

SWEN – Zusammenfassung

Ramon Imper

16. Juni 2022

Inhaltsverzeichnis

1 Einführung Software Entwicklung	4	4.3.6 Refactoring	13
1.1 SWEBOK	4	4.3.7 Pair programming	13
1.1.1 Was ist SWEBOK?	4	4.3.8 Collective ownership	13
1.2 Plan-Driven / Plangetriebene Methoden	4	4.3.9 Continuous integration	13
1.3 Gesetze der Software Entwicklung	4	4.3.10 40 hours week	14
2 Agile Manifesto	6	4.3.11 On-site customer	14
2.1 Values von Agile Manifesto	6	4.3.12 Coding standards	14
2.2 Die "12 Principles" von Agile Manifesto	6	4.3.13 Test-Driven Programming/Development (TDD)	14
2.3 Bedeutung des agilen Manifesto einordnen	7	4.3.14 Slack	14
3 Agile	8	4.3.15 Spike	14
3.1 Risiko	8	4.3.16 Incremental Design	15
3.2 Cost of change	8	4.3.17 Self-organized Team	15
3.3 Variablen der Software Entwicklung	9	5 Software Craft	16
4 eXtreme Programming	10	5.1 Gründe, weshalb das Manifest für Software Craft notwendig wurde	16
4.1 Was ist eXtreme Programming	10	5.2 Prinzipien der Software Craft	16
4.2 Begriffserklärung	10	5.3 Wie übt man als Software Entwickler	17
4.2.1 Core Values	10	5.4 Coding Dojo	17
4.2.2 Principles	11	5.4.1 Für was braucht man Coding Dojo?	17
4.2.3 Practices	12	5.4.2 Prinzipien	17
4.2.4 Cost of change	12	5.5 Coding Katas	18
4.2.5 Business value	12	5.5.1 Was wird geübt	18
4.3 eXtreme Programming Praktiken	12	5.5.2 Merkmale einer Code-Kata	18
4.3.1 The Planning Game	12	6 Pyramid of Agile Competencies	19
4.3.2 Small releases	12	6.1 Die drei Ebenen der Pyramid of Agile Competencies	19
4.3.3 Metaphor	12	6.1.1 Agile Values	19
4.3.4 Simple design	12	6.1.2 Collaboration Practices	19
4.3.5 Testing	13	6.1.3 Technical Practices	19
7 User Stories	21	7.1 Ziel einer User Story	21

7.2	Anwendung	21	10.5	Scrum values	36
7.2.1	User Rollen erfassen	21			
7.2.2	User Stories erfassen	21			
7.3	Format einer User Story	22	11	SAFe (ScaledAgile Framework)	37
7.3.1	User Rollen	22	11.1	Werte von SAFe	37
7.4	Epic, Themes und Stories	23	11.2	Prinzipien von SAFe	37
7.5	Eigenschaften einer guten User Story	23	11.3	Ebenen von SAFe	37
7.5.1	Independent / Unabhängig	23	11.4	Release Train	38
7.5.2	Negotiable / Verhandelbarkeit	24	11.5	SAFe Rollen	38
7.5.3	Valuable / Werthaltigkeit	24			
7.5.4	Estimatable / Schätzbarkeit	24			
7.5.5	Small / Klein	24			
7.5.6	Testable / Testbar	25			
8	Estimation and Planning	26	12	Software Architecture	39
8.1	Begriffserklärungen	26	12.1	Software Architektur	39
8.1.1	User Stories	26	12.1.1	Vorteile guter Software Architektur	40
8.1.2	User Roles	26			
8.1.3	Epics	26	12.2	Software Architekt	40
8.1.4	Themes	26	12.2.1	Die Aufgabe des Architekten	40
8.1.5	Story Points	26	12.3	Allgemeine Systemeigenschaften	41
8.1.6	Velocity	26	12.3.1	Komplexität	41
8.1.7	Planning Poker	27	12.3.2	Konformität	41
8.1.8	Conditions of Satisfactions	27	12.3.3	Formbarkeit	41
8.1.9	Levels of planning	27	12.3.4	Unsichtbarkeit	41
8.1.10	Product Backlog	28	12.4	Architektur Stile	41
8.1.11	Priorisierung	28	12.5	Software Pattern	42
8.1.12	Techniques for Estimating	28	12.5.1	Transactions Script	42
8.1.13	Relative Schätzung	28	12.5.2	Domain Model	42
8.1.14	Planning for value	29	12.5.3	Table Module	42
8.1.15	Cost	29	12.6	C4 Modell	43
8.1.16	Financial value	29	12.6.1	Level 1: System Context diagram	43
8.1.17	Risk	29	12.6.2	Level 2: Container diagram	43
8.1.18	New Knowledge	29	12.6.3	Level 3: Components diagram	44
8.1.19	Kano-Model of Customer Satisfaction	30	12.6.4	Level 4: Code-level diagrams	45
8.2	Planungsebenen im agilen Kontext	30			
9	Build Automation, CI, CD, DevOps	31	13	SOLID Principles	46
9.1	Arten von Software Automation	31			
9.2	Typen der Automatisierung	31			
9.3	Software Automation Pipeline	32			
10	Scrum	33	14	Cynefin Framework / Codefin	47
10.1	Scrum Charakteristiken	33	14.1	Nutzen von Cynefin	47
10.2	Scrum Rollen	33	14.2	Cynefin	47
10.3	Scrum Ceremonies / Events	34	14.3	Framework	47
10.4	Scrum Artifacts	36	14.4	Exaptation	48
			14.5	4+1 Domains (Simple, Complicated, Complex, Chaotic, Disorder)	48
			15	Kanban / Lean Software Development	49
			15.1	Zusammenhänge	49
			15.2	Prinzipien von Kanban	49
			15.3	Praktiken von Kanban	49
			15.4	Werte von Kanban	50

