test)
- No Changes during a Spring

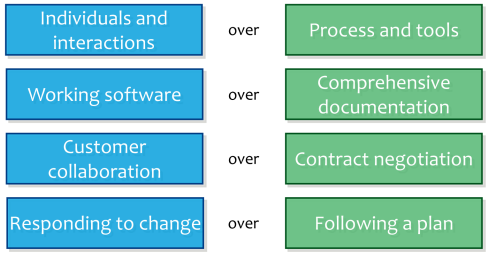

Abbildung 6: Agile Manifesto Values

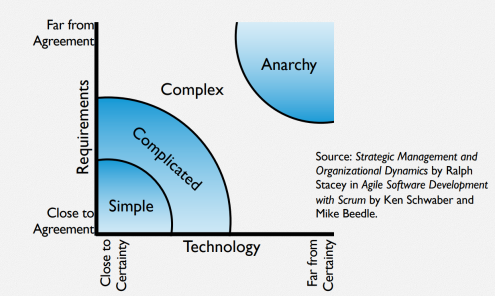

Abbildung 7: Project noise level

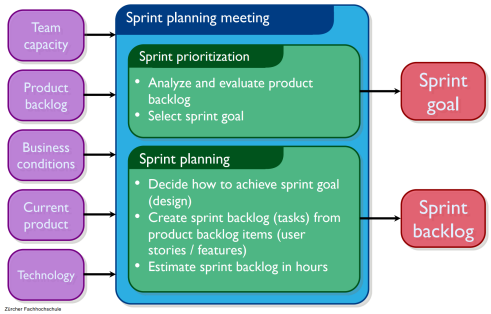

Abbildung 8: Sprint Planning

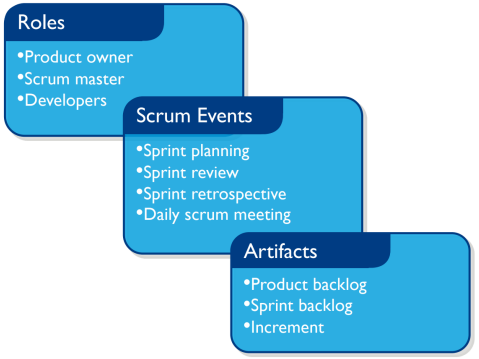

Abbildung 9: Scrum Ingredients

6 Vorlesung 06: SoftwareCraftsmanship

6.1 Dojos und Katas

Dojos und Katas sind bewährte Methoden zur Verbesserung der Programmierfähigkeiten durch kontinuierliches Üben. Katas sind kleine, wiederholbare Programmieraufgaben, die Entwicklern helfen, bestimmte Techniken zu verfeinern und automatisierte Lösungen zu entwickeln. Dojos hingegen bieten eine kollaborative Lernumgebung, in der Entwickler gemeinsam an Problemstellungen arbeiten, verschiedene Ansätze vergleichen und bewerten. Diese Methoden fördern die Codequalität und das tiefere Verständnis für Softwareentwicklungsmuster.

6.2 Softwarepatterns

6.2.1 CQRS (Command Query Responsibility Segregation)

CQRS trennt Lese- und Schreiboperationen in verschiedene Modelle, um Skalierbarkeit, Performance und Wartbarkeit zu verbessern. Während Leseoperationen oft optimiert und in eigenen Datenbanken gespeichert werden, fokussieren sich Schreiboperationen auf die Validierung und Geschäftslogik. Dies führt zu einer effizienteren Nutzung von Ressourcen und einer besseren Anpassung an komplexe Geschäftsanforderungen.

- Vorteile
  - Bessere Skalierbarkeit (read und write getrennt optimierbar)
  - Flexiblere Datenmodelle
  - Erhöhte Performance (durch spezialisierte Modelle)
  - Implementierung komplexer Logik
  - Fehlerisolierung (read verfügbar, wenn write ausfällt)
- Nachteile
  - Erhöhter Aufwand in Architektur und Implementierung
  - Mehr Ressourcen und Infrastruktur erforderlich
  - Daten sind verzögert konsistent (Synchronisation zwischen read und write)
  - Debugging und Testing (ganzes Sytem)

6.2.2 Event Sourcing

Event Sourcing speichert Änderungen an einem System als eine Reihe von Ereignissen. Dadurch kann der Zustand eines Systems jederzeit rekonstruiert und eine bessere Nachvollziehbarkeit gewährleistet werden. Dies erleichtert das Debugging, Auditing und ermöglicht eine flexible Reaktion auf Änderungen, da vergangene Zustände leicht wiederhergestellt werden können.

