

# **Software Architecture and Techniques**

**Zusammenfassung**

Stephan Stofer

14. Mai 2021

# Inhaltsverzeichnis

<b>1</b>	<b>Introduction</b>	<b>6</b>
<b>2</b>	<b>Evolution of Software Architecture over the Last Decades</b>	<b>7</b>
2.1	Architecture Definitions . . . . .	7
2.2	Old School Architect . . . . .	7
2.3	Architecture Kinds . . . . .	8
2.4	Standards ab 2000 . . . . .	8
2.5	First Findings . . . . .	8
2.6	Requirements - SMART . . . . .	8
2.7	Stories - INVEST . . . . .	9
2.8	Backlog - DEEP . . . . .	9
2.9	Backlog Item . . . . .	9
2.10	DDD and Event Storming . . . . .	9
2.10.1	Agile Approach . . . . .	9
2.10.2	Craftsmanship Approach . . . . .	9
<b>3</b>	<b>Agile Architecture</b>	<b>10</b>
3.1	Agile Architecture Prinzipien (SAFe) . . . . .	10
3.2	Agile Architecture . . . . .	10
3.2.1	Pain Points . . . . .	10
3.2.2	Entscheidungen . . . . .	10
3.2.3	Product Backlog . . . . .	12
3.2.4	Quality Tree . . . . .	12
3.2.5	Factors to decide . . . . .	12
3.2.6	Spike . . . . .	15
3.2.7	Wichtige Liste um Architektur zu verbessern . . . . .	15
3.2.8	Requirements Engineering . . . . .	15
3.2.9	Vision . . . . .	15
3.2.10	Minimale Solution . . . . .	18
3.2.11	Architecture Workshop . . . . .	19
3.2.12	Define Evolution Steps . . . . .	20
3.2.13	Architect Skills . . . . .	21
3.2.14	Agile Architectures Pattern . . . . .	21
3.2.15	Identify Bounded Context . . . . .	22
3.2.16	Documentation . . . . .	22
3.3	Agile Architecture Principles . . . . .	24
3.4	Agile Architecture Techniques . . . . .	24
3.5	Scrum Practices . . . . .	24
3.5.1	Scrum Approaches . . . . .	24
3.6	eXtreme Programming Practices . . . . .	24
3.7	Craftsmanship Approach . . . . .	25
3.7.1	Manifesto . . . . .	25
3.8	LeSS Practices . . . . .	25
3.9	SAFe and DAD . . . . .	25

3.10	Refactoring & Clean Code . . . . .	25
3.11	Agile Architecture Approach . . . . .	25
3.11.1	Ubiquitous Language . . . . .	26
3.12	Reality . . . . .	26
3.13	Misconceptions . . . . .	26
3.14	Simplistic Refactoring . . . . .	26
3.15	Mechanical Refactoring . . . . .	27
3.15.1	Typical Mechanical Refactoring . . . . .	27
3.15.2	Advanced Refactoring . . . . .	27
3.16	Refactoring Examples . . . . .	27
3.17	TDD - Test Driven Development . . . . .	28
3.18	Legacy Code Unit Testing . . . . .	28
3.19	The Modern Way of Testing . . . . .	28
3.20	Refactoring Catalog . . . . .	28
3.21	Clean Code . . . . .	29
3.21.1	Tools für CC . . . . .	29
3.21.2	Why Tools . . . . .	29
3.21.3	Ziele . . . . .	30
3.22	Why Pair Programming . . . . .	30
3.22.1	Next Stage - Mob Programming . . . . .	30
3.22.2	Technical Meetings / Dojos . . . . .	30
3.22.3	SonarLint and SonarQube . . . . .	30
3.23	Forensics Approach . . . . .	30
3.24	Advanced Tools . . . . .	30
3.24.1	Module . . . . .	31
3.24.2	ArchUnits . . . . .	31
3.24.3	DevOps Approach . . . . .	31
3.25	Zero Bug Policy . . . . .	31
3.26	Bad API's . . . . .	31
3.27	Bad Scrum / Agile . . . . .	31
3.28	SOLID . . . . .	32
3.29	Patterns . . . . .	32
3.29.1	Builder Pattern Example . . . . .	32
3.30	Layered Architecture . . . . .	32
3.31	Hexagon/Onion Architecture . . . . .	33
3.32	Java Ecosystem . . . . .	33
3.33	Refactor . . . . .	33
3.33.1	OOP Anti-Patterns . . . . .	33
4	Quality Attributes of Software Architecture	34
4.1	Characteristics . . . . .	34
4.2	Functional Requirements . . . . .	34
4.3	Stories as Functional Requirements . . . . .	34
4.3.1	Stories . . . . .	34
4.3.2	Use Cases . . . . .	34
4.3.3	Validation . . . . .	35
4.3.4	Testing Quadrants . . . . .	35
4.3.5	Testing Pyramid . . . . .	35
4.4	Architecture Goals . . . . .	35
4.5	Quality Attributes . . . . .	35
4.6	How to reach these Goals? . . . . .	36
4.7	Quality Citations . . . . .	36

4.8	Refelction . . . . .	36
<b>5</b>	<b>-ility-Attributes</b>	<b>37</b>
5.1	Truths . . . . .	37
5.2	Non-Functional Requirements . . . . .	37
5.3	Fitness Functions . . . . .	38
5.4	Assumptions . . . . .	38
5.4.1	Combining Fitness Functions . . . . .	38
5.4.2	Functions Examples . . . . .	39
5.5	Code Quality . . . . .	39
5.6	Resilience and Operability . . . . .	39
5.7	Performance and Security . . . . .	39
5.8	Micro Profile . . . . .	39
<b>6</b>	<b>Architecture Documentation</b>	<b>40</b>
6.1	Truths . . . . .	40
6.2	Why Should You Document . . . . .	40
6.2.1	Domain Driven Models . . . . .	40
6.2.2	UML for small Models . . . . .	40
6.3	Architectural Design Record . . . . .	41
6.4	Rules for Documentation . . . . .	41
6.5	Acceptance Tests . . . . .	41
6.6	Fitness Functions . . . . .	41
6.7	API Documentation . . . . .	41
6.8	Configuration as Code . . . . .	42
6.9	Static Web Sites . . . . .	42
<b>7</b>	<b>Architectural Trends I</b>	<b>43</b>
7.1	Truths . . . . .	43
7.2	Domain Driven Design . . . . .	43
7.3	Hexagonal or Onion Architecture . . . . .	43
7.4	Domain Driven Development . . . . .	44
7.5	Domain Ubiquitous Language . . . . .	44
7.6	Domain Driven Design . . . . .	44
7.7	Domain Events . . . . .	44
7.8	Multiple Teams . . . . .	44
7.9	Immutability and Functional Style . . . . .	45
7.10	Questions . . . . .	45
7.10.1	DDD Anti-Patterns . . . . .	45
7.11	Event Storming - Ignite your DDD . . . . .	45
7.12	Micro Services . . . . .	46
7.13	Ball of Mud . . . . .	46
7.14	Monolith to Modular . . . . .	46
7.15	Refactor . . . . .	46
7.16	Evolvable Architecture . . . . .	46
7.17	Wisdoms . . . . .	47

# Abbildungsverzeichnis

3.1	Agile Manifesto . . . . .	11
3.2	Pain Points . . . . .	11
3.3	Entscheidungen . . . . .	12
3.4	Backlog . . . . .	13
3.5	Quality Tree . . . . .	14
3.6	Spike . . . . .	15
3.7	Checkliste / wichtige Fragen zur Architektur . . . . .	16
3.8	Requirements Engineering . . . . .	16
3.9	Vision . . . . .	17
3.10	Identify Bounded Context . . . . .	23
3.11	The Modern Way of Testing . . . . .	28
4.1	Testing Quadrants . . . . .	35
5.1	Fett markiert die wichtigsten ility-Attribute . . . . .	38
7.1	Hexagonal or Onion Architecture . . . . .	43
7.2	Domain Driven Design . . . . .	44
7.3	Event Storming . . . . .	45

# 1 Introduction

To be written

## 2 Evolution of Software Architecture over the Last Decades

Ansatz Unterschied Design und Architektur.

Architektur ist Design, aber nicht jedes Design ist Architektur. Architektur repräsentierten wichtige Designentscheide. Wie wichtig der Entscheid ist, ist messbar an den Konsequenzen bzw. Kosten etwas zu ändern.

– Grady Booch

Anders als bei mechanischen Produkten

Software skaliert nicht ökonomisch.

– Allan Kelly

Wenn das System wächst, nimmt Komplexität nicht linear sondern annähernd exponentiell zu. Mit guter Architektur kann man entgegenwirken.

### 2.1 Architecture Definitions

Zwei grosse Standards von ISO und TOGAF. Beide definieren Architektur schwammig.

### 2.2 Old School Architect

- Hoher Status, separate Position
- Entscheidet alleine wie die Architektur sein soll
  - Architekturen sind smart
  - Developers sind dumm
- Ivory tower syndrome - theoretisch, in Praxis nicht umsetzbar
- Powerpoint architect syndrome - nur Schrott im Powerpoint
- Conway Law - Hierarchie und Struktur der Teams soll dem SW Produkt angepasst werden. Damit das beste herauskommt. Da arbeiten wo man am besten arbeiten kann

## 2.3 Architecture Kinds

Alle traditionelle Architekten wollen einen enterprise architect sein.

- Design → developer
- Application Architecture in einem Team
- Solution A mit einem Produkt
- Enterprise A in der ganzen Firma

Auch Technische Architektur

- Business Architecture
- Application Architecture
- Data Architecture
- Technical Architecture

## 2.4 Standards ab 2000

- TOGAF; gute Ideen aber zu gross, man braucht zu viel Zeit
- Arc42; häufig in Deutschland
- RUP; A la SoDa zu lange Zyklen
- Hermes; offizielle Methode vom Bund auch häufig in Kantone
- IEEE; angestaubt

Bei allen gibt es gute Ideen, aber falsch verpackt (zu wenig agil). Gute Dinge rausnehmen.

## 2.5 First Findings

Ziel von Architekt Kundenbedürfnisse erfüllen. Es gibt funktionale und nicht-funktionale Anforderungen. Requirementsdokumente werden widersprüchlich je grösser sie werden. Besser agil, etwas analysieren, umsetzen, testen, verbessern.

- Problem verstehen
- Lösung identifizieren
- moderne Verfahren lösen alte ab (riesen Dokumente)

## 2.6 Requirements - SMART

- **S**pecific
- **M**easurable - acceptance criteria
- **A**ssignable (who will die it?)
- **R**ealistic - within a sprint
- **T**ime-related/Traeable (when should it be done?)/ Akzeptanz Test

Jedes mal SMART in Erinnerung rufen bei Anforderungen.

## 2.7 Stories - INVEST

- Independent - Stories sollten nicht von anderen abhängig sind
- Negotiable - kein Widerspruch sein
- Valuable - wenn kein Wert, dann verwerfen
- Estimate-able - muss verstanden sein, damit man sie schätzen kann
- Small - kleine Stücke an Arbeit
- Testable - müssen Testbar sein damit man sie abschliessen kann

## 2.8 Backlog - DEEP

- **D** Detailed Appropriately
- **E** Estimated
- **E** Emergent
- **P** Prioritized

## 2.9 Backlog Item

Ein Procut Backlog Item PBI ist ein edle ToDo-Liste. Keine Referenzen von Dokument auf Backlog. Ein PBI kann eine Story sein. Eine Schätzung lohnt sich nur wenn man über die Umsetzung diskutiert, wenn man es eh umsetzen muss, ist eine Schätzung nur verschwendete Zeit. Ein PBI ist fertig wenn technische und funktionale Anforderungen (User) erfüllt sind.

## 2.10 DDD and Event Storming

Es ist wichtig, dass man Kundensprache (Domäne) spricht und man benötigt Fachdomänenwissen. Als Architekt muss man Workshop und Diskussionen führen. UX Workshops leiten und Design Thinking.

### 2.10.1 Agile Approach

Vision, wieso will man das? Elevator Pitch. Wohin man in den nächsten 9-18 Monate (Roadmap). Release Planung mit Story Map. Sprint Backlog - Was macht man in den 1-2 Woche. MVP - minimum viable Product. MMP Minimum Marketable Product (nicht vergessen)

- effizienteste und effektivste Methode ist face to face Kommunikation
- die beste Architektur entsteht mit der Zeit von selbst-organisierten Teams

### 2.10.2 Craftsmanship Approach

Ein Architekt ist ein

- Domänenexperte
- Software Craftsmanship
- Lean Leader, entwickelt sein Team - Mentor
- muss auf versch. Levels/Ebenen diskutieren können (Chef-level, Stakeholder, usw.)