15.5 Kanban Board	51	15.8 Prinzipien von Lean	52
15.6 Begriffe und erklärung	51	15.8.1 Verschwendungen eliminieren .	52
15.7 Lean	52	15.9 Kanban vs. Scrum	53

1 Einführung Software Entwicklung

Lernziel

Sie kennen SWEBOK. Sie können Vor-/Nachteile der plangetriebenen Methoden aufzählen. Sie wissen, für welche Arten von Projekten die plangetriebenen Methoden verwendet resp. nicht verwendet werden sollen.

Sie können die drei Gesetze der Software Entwicklung erklären (Selbststudium).

1.1 SWEBOK

SWEBOK steht für **SoftWAre Engineering Body Of Knowledge**

1.1.1 Was ist SWEBOK?

SWEBOK ist eine Wissenssammlung für Softwaretechnik. SWEBOK versucht die Disziplin Software Engineering zu vereinheitlichen. Herausgegeben wird es von der IEEE Computer Society. Infos zu SWEBOOK gibts [hier](#) und [hier](#).

1.2 Plan-Driven / Plangetriebene Methoden

Vorteile	Nachteile
Festgelegte Projektergebnisse und -kosten	Höherer Administrationsaufwand
Prozesse definierbar	Änderungen sind aufwendiger
Definierter/strukturierter Netzplan	Nicht geeignet für Projekte mit unklaren Anforderungen
Optimale Nutzung der Ressourcen	Viele Regeln
Risikomanagement	
Nutzung vom Earned Value Management	
Zielgruppengerechtes Reporting	
Einfache Einbindung von externen Dienstleistern	

Plan-Driven wird von folgenden Projektarten verwendet:

- Projekte, die einen Fixen Umfang haben (Bauprojekte)

Folgende Projektarten verwenden Plan-Driven **nicht**:

- Projekte, die kontinuierlich angepasst und verbessert werden (Software)

1.3 Gesetze der Software Entwicklung

Die drei **Gesetze** der Software Entwicklung sind:

1. Humphrey's Law: "Menschen wissen nicht, was sie wollen, bevor sie es sehen"

Dieses Gesetz ist einer der Hauptgründe, weshalb Google, Microsoft und Apple Versionen zu veröffentlichen.

fentlichen anstatt vollendete Software. Ein Entwickler weiss nicht, was ein User will und ein User weiss nicht, was ein Entwickler will. Also wird dem User etwas gezeigt, er probiert das aus und gibt ein Feedback. Dieses Gesetz ist eine Abkehr/Abweichung vom Versuch, eine perfekte Lösung zu präsentieren. Es gibt keine Perfektion, es gibt nur konstante Verbesserung. Das Gesetz auf unser Leben übersetzt: Was in Ihrem Leben sollten Sie verbessern?