- Vorteile:
  - Vollständige Historie und Auditierbarkeit
  - Flexibilität bei der Zustandsermittlung
  - Unterstützung für komplexe Geschäftslogik
- Nachteile:
  - Erhöhte Komplexität in der Implementierung
  - Potenzielle Performance-Herausforderungen

6.2.3 Strangler Pattern / Online-Migration

Das Strangler Pattern ermöglicht eine schrittweise Migration von monolithischen Systemen zu modernen Architekturen, indem neue Funktionalitäten parallel entwickelt und alte schrittweise ersetzt werden. Dies reduziert Risiken, da der Wechsel inkrementell erfolgt und sich das System stetig weiterentwickeln kann, ohne dass eine vollständige Neuentwicklung erforderlich ist.

6.2.4 Circuit Breaker / Bulkhead / Retry

- **Circuit Breaker:** Verhindert Systemüberlastung durch Unterbrechung fehlerhafter Prozesse. Wenn ein bestimmter Fehlerprozentsatz überschritten wird, werden weitere Anfragen vorübergehend blockiert, um das System zu schützen.
- **Bulkhead:** Isoliert Systemkomponenten, um Fehlerausbreitung zu minimieren. Durch Begrenzung der Ressourcennutzung pro Dienst können einzelne Fehler nicht das gesamte System beeinträchtigen.
- **Retry:** Automatisches Wiederholen fehlgeschlagener Anfragen zur Erhöhung der Ausfallsicherheit. Dies ist besonders nützlich bei vorübergehenden Netzwerkfehlern oder überlasteten Diensten.

6.2.5 Vergleich von Architekturen

- **Serverless:** Skalierbare, ereignisgesteuerte Architektur ohne Verwaltung von Servern. Entwickler konzentrieren sich auf die Geschäftslogik, während die Infrastruktur automatisch skaliert und verwaltet wird.
- **Microservices:** Kleinere, unabhängige Dienste mit eigenen Datenbanken und Schnittstellen. Dies verbessert Skalierbarkeit und Wartbarkeit, kann jedoch zu komplexeren Kommunikationsmustern führen.
- **Self-contained Systems (SCS):** Unabhängige Systeme mit klar abgegrenztem Funktionsumfang. Sie ermöglichen eine bessere Teamautonomie und reduzieren Abhängigkeiten zwischen Systemkomponenten.

Architektur	Vorteile	Nachteile
Monolith	Einfaches Deployment, wenig Overhead	Schwer skalierbar, komplex bei Wachstum
SCS	Unabhängig deploybar, verständliche Grenzen	Redundanzen, komplexe Kommunikation
Microservices	Sehr skalierbar, technologische Freiheit	Hoher Koordinationsaufwand, komplexe Infrastruktur
Serverless	Schnelle Entwicklung, keine Serverwartung nötig	Vendor Lock-in, Debugging schwierig

Abbildung 10: Vor- / Nachteile Architektur Styles

- **Monolith / Modulith:** Ein großes, zusammenhängendes System, bei dem Modularität auf Codeebene beibehalten werden kann. Während Monolithen oft schwer skalierbar sind, ermöglichen Modulithen eine bessere Strukturierung und Modernisierung ohne vollständige Zerlegung des Systems.