# 3 Agile Architecture

Jeder SW Developer ist ein Designer *und* Architekt. Die Summe der gesamten Source Codes ist das wahre Design und Architektur. Die Architektur evolviert jeden Tag, wird besser oder schlechter, wenn Leute programmieren. Master Programmierer beeinflussen die Architektur. Wenn ein Architekt nicht programmiert ist man weg vom Fenster. Jeder Developer ist ein Architekt, jeder sollte entsprechend ausgebildet werden.

## 3.1 Agile Architecture Prinzipien (SAFe)

1. Design entwickelt sich, Architektur ist Zusammenarbeit
2. Je grösser das System, je länger der Weg
3. Bilde die einfachste Architektur die geeignet ist
4. Wenn man Zweifel hat, coden oder modellieren
5. man entwickelt, testet und betreiben es
6. es gibt kein Monopol auf Innovation - jede Person hat gute Ideen
7. Architekturfluss implementieren

## 3.2 Agile Architecture

Die Struktur (Anzahl Teams) der Firma wird sich in der Architektur widerspiegeln (z.B. mehr Teams mehr Services).

1. Impact of Agile on Architecture
2. Decisions at the right time
3. Evolutionary Design - SW läuft oft Jahrzehnte -> auch die Daten!!
4. Architect as Coach
5. Patterns for Agility
6. Documentation

### 3.2.1 Pain Points

Wie schnell kann man reagieren? Schnelle Änderungen/Deploys ermöglichen. Müssen einfach verifizierbar sein. Jederzeit Lauffähig

### 3.2.2 Entscheidungen

- Team: junges Team gibt andere Architektur als mit alten Hasen
- RoI: früh auf den Markt, auch wenn noch nicht alle Features vorhanden - Mut zur Lücke!
- Defer with abstractions: nach hinten schieben in der Timeline, erst einführen wenn wirklich nötig
- keine Sackgassen wählen (Kulturen, Programmiersprachen, wenig Fachleute in Region, usw.)

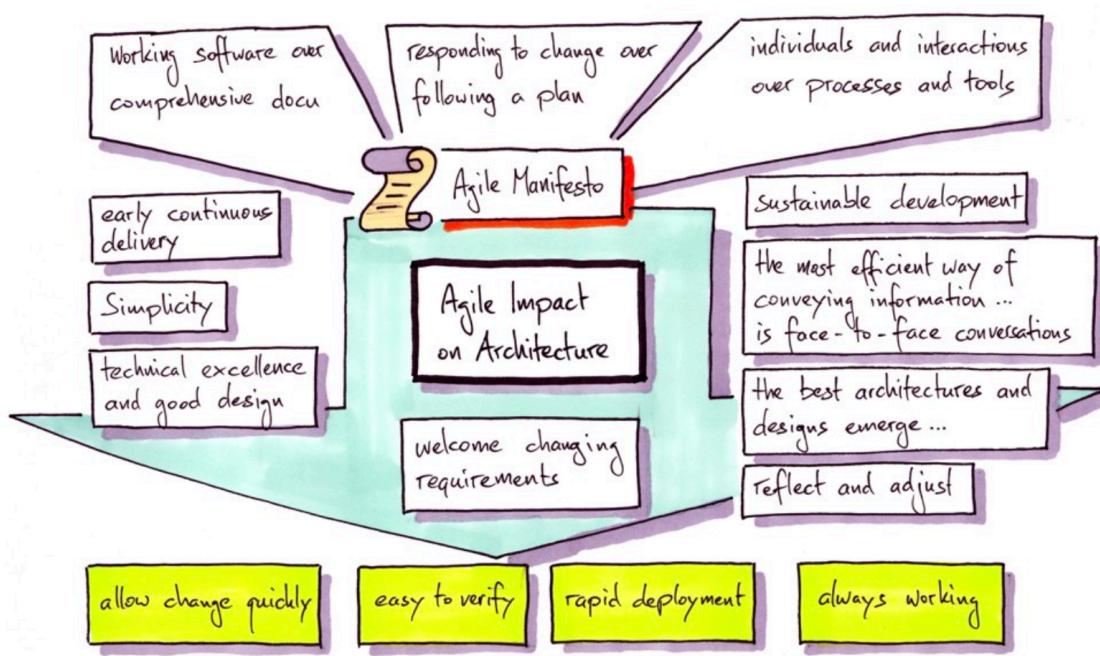


Abbildung 3.1: Agile Manifesto



Abbildung 3.2: Pain Points

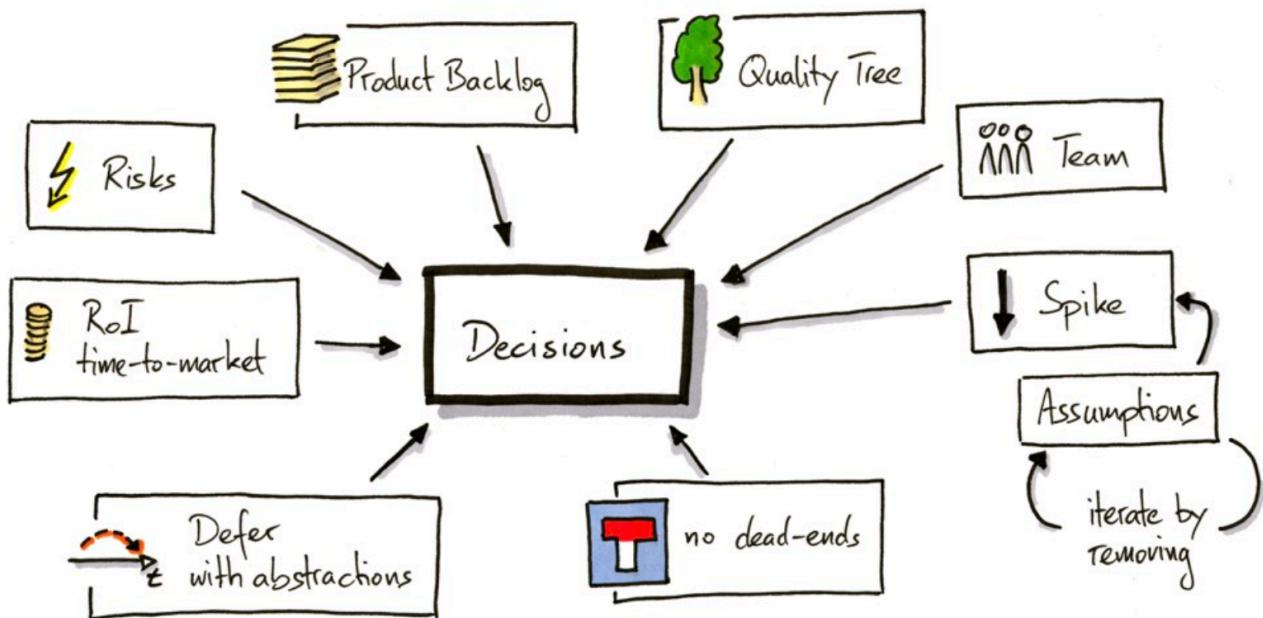


Abbildung 3.3: Entscheidungen

### 3.2.3 Product Backlog

Man muss Entscheide identifizieren und treffen. Je nach Wünsche von Kunden und PO, daraus entwickelt sich Architektur. Architekt entscheidet anhand Backlog.

### 3.2.4 Quality Tree

Identifiziert Tradeoff was ist möglich, was will man (Strategie). Entscheide dokumentieren

### 3.2.5 Factors to decide

Entscheide abhängig von Fähigkeiten von Team treffen. Keine Sackgassen wählen

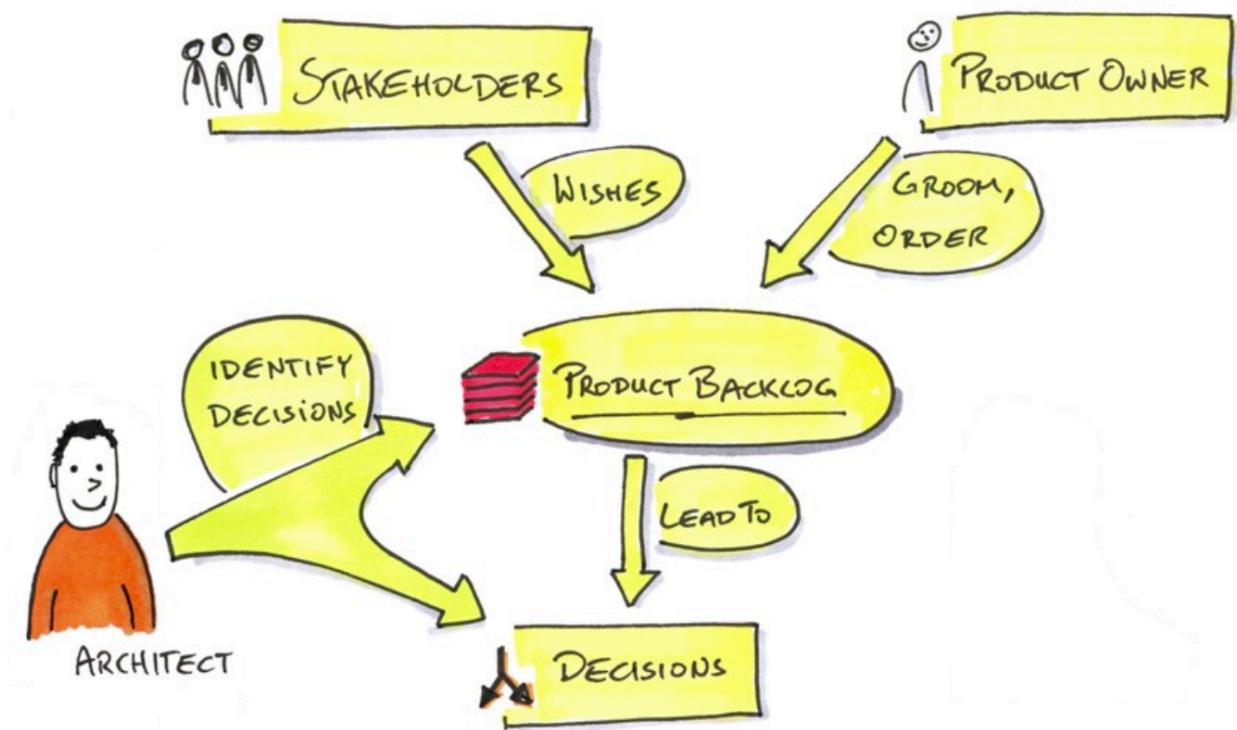
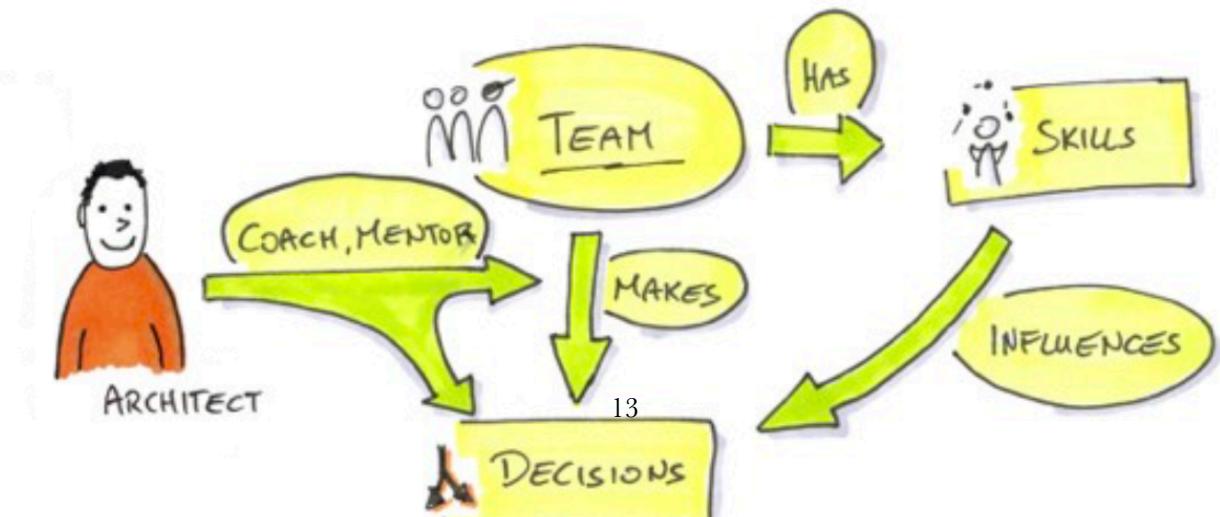
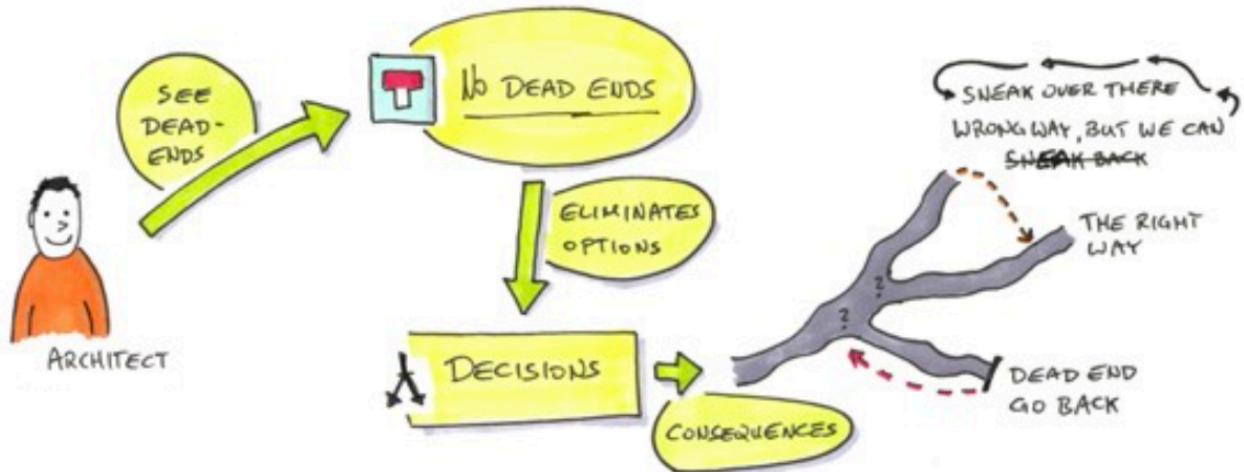