2. Ziv's Law: "Softwareentwicklung ist unvorhersehbar und kann nie vollends verstanden werden"

Auch bekannt als Unsicherheitsprinzip.

Dieses Gesetz besagt, dass bei Softwareentwicklung wie im Leben gilt: Alles ist unvorhersehbar. Besonders, wenn man es zum ersten Mal macht. Der zweite Teil des Gesetzes: Das Meiste im Leben ist so kompliziert, dass wir es nicht verstehen. Frage an sich selber: Was ist meine Strategie, um mit Unsicherheit umzugehen?

3. Conway's Law: "Software ist ein Spiegel der Firma und der Menschen, die sie entwerfen"

Dies ist das älteste der drei Gesetze (1960er Jahre). Ist vielleicht das Weitreichendste. Es besagt, dass alles, alle Produkte, die wir herstellen, alle Arbeiten, die wir verrichten, ein Spiegel unserer selbst sind. Betrachten Sie also heute mal Ihr Tageswerk, und fragen Sie sich: Erkenne ich mich selbst in meiner Arbeit wieder?

2 Agile Manifesto

Lernziel

Sie können die «Values» des «Agile Manifesto» aufzählen und deren Bedeutung erklären. Sie kennen auch die «12 Principles» hinter dem «Agile Manifesto» und können diese erklären.

Sie können die Bedeutung des agilen Manifesto für die Software Entwicklung einordnen.

2.1 Values von Agile Manifesto

Die Werte des Agilen Manifesto sind folgende:

Englisch	Deutsch
We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:	Wir entdecken bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen, es zu tun. Durch diese Arbeit haben wir Werteschätzung erfahren:
Individuals and interactions over processes and tools	Individuen und Interaktion sind wichtiger als Prozesse und Tools
Working software over comprehensive documentation	Lauffähige Software ist wichtiger als durchgängige Dokumentation
Customer collaboration over contract negotiation	Zusammenarbeit mit Kunden ist wichtiger als Vertragsverhandlungen
Individuen und Interaktion sind wichtiger als Prozesse und Tools	Auf Veränderung reagieren ist wichtiger als einem Plan folgen
That is, while we value the items on the right, we value the items on the left more	Das heisst, während wir die Gegenstände auf der rechten Seite schätzen, schätzen wir die Gegenstände auf der linken Seite mehr

2.2 Die "12 Principles" von Agile Manifesto

Die "12 Principles" von Agile Manifesto sind:

Customer Satisfaction:

Höchste Priorität hat die Zufriedenstellung des Kunden durch frühe und regelmässige Lieferung wertvoller Software

Welcome Change:

Neue und veränderte Anforderungen sind auch zu einem späten Entwicklungszeitpunkt willkommen. Agile Prozesse sehen die schnelle Adaption von Änderungen als Wettbewerbsvorteil für den Kunden.

Deliver Frequently:

Lauffähige Software wird regelmässig (einiger Wochen/Monate) geliefert. Mit einer Präferenz auf kürzere Zeitabstände

Working Together:

Fach und Entwicklung müssen während dem Projekt täglich zusammenarbeiten

Motivated Team:

Projekte müssen rund um motivierte Individu-

en organisiert sein. Ihnen muss die notwendige Umgebung und Unterstützung gegeben werden. Und es muss das Vertrauen gegeben werden, dass sie den Auftrag erfüllen werden.

Face-to-Face:

Die effizienteste und effektivste Methode Informationen innerhalb eines Entwicklungsteams auszutauschen, ist die direkte Kommunikation Angesicht zu Angesicht.

Working Software:

Lauffähige Software ist die zentrale Messgröße für die Messung des Projektfortschritts

Constant Pace:

Agile Prozesse stehen für nachhaltige Entwicklung. Die Sponsoren, Entwickler und User sollten fähig sein, einen konstanten Arbeitstakt durchzuhalten.

Good Design:

Der ständige Fokus auf die technische Exzellenz und auf gutes Design verbessert die Agilität

Simplicity:

Einfachheit – die Kunst diejenige Arbeit zu maximieren, die nicht getan werden muss – ist essenziell.