7 Vorlesung 07: Advanced Scrum & SAFe

SAFe! = Scalable Agile

- Sprint aim and user stories
  - Situation
    - Produkt vision
    - Roadmap
    - Release plan
    - Story map
  - Aim
  - Capacity
    - Story Points
    - Estimation
  - Review candidates
    - Target alignment ok?
    - User story ok?
    - Objectives are clear?
- Checklist
  - For every user stroy
    - What has the highest priority
    - What do we need to understand
    - Implementation understandable
    - How to deliver?
    - Effort
    - Everything considered?
  - Backlog verification
    - Conflicts
    - Definition of Done
    - Missing backlog entries
    - Risks
    - Commitment
    - Tooling
- Final (next steps & issues)

7.1 Definition of Done (DoD)

- **Who should define "DONE"**: The whole Team
- **How to define**: The definition of done should be agreed to by the team, written down and rigorously followed by involving the product owner (client) and evolve the definition of done
- **When to define**: Bevore every sprint and adapted sprint by sprint

7.1.1 Definition of Done for implemented User Stories

1. Unit tests pass and coverage met standard (85% or above)
2. Sufficient negative unit tests were written (more negative than positive)

3. Code is reviewed (or Pair programmed)
4. Coding standards are met
5. Continuous integration implemented (automated build, deployment and testing)
6. Code is refactored (to support the new functionality)
7. UAT tests pass (test case requirements)
8. Non-functional tests pass (scalability, reliability, security, etc.)
9. Necessary documentation is completed

7.1.2 Definition of Ready for a User Story

1. User Story defined
2. User Story Acceptance Criteria defined
3. User Story dependencies identified
4. User Story sized by Delivery Team
5. Scrum Team accepts User Experience artefacts
6. Performance criteria identified, where appropriate
7. Person who will accept the User Story is identified
8. Team has a good idea what it will mean to Demo the User Story

CQRS (Circuit Breaker / Bulkhead / Retry) Strangler Pattern / Online-Migration Event Sourcing

8 Vorlesung 08: Nothing important

9 Vorlesung 09

Software Architecture Definition Die Software Architektur eines Programms oder eines Computersystems ist die Struktur oder die Strukturen des Systems, die aus Software-Elementen, den nach aussen sichtbaren Eigenschaften dieser Elemente und den Beziehungen zwischen ihnen bestehen

9.1 What makes building architecture difficult

- Komplexität (Complexity)
- Konformität (Conformity)
- Formbarkeit (Changeability)
- Unsichtbarkeit (Invisibility)

9.2 Architekturtypen

9.2.1 Applikationsarchitektur

Zentrale ist dabei die Fragestellung, wie die Anwendung in einzelne Buildingblocks (Bausteine) zerlegt werden kann:

- Komponenten
- Schichten
- Packages
- Namespace

Organisation des Codes

9.2.2 Systemarchitektur

Bei der Systemarchitektur geht es um das Verständnis des Aufbaus der Software und der Hardware und dessen Zusammenspiels.

9.2.3 Enterprise Architecture

Enterprise Architecture die Art und Weise, wie ein Unternehmen viele Anwendungen nutzt”

9.3 Non Functional Architecture

Quality of software architecture influences all the system properties (characteristics that can be measured during runtime / only indirectly measurable characteristics)

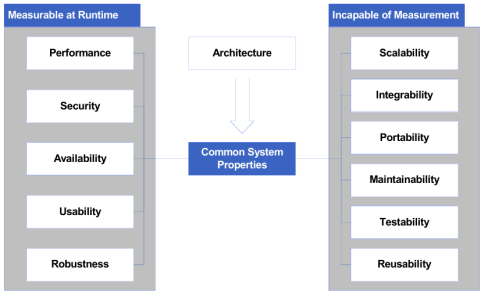

Abbildung 11: Measurable / Indireclty Measurable Characteristics

9.4 Software Architecture Principles

- Modularity: easily interchangable components
- Portability: runs in different env.
- Changeability / Malleability
- Conceptual Integrity: similar parts similar design, use reference / industry standards
- Intellectual Control: detailed understanding for form, content, complexity
- Buildability: (machbarkeit)

10 Vorlesung 10

10.1 Positionierung der Ebenen (engl. layering oder layer positioning)

Softwaresystem in verschiedene Schichten (Ebenen) unterteilt, um Struktur, Verständlichkeit, Wartbarkeit und Flexibilität zu verbessern.