Abbildung 3.4: Backlog



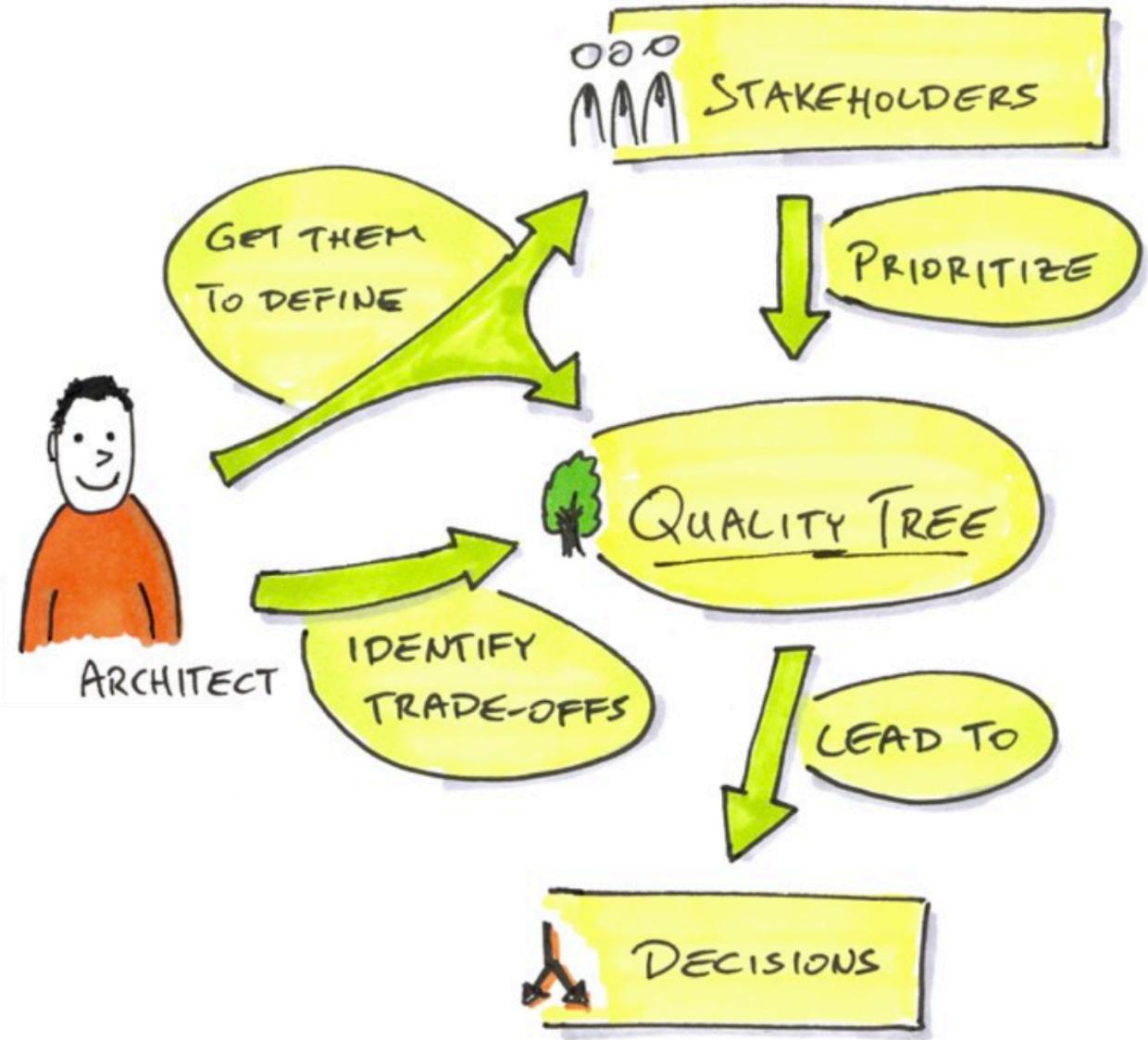


Abbildung 3.5: Quality Tree

### 3.2.6 Spike

Was will ich erreichen?

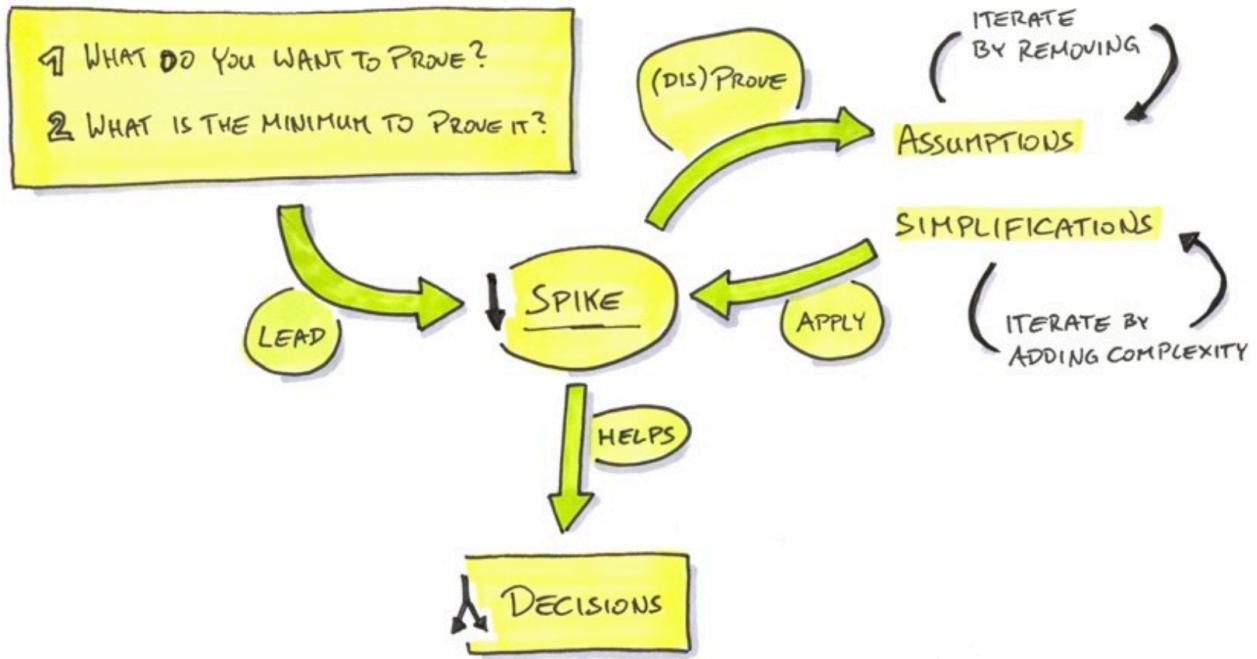


Abbildung 3.6: Spike

### 3.2.7 Wichtige Liste um Architektur zu verbessern

Keine Garantie für Vollständigkeit

2. Characterset, Mehrsprachigkeit
3. Journaling, rechtliche Aspekte absichern (logs)
4. alte Daten auch mit neuem System/Version lesbar?

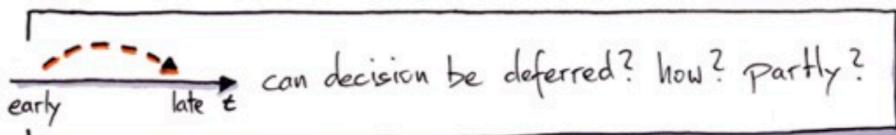
### 3.2.8 Requirements Engineering

Kriterien für Architektur ergeben sich aus Anforderungen, System muss aufgeteilt werden.

Bounding Box definieren, für jede beste Lösung suchen

### 3.2.9 Vision

Entscheide machen die in diese Richtung gehen. Dead-End-Entscheide vermeiden. Lernen, entscheiden, umsetzen. Plan B in Schublade haben



- persist data of your system to survive restart
- how to translate UI and data
- communication between parts of your system
- scaling (run on multiple threads, processes, machines)
- security (how to authenticate, authorize)
- journaling (Activities, data)
- reporting
- data migration / data import
- Releasability
- backwards compatibility
- response times
- Archiving data