Self Organisation:

Die beste Architektur, die besten Anforderungen und das beste Design entsteht durch die Arbeit selbstorganisierter Teams

Reflect & Adjust:

Das Team reflektiert regelmäßig, wie dass es effizienter werden kann, verbessert und adaptiert seine Arbeitsweise entsprechend

2.3 Bedeutung des agilen Manifesto einordnen

Durch die 12 Prinzipien ist klar, dass die Zufriedenstellung des Kunden einen sehr hohen Stellenwert hat. Mit der Kombination, dass Agile Software auch zu späteren Zeitpunkten schnell adaptiert werden kann, ist diese Methode in der heutigen Zeit sehr gefragt. Man bringt z. B. nicht erst jedes Jahr einen neuen Release, sondern alle paar Wochen bis Monate. Somit werden immer kleine Änderungen, die von den Usern bei einem Feedback gegeben wurden, in kurzen Intervallen angepasst. So haben die User eine stetige Verbesserung der App.

3 Agile

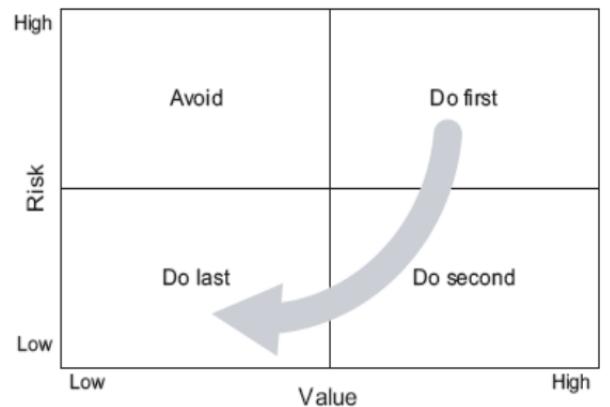
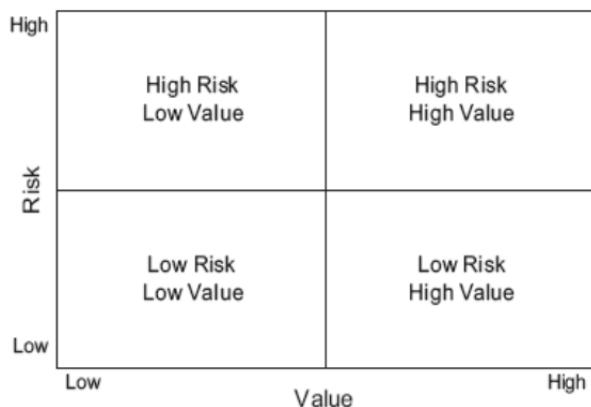
Lernziel

Sie kennen die folgenden Begriffe und können sie erklären: Risk, Cost of change und die Variablen der Software Entwicklung (Iron Triangle) und deren Kräfte erklären.

3.1 Risiko

Risiken sind etwas Alltägliches. Man muss abschätzen, ob sich der Ertrag der Tätigkeit mit dem jeweiligen Risiko lohnt. Dies ist auch bei Software-Projekten so. Zudem sind fast alle Projekte mit einem enormen Risiko behaftet.

- Risiken sind integraler Bestandteil der meisten Projekte
- Risiken haben eine Eintretenswahrscheinlichkeit und einen Impact auf das Projekt im Falle eines Eintretens
- Es gibt verschiedene Risiken
 - Zeitliche Risiken
 - Risiken bezüglich kosten
 - Funktionale Risiken

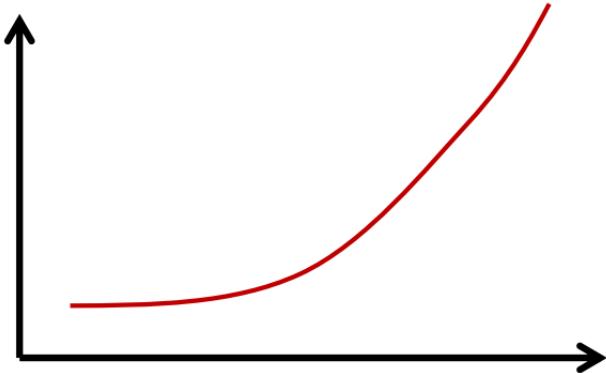


3.2 Cost of change

Wie viel kostet es, etwas am Projekt zu Ändern?