1. Presentation Layer (UI / Client)
   - Zeigt Informationen an, nimmt Eingaben entgegen.
   - Beispiel: HTML-Seite, App-UI.
2. Application / Business Logic Layer / Funktion / Logik
   - Enthält Geschäftsregeln und verarbeitet Benutzeraktionen.
   - Beispiel: „Wenn der Benutzer bestellt, ziehe Lagerbestand ab.“
3. Daten (Datenhaltung / Data Layer)
   - Speicherung, Datenbank-Zugriffe
   - Beispiel: SQL, NoSQL, File-Storage

10.1.1 Mögliche Rollenschemas Client-Server

- **Distributed Presentation:** Die verteilte Präsentation versteckt die Lokalität der einzelnen Clients, während der Server alle Clients zentral (beispielsweise auf einem Bildschirm) darstellt
- **Remote Presentation:** Die entfernte Präsentation bedeutet, dass ein Client alle Präsentationsaufgaben übernimmt, während Anwendung und Daten vom Server verwaltet werden
- **Distributed Function:** Die verteilte Funktionalität setzt eine Arbeitsteilung auf funktionaler Ebene zwischen Client und Server um (Cooperative Processing)
- **Remote Data:** Der Server hält alle Daten an einem Ort, es werden keine Daten auf den Clients gespeichert

- **Distributet Data:** Die Daten werden verteilt auf verschiedenen Servern gehalten, der Client greift gleichzeitig auf verschiede Server zu

10.2 Standard – Stil - Pattern

- **Architektur-Standards:** Auswahl und Anwendung des Architekturstandards
- **Architektur-Stil:** Auswahl und Anwendung der Kombination der verschiedenen Architektur-Stilelemente
- **Pattern**: Auswahl und Anwendung der passenden Entwurfsmuster

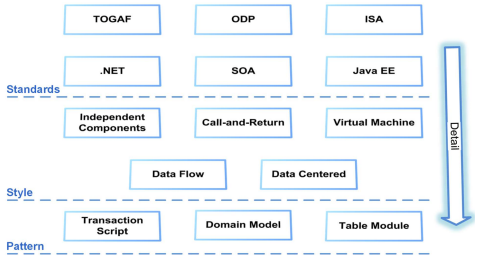

Abbildung 12: Standard – Stil - Pattern

10.3 Kommunikation (Welche Art Diagramme, Notationen, Detailgrad, für Wen?)

- Vermittelt das große Ganze: Erklärt, was gebaut wird und wie es ins Gesamtbild passt.
- Schafft eine gemeinsame Vision: Einheitliches Verständnis im Entwicklungsteam.
- Dient als Anlaufstelle für das Team: Fokus auf Software und deren Entwicklung.
- Technischer Fokuspunkt: Grundlage für Diskussionen über die Umsetzung neuer Features.
- Landkarte für den Quellcode: Erleichtert die Navigation im Code für Entwickler.
- Erklärt das Projekt nach außen: Verständlich für technische und nichttechnische Stakeholder.
- Beschleunigt Onboarding: Neue Entwickler finden sich schneller zurecht.

10.4 Architekturstyle

Architektur Stil	Anwendung	Vorteil	Nachteil
Independent Components			
Communicating Processes	Parallelverarbeitung	Einfache Modellierung, Skalierbarkeit	Komplexität der einzelnen Elemente
Event Systems	GUI's, Real Time Systems	Unabhängigkeit der Elemente, Änderungs-Freundlichkeit	Non-Deterministisches Verhalten der Elemente
Call-and-Return			
Main Program & Subroutine	Structured Programming, Client-Server (RPC)	Definierter Kontrollfluss	Skalierbarkeit, Erweiterbarkeit
Object Oriented	Allgemeines Design, Client-Server	Universell	Komplexität, Anwendungsfreiheit
Layered	SOA, Multi-Tier Architectures	Konzeptionelle Integrität, Lokalität der Änderungen	Performance, Komplexität
Virtual Machine			
Interpreter	Prozessor- und Betriebssystem-Simulation	Portabilität, Flexibilität	Performance
Rule-Based Systems	Expertensysteme	Flexibilität durch Regelwerk	Komplexität, Performance
Data Flow			
Batch Sequential	Host Systeme	Datensteuerung	Flexibilität, Interaktion
Pipes and Filters	Software Converter, Compiler	Flexibilität, Verteilung	Komplexität
Data Centered			
Repository	Stammdatenverwaltungen	Einfach	Single Point of Failure
Blackboard	Datengesteuerte Kontrollsysteme	Skalierbar	Anwendung eingeschränkt

Abbildung 13: Architekturstyle: Vorteile und Nachteile

10.5 Patterns

- Das **Transaction Script Pattern** organisiert und unterteilt die Geschäftslogik in einzelne Prozeduren, sodass jede Prozedur eine einzelne Anfrage vom Presentation Layer
- abdeckt.
- Das **Domain Model Pattern** beschreibt ein Objektmodell der Problemdomäne, welches Verhalten und Daten
- umfasst.