design to be independent  
on decision

Abbildung 3.7: Checkliste / wichtige Fragen zur Architektur

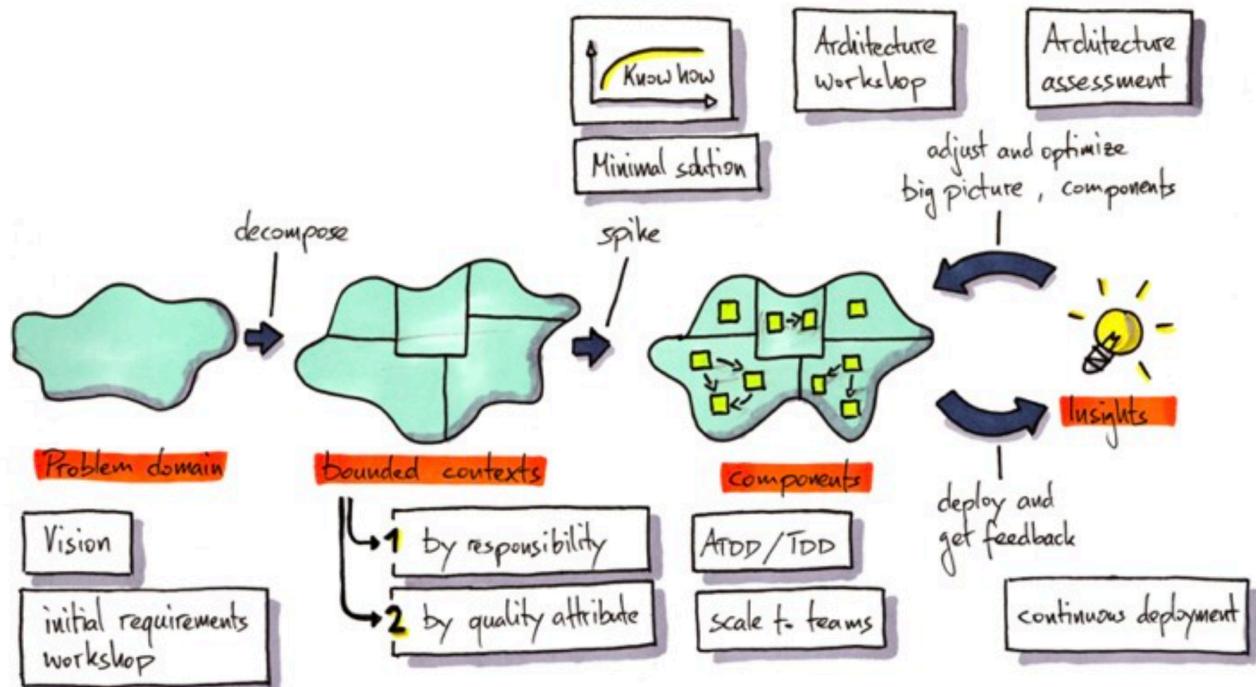


Abbildung 3.8: Requirements Engineering

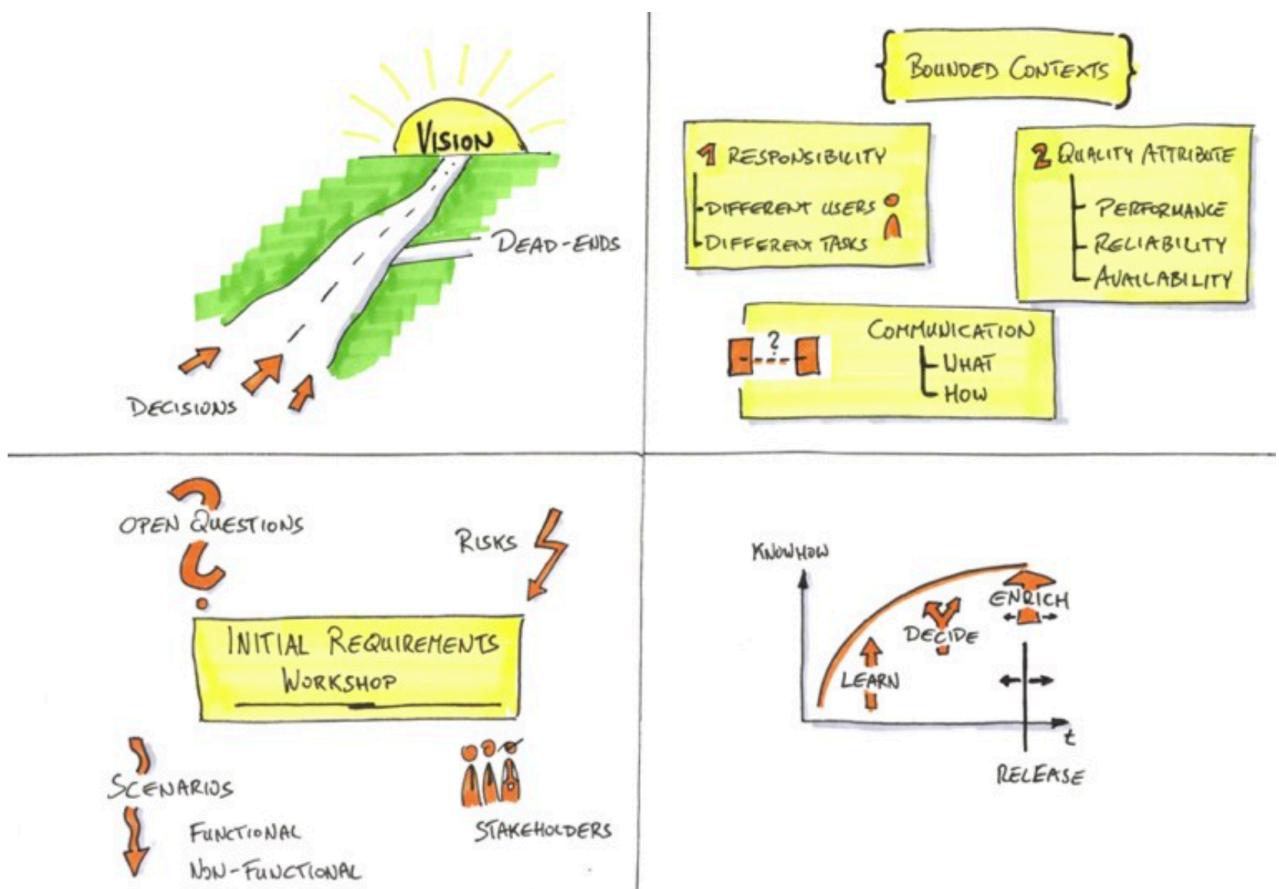
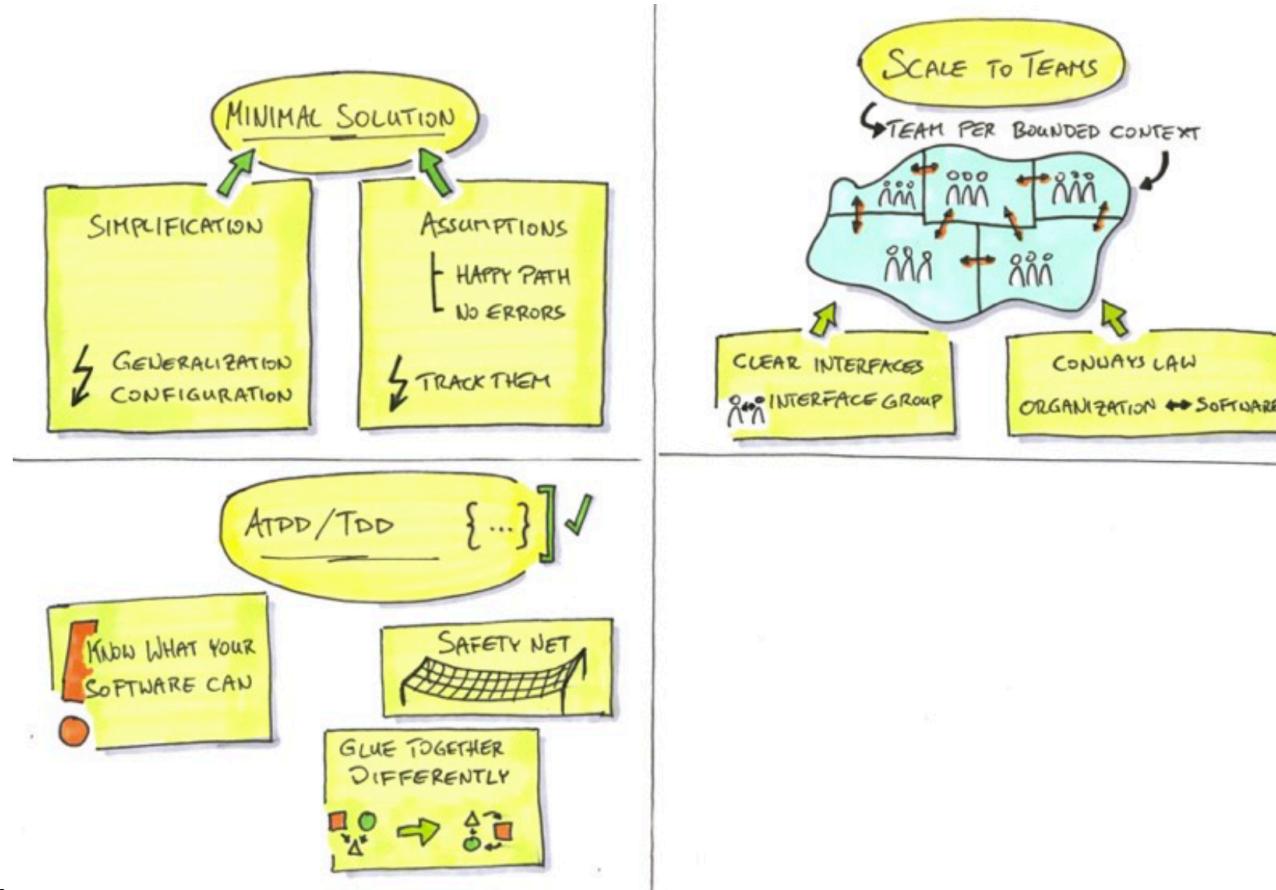


Abbildung 3.9: Vision

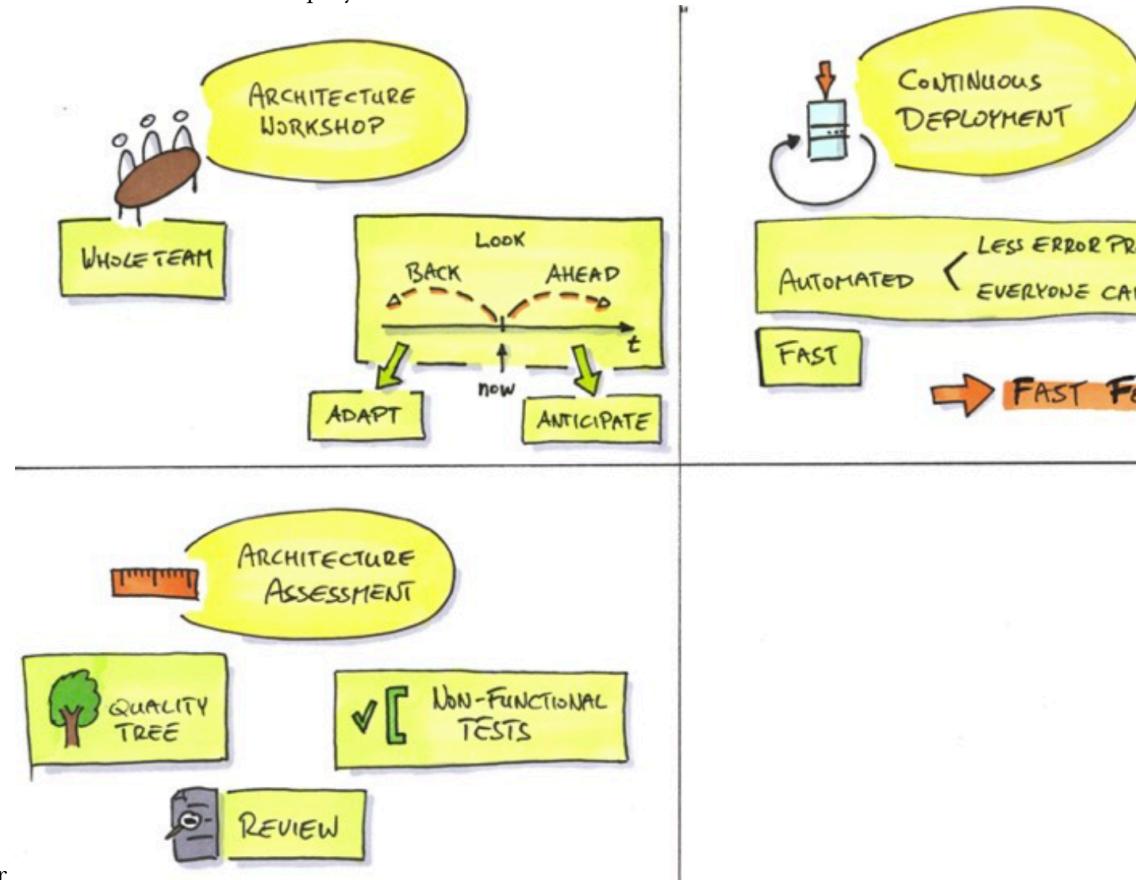
### 3.2.10 Minimale Solution



klein anfangen

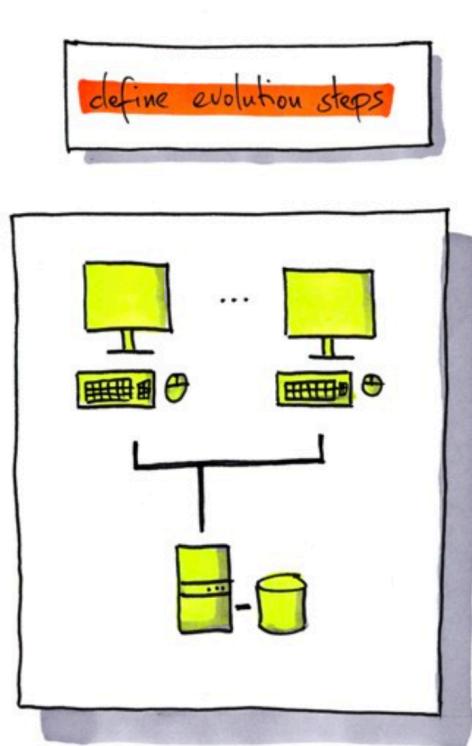
### 3.2.11 Architecture Workshop

Regelmässige Workshops machen Automatisches Deployment um schnelles Feedback zu erhalten Review wenn



etwas nicht autom. Testbar

### 3.2.12 Define Evolution Steps



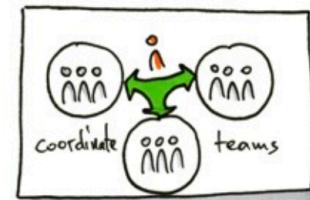
Soll in einer Iteration umsetzbar sein.