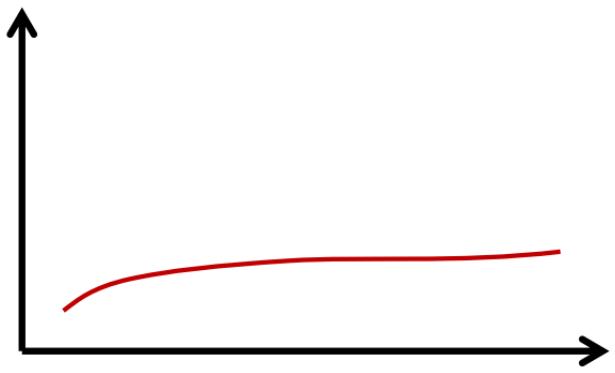
Es gibt zwei verschiedene Ansichten von Kosten in der Informatik.

Die traditionelle Ansicht der Kostenentwicklung



Änderungen **sind** kostspielig

Die eXtreme Programming Ansicht



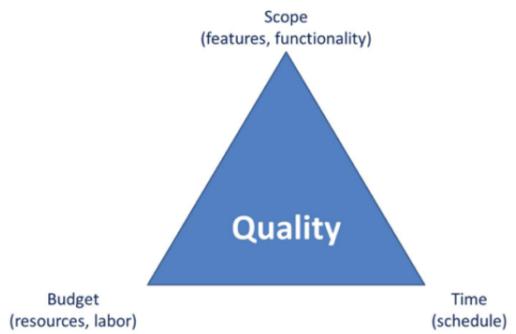
Änderungen sind **nicht** kostspielig

3.3 Variablen der Software Entwicklung

Die Variablen der Software Entwicklung sind Variablen, die ein Software-Produkt definieren.

Die vier Variablen der Software Entwicklung sind:

- Zeit → Normalerweise fix
- Budget → Normalerweise fix
- Qualität → Nicht verhandelbar
- Scope → Variable (vom Kunden aktiv verwaltet)



Externe (Kunden, Vorgesetzte) können drei dieser 4 Variablen definieren. Das Entwicklungsteam definiert die Vierte.

Diese vier Variablen sind denke ich selbsterklärend, wenn nicht mir sagen dann füge ich sie hinzu.

4 eXtreme Programming

Lernziel

Sie können erklären, was eXtreme Programming ist. Sie kennen die folgenden Begriffe und können diese erklären: Core values, Principles, Practices, Cost of change, Business value.

Sie kennen folgende XP Practices und können sie erklären: The Planning Game, Small releases, Metaphor, Simple design, Unit testing, Refactoring, Pair programming, Collective code ownership, Continuous integration, 40 hours week, On-site customer, Coding standards, Test-Driven Programming (TDD), Slack, Spike, Incremental Design, Self-organized team.

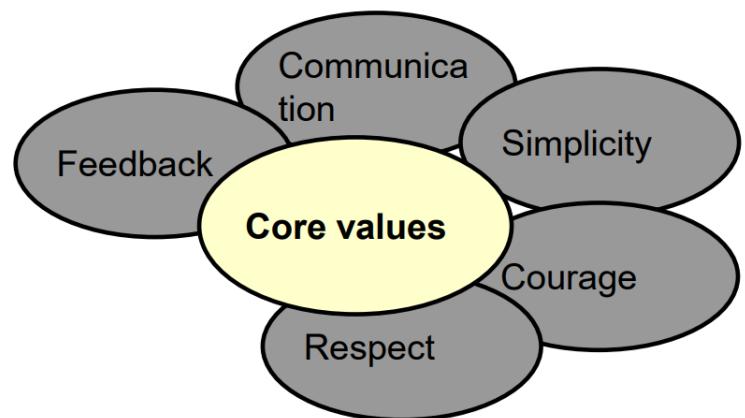
4.1 Was ist eXtreme Programming

eXtreme Programming (XP) ist eine **Methode zur Verbesserung der Softwarequalität und der Reaktionsfähigkeit** auf sich ändernde Kundenanforderungen. Als eine Art der agilen Softwareentwicklung befürwortet XP häufige Veröffentlichungen in kurzen Entwicklungszyklen, um die Produktivität zu verbessern und Kontrollpunkte einzuführen, an denen neue Kundenanforderungen übernommen werden können.