- Das **Table Module Pattern** beschreibt eine einzelne Instanz (Singleton), die die Geschäftslogik für alle Zeilen in einer Datenbanktabelle oder View kapselt.

10.6 C4 Model

1. Context: Systemkontextdiagramm
    - Was ist das Software-System, das gebaut wird?
    - Von wem wird es verwendet?
    - Wie passt es in die bestehende Umgebung?
2. Containers: Container diagram
    - Wie ist die Gesamtform des Softwaresystems?
    - Was sind die High-Level-Technologieentscheidungen?
    - Wie sind die Verantwortlichkeiten im System verteilt?
    - Wie kommunizieren die Container miteinander?
    - Wo muss der Entwickler Code schreiben, um Funktionen zu implementieren?
3. Components: Component diagram
    - Aus welchen Komponenten setzt sich jeder Container zusammen?
    - Haben alle Komponenten ein Zuhause (d. h. sie befinden sich in einem Container)
    - Ist es klar, wie die Software auf hoher Ebene funktioniert?
4. Code

: Code Level Diagrams

10.7 Documentation Fragestellung

- Wie fügt sich das Softwaresystem in die bestehende Systemlandschaft ein
- Warum wurden die verwendeten Technologien gewählt
- Wo werden die verschiedenen Komponenten zur Laufzeit eingesetzt und wie kommunizieren sie
- Welcher Ansatz wurde für Logging / Konfiguration / Fehlerbehandlung / etc. gewählt
- Muster und Prinzipien
- Wie und wo neue Funktionalität hinzugefügen
- Wie die Sicherheit im gesamten Stack implementiert worden ist

11 Vorlesung 11: Ausfall

12 Vorlesung 12: Cynefin Framework

12.1 Begriffe

- **Constructive irritants** see a problem and courageously bring it into the light and offer concrete solutions. (Contrary of a complainer Complainers are toxic and increase the negativity in the environment. They are finger-pointers and focus on everything that isn't working.)
- **Cogent** compelling belief or assent; forcefully convincing
- **Exaptation** The process by which features acquire functions for which they were not originally adapted or selected.

12.2 CAS

- **Acomplex adaptive system** is a system that is complex in that it is a dynamic network of interactions, but the behavior of the ensemble may not be predictable according to the behavior of the components. It is adaptive in that the individual and collective behavior mutate and self-organize corresponding to the change initiating micro-event or collection of events.
- Interactions in a CAS are non-linear: small changes in inputs, physical interactions or stimuli can cause large effects or very significant changes in outputs

- CAS have a history. They evolve and their past is co-responsible for their present behaviour

Abbildung 14: Liminal Cynefin

12.3 Understanding the Cynefin Framework

categorizes problems into five domains:

- **Obvious (formerly Simple)**: Problems with clear cause-and-effect relationships, solvable through best practices. For example, programming a turtle to move in a square.
- **Complicated**: Problems that require expert analysis but have predictable outcomes. An example is developing a standard CRUD form.
- **Complex**: Problems where cause and effect can only be understood in retrospect. Solutions emerge through probing and experimentation.
- **Chaotic**: Situations lacking clear cause-and-effect relationships, requiring immediate action to establish order.
- **Disorder**: When it's unclear which domain applies, leading to confusion and potential missteps.

12.3.1 Application in Software Development

- software development often spans multiple domains, particularly the Complicated and Complex.
- some aspects of development are predictable and benefit from upfront analysis, others are unpredictable and require iterative feedback and adaptation.
- Over-reliance on upfront analysis can bog down development processes, especially when dealing with complex problems that benefit more from feedback and iterative approaches.

12.3.2 Conclusion

By recognizing the domain a problem falls into, teams can balance analysis and feedback, leading to more effective and adaptive software development practices.