Sirius Cybernetics  
Druid Resources Dep

Step I manage data of all  
• assembly  
• retirement  
• serial number  
• jobs  
- work queue  
- from/  
- custom

Step II journaling

### 3.2.13 Architect Skills



lead technically



technology evangelist  
engineering practises  
technical spikes  
non-functional specs  
write code

understa



big pic  
talk to  
learn a  
understa  
help the

communicate

use effec



communicate

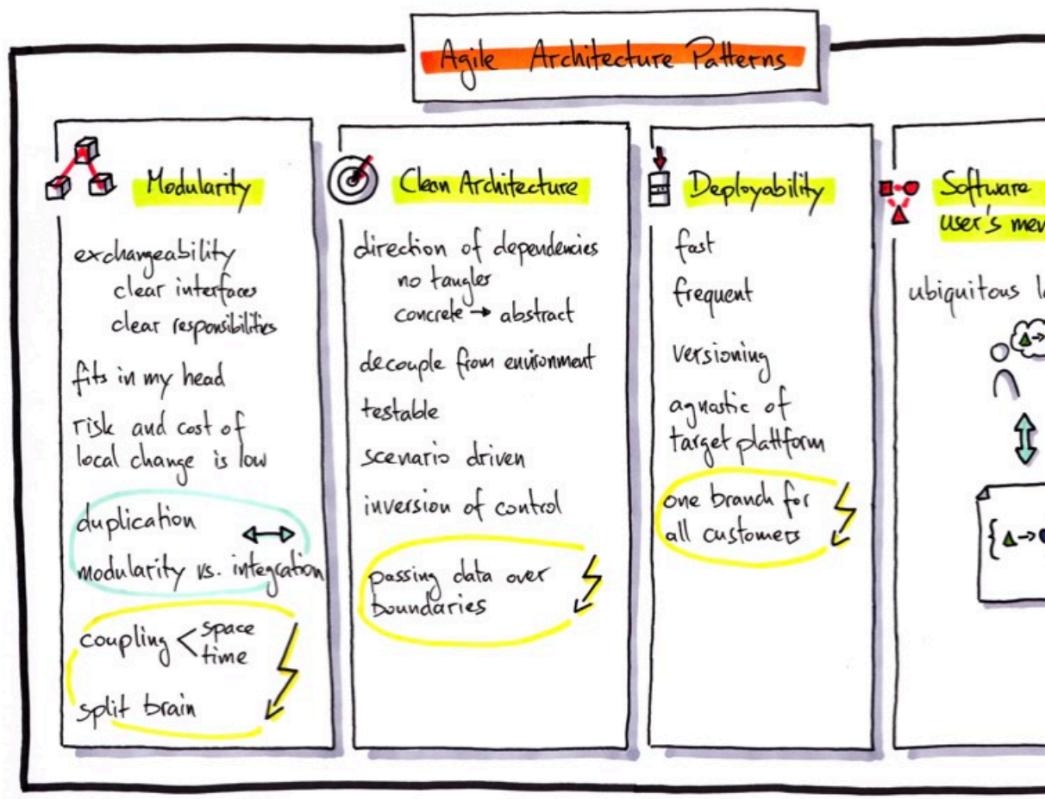
use effec

learn to be a good designer

Skills, Aufgaben, Herausforderungen und Voraussetzungen eines Architekts

### 3.2.14 Agile Architectures Pattern

- Ideen wie SOLID, KISS

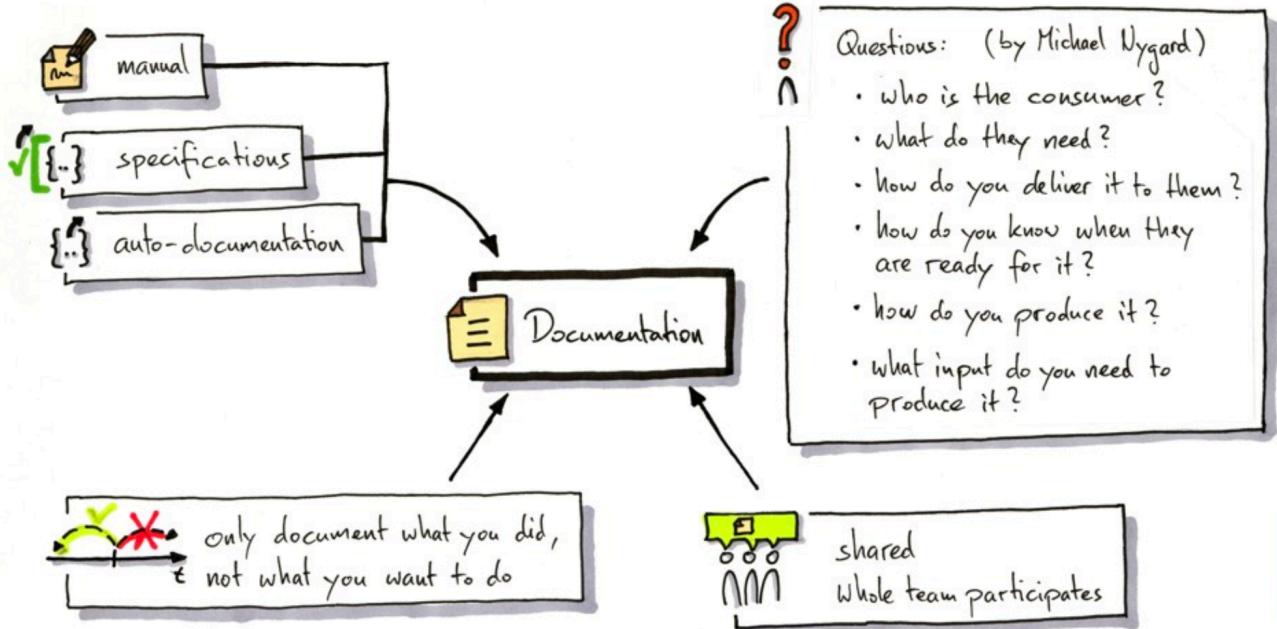


- Austauschbarkeit vorsehen

### 3.2.15 Identify Bounded Context

### 3.2.16 Documentation

Spezifikation ist wie das System gebaut ist ?? //todo: check in lecturer



### # Agile Approaches

Architekt ist eine **Rolle** im agilen Umwelt und nicht ein Titel oder Position. Dabei ist man ein

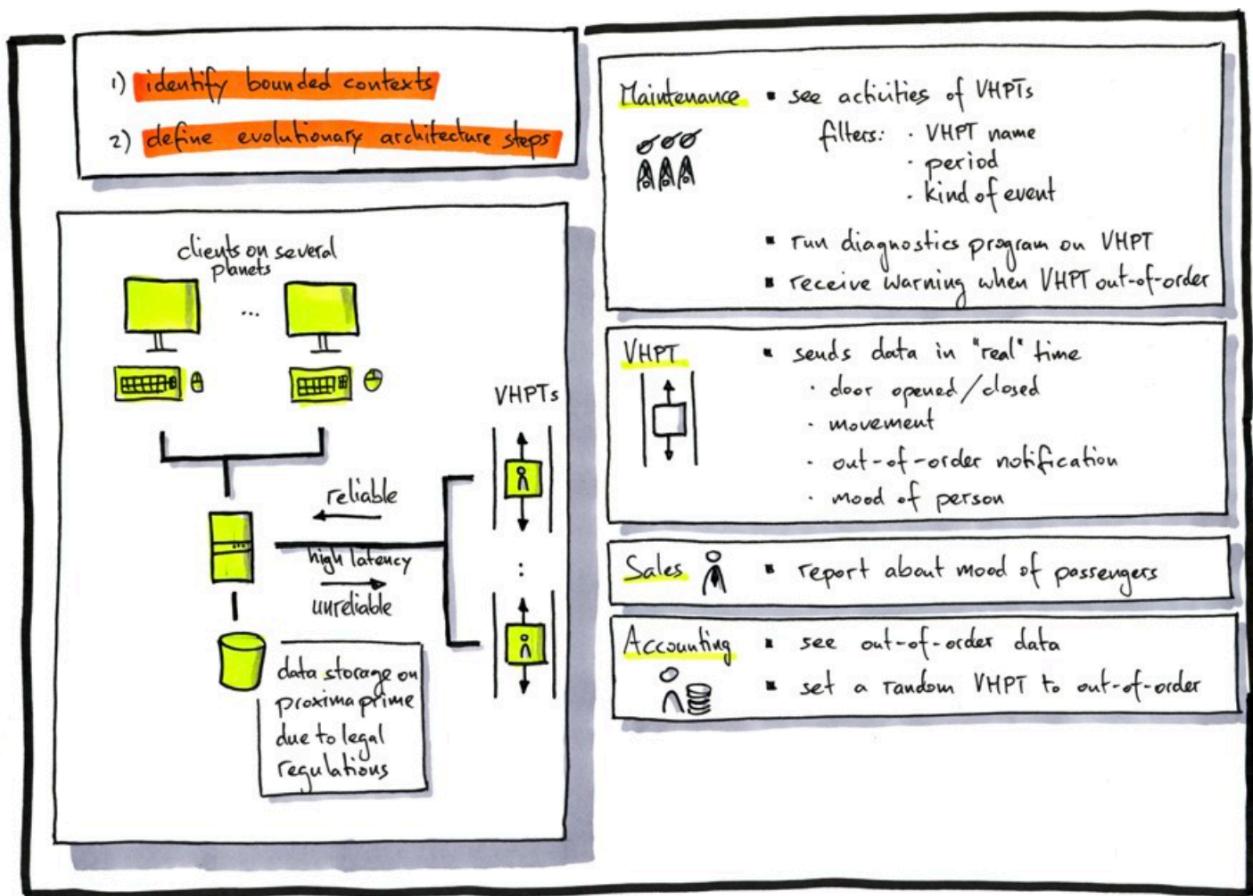


Abbildung 3.10: Identify Bounded Context

- Domäneexperte
- Technologieexperte
- Vermittler zwischen Stakeholder
- Coach, Mentor, Lehrer

### 3.3 Agile Architecture Principles

- Simples Design
- Architektur entsteht durch die Zeit
- Runaway Architecture - man muss voraus sein, mehr liefern - vor dem ersten Prototyp, muss Architektur stehen
- Hexagon approach - Domainspezifische Architektur
- unaufhörliches Refactoring

Gute Architektur ist proportional zu der Fläche des Whiteboards.

### 3.4 Agile Architecture Techniques

- CI/CD/CD - je schneller Feedback, umso besser kann man reagieren
- Agile implies Automation
- Git impact - GitOps, GitFlow
- Potentionell *Shippable* Produkt - feature toggle (ausrollen auf kleine Pilotgruppe)
- Clean Architektur, Code, Coder

## 3.5 Scrum Practices

Scrum sagt nichts über die technische Umsetzung. Scrum schwergewicht ist Vision (warum), Kontext (wer benutzt) und Roadmap (was wird wann geliefert). Es geht alles um Wert (Business Value) - *outcome over output*. Scrumm pushen eXtreme Programming Techniken. Scrumm-Allianz arbeitet mit *LeSS*.

### 3.5.1 Scrum Approaches

Das Team lernt *zusammen* und entwickelt kontinuierliche Verbesserung.

- Retrospektive
- Review
- Daily Meeting
- Immer

## 3.6 eXtreme Programming Practices

- Pair Programming
- Test Driven Development
- CI
- Refactoring
- Coding Standards
- Collective Code Ownership

- Simple Design
- System Metaphor

XP fördert individuelles Lernen. Mit motivierten Individuen ein Projekt umsetzen.