4.2 Begriffserklärung

4.2.1 Core Values

Die 5 Core Values sind:



4.2.1.1 Kommunikation

- Jeder ist Teil des Teams und es wird täglich Face-to-Face kommuniziert
- Wir arbeiten immer gemeinsam
- Zusammen werden wir die bestmögliche Lösung für unser Problem finden

4.2.1.2 Einfachheit

- Wir werden tun, was notwendig ist und um was wir gebeten worden sind, nicht mehr und nicht weniger
- Was wird den geschaffenen Wert im Verhältnis zum Aufwand maximieren
- Wir werden kleine einfache Schritte auf unser Ziel hin machen und Fehler zum Zeitpunkt ihres Auftauchens sofort lösen
- Wir werden etwas kreieren, auf was wir stolz sind und wir werden es lange zu vernünftigen Kosten unterhalten

4.2.1.3 Feedback

- Wir nehmen jedes Commitment einer Iteration ernst, indem wir lauffähige Software liefern
- Wir zeigen unsere Software früh und oft und hören sorgfältig zu und machen sämtliche notwendigen Änderungen
- Wir sprechen über das Projekt und adaptieren unseren Prozess entsprechend und nicht andersherum

4.2.1.4 Mut

- Wir sprechen die Wahrheit über den Fortschritt und die Schätzungen
- Wir dokumentieren keine Ausreden für das Scheitern, weil wir für den Erfolg planen
- Wir fürchten nichts und niemanden, weil niemals jemand allein arbeitet
- Wir werden uns Veränderungen anpassen wann immer sie auftreten

4.2.1.5 Respekt

- Alle geniessen und geben anderen den Respekt, den sie als wertvolles Teammitglied verdienen
- Alle bringen etwas ein, wenn es auch nur Enthusiasmus ist
- Entwickler respektieren die Expertise des Kunden und umgekehrt
- Vorgesetzte respektieren unser Recht, Verantwortung zu akzeptieren und für unsere Arbeit zu übernehmen

4.2.2 Principles

Die Fundamentalen Prinzipien des XP sind:

Rapid Feedback:

Schnellst mögliche Rückmeldung

Assume Simplicity:

Es kann davon ausgegangen werden, dass eine einfache Lösung existiert

Incremental change:

Änderungen erfolgen in der Regel inkrementell

Embracing change:

Änderungen sind zu erwarten und vorzusehen

Quality work:

Bestmögliche Lösung

Weitere Prinzipien:

- Teach learning
- Small initial investment
- Play to win
- Concrete experiments
- Open, honest communication
- Work with people's instincts, not against them
- Accepted responsibility
- Local adaption
- Travel light
- Honest measurement

4.2.3 Practices

Wird in Kapitel 4.3 beschrieben

4.2.4 Cost of change

Wird in Kapitel 3.2 beschrieben

4.2.5 Business value

4.3 eXtreme Programming Praktiken

Die Praktiken des eXtreme Programming sind:

4.3.1 The Planning Game

- Balance zwischen geschäftlichen und technischen Überlegungen
- Business entscheidet über:
 - Scope, Priorität und Zusammensetzung der Releases und Release-Daten
- Techniker entscheiden über:
 - Schätzungen, Impacts, Prozesse und Feinplanung

4.3.2 Small releases

- Jedes Release sollte so klein wie möglich sein und die wichtigsten Geschäftsanordnungen enthalten
- Das Release muss als Ganzes Sinn machen (keine halbwegs funktionierenden Features)
- Lieber einmal im Monat als zweimal im Jahr

4.3.3 Metaphor

- Jeder im Team muss ein "gemeinsames Verständnis-für das System haben"
- Jeder im Team muss einen "gemeinsamen Wortschatz-haben"
- Dies gilt für technische und nichttechnische Personen
- Was sind die Grundelemente des Systems und in welchen Beziehungen stehen sie zueinander?

4.3.4 Simple design

Das **richtige Design** für ein Softwaresystem ist eines, dass:

- Alle Tests ausführt
- Keine redundante Logik enthält

- Hat die kleinstmögliche Anzahl Klassen und Methoden
- Nur das einsetzen was man braucht, und nicht Dinge, die man evtl. noch braucht
- Entstehendes, wachsendes Design: Kein "Over designing" (Refactoring wird schwierig)

4.3.5 Testing

- Programmfunctionen ohne automatisierten Test gibt es nicht
- Die Tests werden Teil des Systems
- Die Tests ermöglichen es dem System, Änderungen zu akzeptieren
- Entwicklungszyklus:
 - Zuhören (Voraussetzungen)
 - Test (zuerst schreiben)
 - Code (am einfachsten)
 - Design (Refactor)

4.3.6 Refactoring

- Fragen Sie sich bei der Implementierung einer Funktion, ob es eine Möglichkeit gibt, den vorhandenen Quellcode zu verbessern, damit die Implementierung der Funktion einfacher ist
- Automatisierte Tests bieten ein Sicherheitsnetz für ein angstfreies Refactoring

4.3.7 Pair programming

- Der gesamte Produktionscode wird von zwei Personen geschrieben, die mit einer Tastatur und einer Maus einen Bildschirm betrachten
- Zwei Rollen, der Programmierer auf der Tastatur konzentriert sich auf die aktuelle Methode, der andere Programmierer denkt über den weiteren Kontext nach (Refactoring usw.)
- Paare ändern sich häufig

4.3.8 Collective ownership

- Jeder, der die Möglichkeit sieht, einen Teil des Codes aufzuwerten, muss dies jederzeit tun
- Jeder übernimmt die Verantwortung für das gesamte System. Nicht jeder kennt jeden Teil gleich gut, aber jeder weiss etwas über jeden Teil.

4.3.9 Continuous integration

- Code wird mindestens einmal am Tag integriert und getestet
- Der Build-Prozess muss automatisiert auf einem dedizierten Computer ablaufen
- Es werden automatisierte Tests durchgeführt, die es ermöglichen, Probleme frühzeitig zu erkennen.

4.3.10 40 hours week

- Nachhaltige Entwicklung. Der Aufwand sollte gleichmäßig verteilt werden
- Längere Überstunden wirken sich negativ auf die Produktivität aus
- Ziel: Jeden Morgen frisch, jeden Abend müde und zufrieden sein
- Sich nicht vor einem Computer zu befinden, bedeutet nicht, das System zu vergessen. Ein Schritt zurück führt oft zu "Aha!" SMomenten

4.3.11 On-site customer

- Ein echter Kunde muss physisch im Team sein, um seine Fragen beantworten zu können
- Echter Kunde = Benutzer, der das System nutzen wird.
- Der echte Kunde arbeitet nicht zu 100% an dem Projekt, sondern muss "da" sein, um Fragen schnell zu beantworten
- Der echte Kunde hilft auch bei der Priorisierung

4.3.12 Coding standards

- Kollektives Eigentum und ständiges Refactoring bedeuten, dass die Codierungspraktiken im Team einheitlich gelebt werden müssen

4.3.13 Test-Driven Programming/Development (TDD)

Bei der testgetriebenen Entwicklung werden Tests dazu benutzt, um die Softwareentwicklung zu steuern. Der Ablauf dieser Programmierung ist zyklisch:

- Ein Test wird geschrieben, der zunächst fehlschlägt
- Genau so viel Code implementieren, dass der Test ordentlich durchläuft
- Refactoring des Tests und des Codes

4.3.14 Slack

Planen Sie immer einige kleinere Aufgaben ein, die bei Bedarf gestrichen werden können, um die Gesamtverpflichtungen zu erfüllen. Übertreiben Sie nicht und liefern Sie nicht zu wenig, sondern setzen Sie realistische, erreichbare Ziele

4.3.15 Spike

- Ein Entwicklungsteam weiss nicht genug, um eine bestimmte Story formulieren zu können
- User Story um Know-How zu erlangen
- Timeboxing ist wichtig

4.3.16 Incremental Design

Das Inkrementelle Design beschreibt ein Vorgehensmodell zur Softwareentwicklung der **kontinuierlichen** Verbesserung, bei dem häufig in kleinen oder sogar kleinsten Schritten vorgegangen wird. Die Agile Softwareentwicklung basiert auf Inkrementellem Design.