## 3.7 Craftsmanship Approach

- Architect is a domain expert
- Architect is a software craftsmanship
- Architect is a lean leader – teacher, coach, mentor
- Architect discuss with stakeholders and C-level representatives

### 3.7.1 Manifesto

- Ich will nicht nur funktionierende Software, aber auch *saubere* Software
- Nicht nur auf Änderungen reagieren, sondern stetig Wert hinzufügen
- Nicht nur Individuen und Interaktion, sondern eine gemeinschaft von Profis
- nicht nur Kundenzusammenarbeit, sondern produktive Partnerschaft

## 3.8 LeSS Practices

LeSS nutzt Scrum als Fundament, alles was gemacht wird ist Scrum. Wenn Team zu gross wird, Topleute einstellen und Team damit kleiner machen. LeSS hat keine Ahnung wie man etwas umzusetzen hat, gibt aber Ratschläge die man anwenden oder verwerfen kann. Vermarkted Architektur als Gärtner. Muss dauerhaft gepflegt werden (nicht nur einmal gebaut wie ein Haus).

## 3.9 SAFe and DAD

Architekturrolle ist immer noch klassisch. Immerhin entwickelt es sich.

## 3.10 Refactoring & Clean Code

Jeder Trottel kann Code schreiben welcher ein Computer verstehen kann, gute Programmierer können auch Menschen verstehen.

## 3.11 Agile Architecture Approach

*Domain Driven Design* und Architektur mit Bounded Domains und Event Storming. Software craftsmanship und clean code schreiben. Architektur ist nicht Technologie neutral. Technologiestack sehr klein halten. Google mit 40000 MA hat 8 Technologien.

### 3.11.1 Ubiquitous Language

Universelle Sprache verwenden. Mix zwischen DomänenSprache und Technischer Sprache. # Refactoring

Refactoring ist eine disziplinierte Technik um bestehenden Code intern umstrukturieren, ohne das externe Verhalten zu verändern. Meist sind es kleine Veränderungen die aber eine signifikante Verbesserung der Les-, Wartbarkeit oder ähnlichem mitsichbringt. Weil die veränderte Einheit auch sehr klein ist, ist die Wahrscheinlichkeit das etwas schief geht sehr klein. Das System wird so verändert, dass es trotzdem immer noch voll funktionsfähig ist.

Ziel ist es den Source Code mit kleinen Schritten zu verbessern. Kleine Änderung - kleine Auswirkung, aber stetiges Improvement!

Durch das kontinuierliche Verbessern des Codes wird immer einfacher um damit zu arbeiten. Dadurch ist er einfacher wartbarer und erweiterbar.

- Wieso: Les- und Wartbarkeit des SourceCodes verbessern
- Wer: jeder Developer macht es
- Wann: jederzeit
- Wo: Jeder Code der modifiziert oder geschrieben wird (nicht irgendwo)

Vor neuen Feature, erst Refactoring machen. Dann mit neuem Feature starten

### 3.12 Reality

Refactoring gibt es seit Jahrzehnten. Obwohl viele Rezepte vorhanden, wird es oft nicht konsequent gemacht. Wieso? - unverständlich, unprofessionell

Es hat früher oder später Konsequenzen wenn man es unterlässt.

### 3.13 Misconceptions

Man muss nicht fragen müssen und man braucht dazu keine Backlog Item. Es ist keine spezielle Aufgabe welche im Projektplan auftaucht. Gut gemacht, ist es eine reguläre Aufwand der Programmierung.

Es ist ein Mindset, will ich Qualität liefern oder möchte ich reparieren.

### 3.14 Simplistic Refactoring

- Code wird nach Coding Guidelines formattiert
- Gute Qualität von Methoden- und Variablennamen
- Import/Using directives sind aktuell
- keine auskommentierter Code -> dazu gibts Git
- keine TODO, FIXME's
- keine leeren Methoden
- keine leeren Catch-Blöcke

## 3.15 Mechanical Refactoring

Automatische Verbesserungen durch die IDE. Kein Risiko um den Code zu brechen. Immer machen, wenn eingeblendet. Jede kleine Änderung committen. IDE sollte Hinweise liefern, was verbessert werden soll.

- renaming über das ganze Projekt
- Verschieben on anderes Package
- Parameterliste ändern, Name oder Typ
- Methoden, Klassen extrahieren

### 3.15.1 Typical Mechanical Refactoring

- Umbenennen
- Verschieben
- Extrahieren
- Inline
- Signatur ändern
- Delegate
- überflüssige Keywords entfernen
- erweiterte Loops verwenden
- Streams verwenden
- Methodenreferenzen verwenden
- Try with resources verwenden
- multiple exception in catch
- unchecked exceptions verwenden
- usw.

Typisch heisst nicht trivial. Etwas schwieriges sind Namen. Er sollte beschreibend sein.

### 3.15.2 Advanced Refactoring

Code so verändern, damit er verbessert wird. Dazu braucht man die Garantie, dass Code genau gleich funktioniert. TDD und ATDD (acceptance TDD) sind ein muss. Dazu braucht es min. 60% Test-Coverage für eine angemessene Sicherheit.

## 3.16 Refactoring Examples

eher bei Java

- nie public fields
- keine Parameter als lokale Variablen nutzen
- nie null retournieren, sonder Empty List oder optional
- Standard Bibliotheksklassen und Exceptions nutzen
- Private Prädikatmethoden anstatt komplexen if-conditions nutzen -> extrahieren
- Loops mit Streams ersetzen
- Conditions mit filter ersetzen
- Immutable Klassen bevorzugen
- keine checked exceptions

## 3.17 TDD - Test Driven Development

1. Test schreiben
2. code schreiben
3. testen
4. fail
5. korrigieren
6. pass

## 3.18 Legacy Code Unit Testing

Jeder bestehender Klasse kann ein Unit Test hinzugefügt werden:

1. Kontext definieren/bauen wie Methode aufgerufen werden kann
2. Letztes Statement (Condition) (ganz rechts) in der Methode testen, Parameter daraus setzen und testen
3. Test mit zweiter Condition von rechts erweitern
4. dann ein Step hoch

## 3.19 The Modern Way of Testing

Shift left for fast feedback

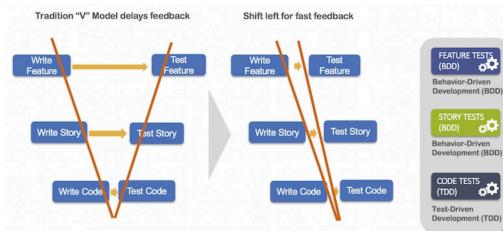


Abbildung 3.11: The Modern Way of Testing

## 3.20 Refactoring Catalog

Online Refactoring by Martin Fowler

Es ist eine Sünde Refactoring-Gegenheiten verstreichen zu lassen, vor dem Pushen der Changes  
– Marcel Baumann # Errors, Vulnerabilities, Smells

Code muss stetig bearbeitet werden, um die Qualität zu halten.

## 3.21 Clean Code

Tools nutzen um Source Code verbessern. Werkzeuge sind günstig, schnell und benötigen keine grosse Koordination. Tools finden aber nur **Fehler**, Qualität können es nicht beurteilen. Es ist ein Mindset, jeden Tag systematisch sich verbessern.

- Compiler Fehler müssen behoben werden
- Compiler Warning auch, es gibt wenige Ausnahmen
- Statische Fehler
  - Bugs, hohe Wahrscheinlichkeit für Crash (Tool Spotbug)
  - Errors, es kann Crash
  - Vulnerabilities, Hackbar
  - Smells, Wartung wird teuerer

### 3.21.1 Tools für CC

- Analyzer in IDE
- Jacoco
- SpotBugs
- SonarLint and SonarQube
- Checksyle (Historisch)
- PMD (Historisch)

#### 3.21.1.1 Sonar Rules

Enthält mehr als 500 Regeln, Enthält Subset von OWASP Vulnerabilities, defacto Standard.

Das Ziel Qualität des Produktes steigern und somit auch Source Code.

#### 3.21.1.2 OWASP

- Injection
- Broken Authentication
- Sensitive Data Exposure
- XML External Entities XEE
- Broken Access Control
- Security Misconfiguration
- Cross Site Scripting XSS
- Insecure Deserialization
- Using Components with known vulnerabilities
- Insufficient Logging and Monitoring

### 3.21.2 Why Tools

- sind günstiger als ein Review durch Personen
- man kann es jede wenige Minuten machen
- Niemand schaut über die Schulter
- finden nur einfache Probleme

### 3.21.3 Ziele

- keine Compiler Errors
- keine Compiler Warnung
- keine Sonar, Spotbugs errors, Vulnerabilities or smells
- Code Coverage sollte höher als 60% sein
- **Jeder gefundene Bug erhält ein Test, welcher den Fehler reproduziert, bevor der Fehler behoben wird**

## 3.22 Why Pair Programming

Werkzeuge finden nur einfache sematische Probleme. Andere Menschen helfen das Design und Architektur zu verbessern. Mehrere Personen kennen den Code.

**Wisdom of the crowd** Pair Programming, Mob Programming

### 3.22.1 Next Stage - Mob Programming

Ganzes Team arbeitet zusammen um ein Problem zu lösen. Kompromiss zwischen Kosten und Zykluszeit.

**Wisdom of the crowd**

### 3.22.2 Technical Meetings / Dojos

Weil Tools Qualität nicht bzw. nur teilweise steigert. Coding Dojos, oder Architectur-Workshop abhalten. Zum Beispiel nach jedem Sprint.

### 3.22.3 SonarLint and SonarQube

- Die beiden Tools machen ein Tool
- Analyse mit SonarLint ausführen
- Generierten Report lesen
- Regelbeschreibung studieren
- wiederholen

## 3.23 Forensics Approach

- Codescene
- `git log --pretty=format: --name-only | sort | uniq -c | sort -rg | head -10`
- `git log --numstat --pretty=format:'[%h] %an %ad %s' --date=short`

## 3.24 Advanced Tools

- Modul-Konzepte nutzen
- ArchUnit - Dependencyregel (Naming, usw.) als UnitTests

### **3.24.1 Module**

Hat grossen Einfluss auf Architektur. Formalisiert Boundend Domains. Compiler Validation. Für kleinere Software Module in Monolith aufbauen.

### **3.24.2 ArchUnits**

Guter Ansatz bevor Module. Benutzerdefinierte Regeln für spezifische Nutzen. Manchmal zögernde Umsetzung von Features aus Java.

### **3.24.3 DevOps Approach**

Tools machen nur Sinn, wenn in CI/CD eingebunden, idelerweise Teil von Gradle/Maven. Quality-Gates ist die logische Konsequenz.

## **3.25 Zero Bug Policy**

Es gibt keine offene Bugs (wenn dann terminiert). Korrigieren (jetzt!) oder verwerfen (löschen). Es wird nur Qualität geliefert. Dadurch nur glückliche Benutzer. Kein Bug-Board mehr nötig. Kein Diskussion über Bugs, entweder werden sie gefixt oder gelöscht.

Das Pflegen von Bugs über Monate macht keinen Sinn.

## **3.26 Bad API's**

- Kunden werden gezwungen schlechten Code zu schreiben
- Namenskonvention ist nicht konsistent
- Zentraler Zugang zu Features in einzelner Klasse
- keine immutable Objekte werden genutzt
- nicht dokumentierte API
- Verwendung von alter Code-Style (for-Loops anstatt streams)

## **3.27 Bad Scrum / Agile**

- Definition of Done fehlt
  - Bsp: in Git eingecheckt und kompiliert
- Git Training fehlt
- Coding Guidelines fehlen
- Application wird nicht mehrmals pro Woche deployt
- fehlende DevOps-Disziplin # Architecture of Components and Subsystems

Ziel der Architektur ist es die menschlichen Ressourcen, die benötigt werden um die Software zu bauen oder weiterentwickeln, zu minimieren.

– Robert Martin

Grosses Design von Beginn an ist dumm. Aber kein Design von Beginn an, ist noch dümmer.

– Dave Thomas

Das System das wir aktuell bauen, gibt es sicher gleichwertige auf der Welt. Open Source Lösungen und Artikel ist eine ideale Quelle um an Informationen zu gelangen. Verführt zu *NIH - Not Invented Here Syndrome*. Gebot; Kopieren, anpassen, verbessern.

## 3.28 SOLID

Auf Komponentenebene gilt SOLID

- S - Single responsibility principle: hohe Kohäsion, nur ein Grund für Änderung
- O - Open/closed principle: offen für Erweiterung, geschlossen für Änderungen
- L - Liskov substitution principle: jede Subklasse kann Oberklasse ersetzen
- I - Interface segregation principle: Immer Collection zurückgeben oder als Parameter erwarten ( List und nicht ArrayList)
- D - Dependency Inversion principle: high-level Klassen sollten nicht von low-level Klassen abhängig sein, beide aber via Abstraktion

## 3.29 Patterns

Die GoF-Pattern sind eine gute Anlaufstelle um etwas umzusetzen. Aufgeteilt in drei Teile:

- Creational
- Structural
- Behavioral

Pattern sind Lösungsvorschläge, wie wiederkehrende Probleme elegant gelöst werden können. Die Patterns müssen für Problem evaluiert und ausgewählt werden. Patterns definieren auch eine gemeinsame Sprache, damit man sicher besser austauschen kann.

### 3.29.1 Builder Pattern Example

Insert code example from slides

## 3.30 Layered Architecture

- Ebene-Architektur verhindert Encapsulation. Die Interfaces zwischen den Layers sind auf die Daten (die ausgetauscht werden) abgestimmt.
- Abstraktion wird insofern verhindert, weil nahezu jeden Layer muss alle Konzepte kennen.
- Kohäsion und Koppelung
- Law of Demeter
- Tell don't ask

//todo: add slide content

### **3.31 Hexagon/Onion Architecture**

Fördert Domain Models und Event-basierte Ansätze. Weiter fördert sie das Mocking von Verbinder (Connectors), testing wird einfacher und die Integration vereinfacht

### **3.32 Java Ecosystem**

// todo some more stuff

- Exception Handling - runtime exceptions nutzen, Streams können checked exceptions nicht nutzen
- Für Multi Threading java.util.concurrent
- Patterns in API
- Immutability API (record)
- Java Trends (//todo research)
  - functional programming
  - immutability
  - reification
  - memory-efficiency
  - heterogeneous processors
  - fibers

### **3.33 Refactor**

Agressives Refactoring des Codes und Designs. Setter- wirklich nötig? Viele Setter verhindern Multithreading. Produktentwicklung ist lernen, das Design entsprechend so gestalten, dass es refactored werden kann. Agile meint Improvement.

#### **3.33.1 OOP Anti-Patterns**

- Singletons sind schlecht
- nie null zurückgeben
- modifizierbare Collections nicht zurückgeben
-

# 4 Quality Attributes of Software Architecture

Architektur ist eine Hypothese, welche durch die Implementation und einer Messung prüfbar wird.

The only way to go fast, is to go well.

– Robert C. Martin

## 4.1 Characteristics

Änderungen sollten günstig sein. Feedback Loop sollte implementiert sein - Design/Development sind empirische Aktivität. Keine Spekulationen machen, weil nur unnötig Komplexität hinzugefügt wird. Immer an *drei* Dinge denken (Risiken), die schiefgehen können. In *kleinen* Teams arbeiten, um *gute* Software zu produzieren.

## 4.2 Functional Requirements

Anforderungen sollen **Requirements - SMART** definiert sein.

## 4.3 Stories as Functional Requirements

Stories müssen **Stories - INVEST** sein.

### 4.3.1 Stories

Als *Rolle* möchte ich *Ziel/Wunsch*, um *Nutzen*.

#### Akzeptanzkriterien

- Kriterium1
- Kriterium2

Eine Story wird erzählt, sodass eine Diskussion gestartet werden kann. Daraus ergeben sich Umsetzungen.

### 4.3.2 Use Cases

Use Cases sind tot - viel zu Papierlastig, sollte vergessen werden.

- related Use Cases sind Epics
- Primary Actors sind Personas
- Main Scenario sind Stories
-

### 4.3.3 Validation

TDD ist ein Sicherheitsnetz für Refactoring und Dokumentation. ATDD auf Stufe Subsystem-Level oder System Level mit Java Modules (ArchUnit). User Interface Tests, Selenium - minimal halten – Sind umständlich.

### 4.3.4 Testing Quadrants

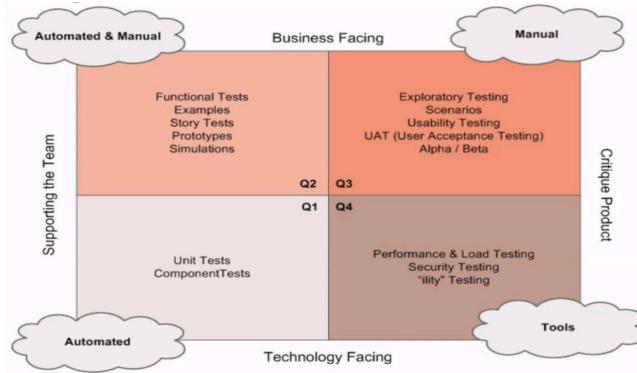


Abbildung 4.1: Testing Quadrants

### 4.3.5 Testing Pyramid

Möglichst alle Tests automatisieren.

## 4.4 Architecture Goals

- Komplexität *reduzieren*
- Änderbarkeit *steigern*
- Parallele Entwicklung *ermöglichen*

drei Paradigmen: strukturiert, objekt-orientiert und funktional

## 4.5 Quality Attributes

- lose Kopplung
- hohe Kohäsion
- Design für einfache Änderungen
- Separation of Concerns
- Information Hiding (DTOs zerstören es)
- Good Practices; DDD, legibility of artifacts, git, Infrastructure as Code
- Abstraktion
- Modularität
- Verfolgbarkeit
- Betriebskosten minimieren (DevOps) - tracing, logging, monitoring
- Selbst dokumentierend - Clean Code und JavaDoc
- Incremental Design

## 4.6 How to reach these Goals?

- Spikes
- Experience and ask experts
- Codified knowledge
- Copy, modify, mutate, improve
- Refactor
- Unlock collective wisdom - fragen im Internet (Foren) stellen
  - bei fragen Fragen!
  - BugReports/Change Requests stellen

## 4.7 Quality Citations

- Lowering quality lengthens development time
- Die Codequalität ist invers proportional zum Effort um ihn zu verstehen
- Guten Code cleveren Code bevorzugen
- schlechte Qualität verhindert Performance

## 4.8 Refelction

- Wie kann ich schneller lernen?
- Was kann im Team geändert werden, um uns zu verbessern? - wie kann Risiko minimiert werden
- Wie können wir bessere Produkte liefern?
- wie kann die Qualität gesteigert werden?

# 5 -ility-Attributes

Non-functional Requirements wie Scalability, Performance Efficiency, Security, Reliability usw.

## 5.1 Truths

Funktion muss erfüllt sein und korrekt laufen.

Build the right product, build the product right

– unbekannt

Anforderungen definieren nicht User und Kunden, sondern sie äußern *Wünsche*. Technische Umsetzung, wie Wunsch umgesetzt wird, damit Wunsch erfüllt wird ist unsere Sache. Validation (Architect ist Berater) und Verifikation (Architekt ist Bestimmer) der Anforderung im Auge behalten.

Weil Akzeptanztest die Requirements prüfen ist sichergestellt, dass Anforderung erfüllt werden. Reine Metriken bringen wenig, Argumentationskette ist wichtig.

Scrum ist empirisch-, Resultat-driven. Etwas versuchen, prüfen und Erkenntnisse sammeln und Entscheide ableiten.

*ATAM* ist zu langsam und ist sehr teuer - sowohl auch *architecture driven design*. Manuelle reviews sind obsolet, keine pull requests weil langsam und daher teuer. Sinnvoll ist pair-work ( virtuell oder co-located).

## 5.2 Non-Functional Requirements

Jedes hat folgende Inhalte

- Context (Zeitraum, Wo - Kunde, intern, usw.)
- Warum
- Muss messbar sein (benutzerfreundliches UI, System muss schnell sein sind schlechte Requirements)
- Minimalen Wert
- Optimalen Wert
- Maximalen Wert - overengineerd

Für Werte ist besseren Ansatz; in 95% der Anfragen sind unter 0.5s, 98% sind unter 1s, usw..

Sammlung der ility-Attribute der letzten Jahre.

## Some -ility Attributes

Accessibility, accountability, accuracy, adaptability, administrability, affordability, agility, **auditability**, autonomy, availability, compatibility, composable, configurability, **correctness**, credibility, customizability, **debugability**, degradability, determinability, demonstrability, dependability, **deployability**, discoverability, distributability, durability, effectiveness, efficiency, **usability**, extensibility, failure transparency, fault tolerance, fidelity, flexibility, inspectability, installability, integrity, interoperability, learnability, **maintainability**, manageability, mobility, modifiability, modularity, operability, orthogonality, **portability**, precision, predictability, process capabilities, producibility, provability, recoverability, relevance, **reliability**, repeatability, reproducibility, resilience, responsiveness, reusability, robustness, **safety**, scalability, seamlessness, self-sustainability, serviceability, sustainability, tailorability, **testability**, timeliness, **traceability**

Abbildung 5.1: Fett markiert die wichtigsten ility-Attribute

## 5.3 Fitness Functions

Sind die Units-Tests für non-functional Requirements. Ist Mischung aus Umsetzung, Analyse und xxx mit Fitness Functions kann Analyse verifiziert werden.

todo: erklärung hier einarbeiten. <https://www.thoughtworks.com/insights/articles/fitness-function-driven-development>

Führt zu *double loop architecture* ist ein Prozess welcher sicherstellt, dass die Architektur dem Businessnutzen widerspiegelt.

Beispiele für Fitness Functions

- Static code analysis (Security, ...)
- Unit test frameworks
- Penetration testing
- load testing
- monitoring tools
- logging

## 5.4 Assumptions

Architektur, wie Business Kapazitäten und Infrastruktur wird in Code ausgedrückt durch den Nutzen von Fitness Functions. Diese bestehen aus *Code* und werden als Teil des CI/CD oder Monitoring ausgeführt.

### 5.4.1 Combining Fitness Functions

- Atomic + triggered: ArchUnit rules während CI
- Holistic + triggered: kombiniert security und Scalability (das ganze System wird getestet)
- Atomic + continual: Test REST endpoints Verbs und Error-Messages (durch heartbeat)
- Holistic + continual: Test Resilienz

Triggert eher in CI/CD, Continual

#### **5.4.2 Functions Examples**

- Code Quality in SonarQube messen, Qualität in SQ konfigurierbar
- UAT (User Acceptance Test) darf nicht zwei Versionen vor der Prod-version sein
- Keine Geheimnisse in Git - OWASP
- Security Testing Stage
- Deploy mit einem anderen Application-Service Account
- zwei Approver vor dem Release in Produktion

Fitness Function sind auch Teil der Produktionsumgebung. Auch für heikle Daten - dafür gibt es Geheimhaltungsklauseln.

Geben Auskunft über

- Mean Time between Failures
- Max Time to Recover (wichtigstes Kriterium in agilen Umfeld)
- Response Time
- Latency in your network (grossen impact auf Architektur)
- Resource usage (was ist Nutzlast, Spitzen?), lohnt sich mehr HW-Ressourcen oder mehr Leute

In Produktion erhalten wir wertvolle Informationen, die in die Entwicklung zurückfliessen sollten. Damit Produkt besser wird.

### **5.5 Code Quality**

- Modifiability
- Manageability
- Adaptability
- Legibility (lesbarkeit, wie messbar? z.B. kein Typo in Kommentar, wenn international dann englisch )

### **5.6 Resilience and Operability**

- Stability
- Resiliency
- Availability
- Recoverability

### **5.7 Performance and Security**

- Scalability
- Stability
- Response Time
- Security

### **5.8 Micro Profile**

Von Java zur Verfügung gestellt, z.B. Open API, Rest client, Heath, Jakarta, Streams, usw.

# 6 Architecture Documentation

Die Geschichte sollte auch in die Dokumentation einfließen, wieso gewisse Entscheide gefällt wurden.

## 6.1 Truths

- Source Code ist die Architektur
- es ist teuer und aufwändig die Dokumentation mit dem SourceCode zu synchronisieren
- bei Agile geht es um Menschen, deren Interaktionen, Stories und Diskussionen nicht über Prozesse und Werkzeuge
- ATAM, TOGAF, IEEE-SW sind veraltet und obsolet
- Traditionelle PM-Standards wie Hermes, Prince2 und PMI sind archäologischer Natur
- nie Word nutzen, es ist proprietär und kann nicht in VCS integriert werden
- je mehr Textdokumente, umso mehr Synchronisierungsfehler können es geben.
- Papier ist nutzlos

## 6.2 Why Should You Document

Es sollte kommunikativ und informativ sein und auf Leser angepasst. Erklärungen sollten überwiegen, anstatt korrekte formelle Notation. Kritische Informationen enthalten. Erklärende Einschränkungen des Systems. Einfachheit der Schönheit vorziehen. Erkenntnisse und/oder Kausalitätsketten sind relevant nicht Meinungen.

### 6.2.1 Domain Driven Models

- Code ist Dokumentation
- Kleine Modelle mit Erklärungen
- Event Diagramme
- Akzeptanztest Reports
- Verfolgbarkeit zwischen Code, Tests und dazugehörigen Anforderungen

### 6.2.2 UML for small Models

PlantUML! Oder mindestens Textbasiert, damit in VCS verwaltbar.

#### 6.2.2.1 C4 Model for Systems

Variante von Dude auf youtube. Sagt dass Overview unbedingt graphisch dokumentieren - ist eher statisch, gibt Überblick. Lohnt sich also. Rest nur wenn wirklich nötig.

1. SW Arch is not about Big Design up-front
- 2.

## Vorgehen

1. Overview first
  - System Context (Schnittstellen nach aussen)
  - Containers (große Blöcke der Applikation, wegen untersch. Technologie oder Verteiltes System) (oft Docker)
1. Zoom and Filter
  - Komponenten (Module (Java))
1. Detail on demand
  - Klassen

## 6.3 Architectural Design Record

Der ADR ist ein Dokument wieso Entscheide gefällt wurden. Inklusive Kontext und Gründe die zum Entscheid gefüllt haben. Die Geschichte gehört zum Model und kann in die Versionkontrolle hinzugefügt werden.

## 6.4 Rules for Documentation

Nur stabile, implementierte Konzepte erfassen. Keine spekulativen Ideen. *Lebende* Dokumentation sind berreichend, kollaborativ und benötigen oft weniger Pflege. Siehe auch [www.javadoc.io](http://www.javadoc.io)

Dokumentation sollte simple, aber nicht zu simpel verfasst sein. Aufwand und Ertrag muss stimmen, auch beim Einsatz von Tools (Lizenzen).

Informationen öffentlich und zentral zur Verfügung stellen. Es sollte durchsuchbar sein!

## 6.5 Acceptance Tests

Jede Story oder Anforderung sollte Akzeptanzkriterien haben welche mit Acceptanztests verifiziert werden. Akzeptanzkriterien ist eine ausführbare Spezifikation und immer aktuell. Hier ist die Tracability implizit

## 6.6 Fitness Functions

Automatisch Tests für non-functional Requirements, Reports bestätigen die Validation von allen non-functional Requirements. Traceability ist implizit.

## 6.7 API Documentation

- JavaDoc
- Teil einer statischen Website
- Integriert in moderne IDE

## 6.8 Configuration as Code

Jeder Aspekt des Systems sollte als Source Code gehandhabt werden, alles ist unter Git. Die Traceability ist implizit gegeben und auch auditfähig.

## 6.9 Static Web Sites

- Hugo, Jekyll
- Docsy plugin für Hugo
- Pages für Github, Gitlab, Bitbucket
- JavaDoc als Teil der statischen Website
- Synchronisiert mit git-Repo
- Published täglich

# 7 Architectural Trends I

Aktuelle Trends in Schweiz und Europa.

## 7.1 Truths

- Agile Architektur entsteht während der Entwicklung
- Agile Architektur entwickelt sich
- Architektur ist für die Technologie/Infrastruktur relevant bzw. abhängig (Mobil First, FatClient)
- Nicht alle Architekturaspekte sind Technologie relevant (Reliability)
- SOA ist tot - ist nicht angepasst auf multi-verteilte Firmen Workflow/Orchestration nicht möglich über mehrere Standorte
- Monolithische Lösungen müssen vorsichtig angewandt werden - Modular Monolith Solution wäre ein guter Ansatz
- Applications sind häufig mobile first oder Browser first - PWA!
- Browser Lösungen müssen oft neu geschrieben werden - alle 18 bis 24 Monate

## 7.2 Domain Driven Design

Fokus ist die fachliche Domäne anstatt die technische Welt. Man muss Benutzer und seine Sprache verstehen. Der Code reflektiert das Benutzermodell.

## 7.3 Hexagonal or Onion Architecture

Hexagonal kommt aus OpenSource/Java Ecke, Onion aus .net die Idee ist aber dasselbe.

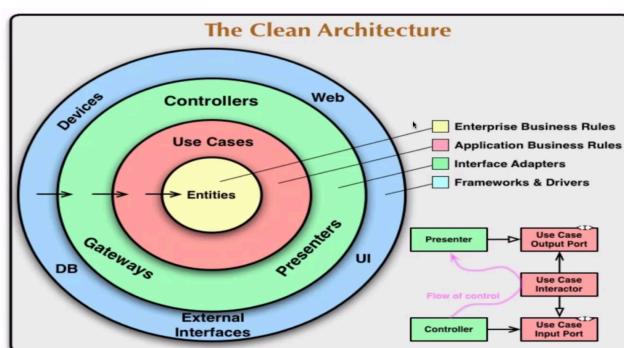


Abbildung 7.1: Hexagonal or Onion Architecture

## 7.4 Domain Driven Development

- Man spricht Benutzersprache, Klassen und Methode entsprechen der Domainlanguage
- Bounded Domains - in Subsysteme aufteilen
- benötigt aber klares Interface zwischen Bounded Domains
- Definition of Shared Kernels - Shared models wenn möglich vermeiden - Domains müssten sonst immer von allen direkt übernommen werden.

## 7.5 Domain Ubiquitous Language

Es braucht eine *gemeinsame* Sprache zwischen Kunde und SourceCode.

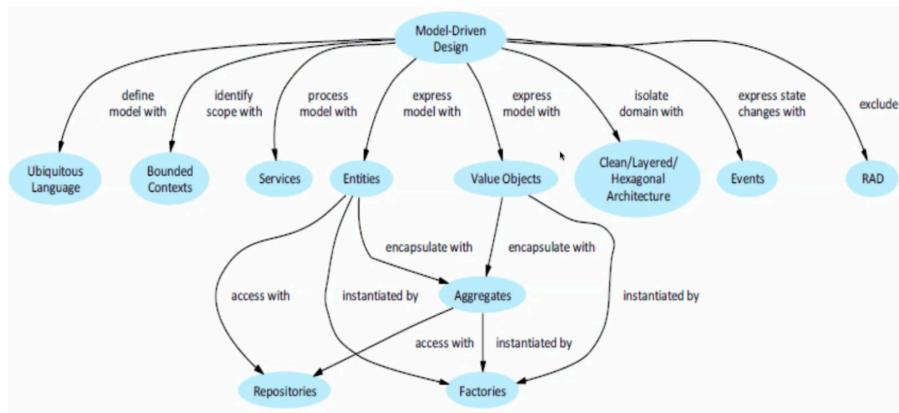


Abbildung 7.2: Domain Driven Design

## 7.6 Domain Driven Design

- Entitäten haben Identitäten (und einen Zustand, Verhalten)
- ValueObjekte (Adressen/Telefonnummer) haben keine Identität, sie tragen nur Informationen.
- Aggregate - ver-/binden Entities und ValueObjects
- Repositories speichern Entities und ValueObjects
- Factories erstellen Entities und ValueObjects
- Ein Service hat keinen Zustand, sondern *nur* ein Verhalten auf Objekten.

## 7.7 Domain Events

Events werden in Vergangenheit definiert (es wurde gebucht) und lösen einen Command aus.

- Event - Etwas hat sich geändert (Order Placed)
- Command - Objekt wird an Command-Handler geschickt (Place Order)

## 7.8 Multiple Teams

Mit DDD bzw. Bounded Domains kann die Arbeit parallelisiert werden. Änderungen (in einer Domäne) können unabhängig voneinander gemacht werden.

## 7.9 Immutability and Functional Style

- Entity Objekte sollten eine nicht änderbare Identity
- ValueObjekte sollte nicht änderbar sein -> man mutiert (erstellt neue). Dadurch überrascht man niemand (Record Konzept von Java)
- Alle Werte werden beim Erzeugen gesetzt

## 7.10 Questions

- *What is an identity?*  
Etwas das nicht ändert (eduId)
- *Why should value object be immutable?*  
Um andere Services nicht zu überraschen die mit dem gleichen Objekt arbeiten
- *Why is a bounded context important?*  
Es handelt sich um fachliche abgegrenzte Einheiten
- *Why is continuous Integration a central activity?*  
Durch viele kleine Änderungen benötigt man schnelles Feedback auch um schnell allfällige Fehler zu finden.
- *Reflect about Domain Specific Languages DSL* DSL codifiziert eine Domäne, die Sprache der Benutzer

### 7.10.1 DDD Anti-Patterns

- Anemic Domain Objects
- Repetitive DAO's - Data Access Object
- Fette Service Layer wo Service-Klassen alle Business Logik enthalten
- Feature Envy: Methoden sind zu interessiert an Datene von anderen Klassen

## 7.11 Event Storming - Ignite your DDD

- Start beim Domain Event - Was passiert im System
- Was triggert diesen Event? -> Command
- Das Read-Model zeigt etwas an

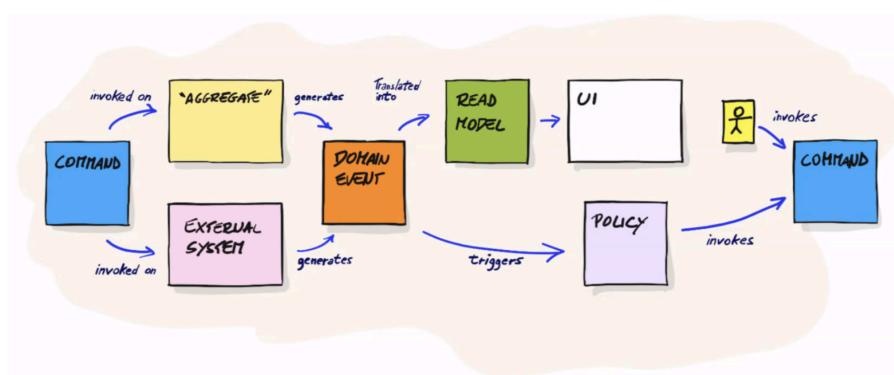


Abbildung 7.3: Event Storming

## 7.12 Micro Services

- Mapping Bounded Domains und Microservices
- Informationsaustausch über JSON oder Protobuf (wenn JSON zu langsam)
- REST Services
- GraphQL services (erst wenn REST zu langsam)
- asynchronous service
- reactive systems (erst am Schluss - schwierig zu debuggen und Support im Betrieb wird teurer)
  - Event based
  - Eventual consistency
- difficulties: latency, Netzwerkstabilität, Bandbreite ist erschöpflich, Netzwerk sollte sicher sein, Topologie ändert, es gibt mehrere Administrator, Transportkosten, Homogenes Netzwerk
- User Interfacee sind schwieriger
  - micro user interfaces
  - GraphQL
- Logging ist schwieriger
  - ELK nutzen - Elasticsearch, logstash, Kibana

## 7.13 Ball of Mud

Grosser Ball an Verknüpfungen und Abhängigkeiten in einem System.

## 7.14 Monolith to Modular

1. Grossen Service und dessen Klassen als separate Solution oder Modul auslagern
2. Bounded Domain definieren
3. Gespeicherte Daten in ein eigenes Schema migrieren
4. Deploy als separaten Micro Profile Service

## 7.15 Refactor

Aggressiv und wiederholend Refactoring betreiben.

## 7.16 Evolvable Architecture

1. Dimensionen die von Evolution betroffen sind identifizieren
2. Fitness Functions pro Dimension definieren
3. Deployment Pipelines um Fitness Funktionen zu automatisieren
4. Starte mit der Weiterentwicklung

## 7.17 Wisdoms

Das *business problem* muss verstanden sein. Je mehr wiederverwendbar der Code ist, umso weniger verwendbarer ist er. Lieber kopieren, statt Vererbung. COTS (Commercial off-the-shelf) -Lösungen wenn man agil sein will. Frameworks vermeiden, aber Bibliotheken verwenden. Nutzlose Variabilität (mehr Plattformen, mehr DBs unterstützen) vermeiden.

Bester Ansatz für Architektur ist eine *Monolithischer Architekt*, günstiger, weniger komplex und schnellere Zyklen.

Um Modularität zu garantieren Java Module oder ArchUnit verwenden.

Die Speicherung muss auch modular aufgebaut sein (mehrere Schemas)