4.3.17 Self-organized Team

Das Unternehmen legt die Prioritäten fest. Die Teams organisieren sich selbst, um den besten Weg zur Bereitstellung der Funktionen mit der höchsten Priorität zu finden.

5 Software Craft

Lernziel

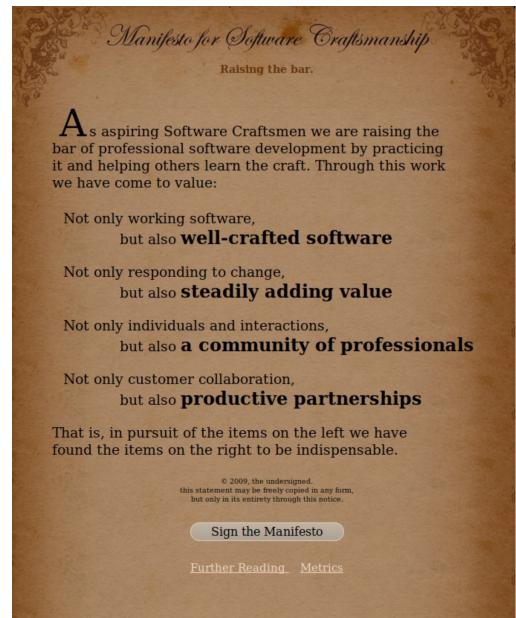
Sie kennen die Gründe, wieso das Manifest für Software Craft notwendig wurde. Sie kennen die Formate der Software Craft Community wie ein Austausch geschaffen wird und deren Prinzipien. Sie können erklären, wie man als Software EntwicklerIn üben kann. Sie können Coding Dojo und Code Katas ausführlicher erklären.

5.1 Gründe, weshalb das Manifest für Software Craft notwendig wurde

Ein weiteres Manifest wurde aus folgenden Gründen erschaffen:

- Gemeinschaft der Stimmen
- Sichtbarkeit schaffen
- Prinzipien etablieren
- Schulen entwickeln
- Leitfaden für neue Entwickler

Zudem wird beim **Agile Manifesto** stark auf den Projektprozess fokussiert. Die technische Exzellenz wird aber weitgehend vernachlässigt.



Das Motto der Craftsmanship-Bewegung: Wir sind es leid, Mist zu schreiben.

- Wie werden keine Unordnung machen, um einen Zeitplan einzuhalten
- Wir werden die dumme alte Lüge vom späteren Aufräumen nicht akzeptieren
- Wir werden nicht die Behauptung glauben, dass schnell "not-clean" bedeutet
- Wir werden die Option, es falsch zu machen, nicht akzeptieren
- Wir werden nicht zulassen, dass uns jemand zwingt, uns unprofessionell zu verhalten.

5.2 Prinzipien der Software Craft

- Individuals & Interactions (Voneinander Lernen)
- Clean Code
- Lebenslanges Lernen
- Ständige Verbesserung

5.3 Wie übt man als Software Entwickler

Es gibt verschiedene Möglichkeiten um zu üben.

- Code Dojo
- Code Katas
- Code Retreats
- Clean Code Developer
- Code Koans
- ...

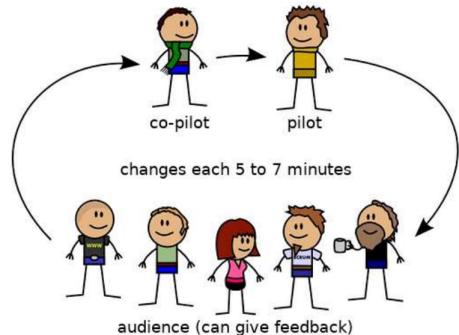


5.4 Coding Dojo

"Ein Haufen Coder kommt zusammen, programmieren, lernen und haben Spass. Das muss eine Erfolgsformel sein!"

~Emily Bache

Siehe auch: [Coding Dojo](#)



5.4.1 Für was braucht man Coding Dojo?

- Sichere Umgebung
- Keine Manager, keine Deadlines
- Alle Profis müssen üben!
- Spezielle Art zu üben
- Nicht alle Übungsformen sind gleichwertig oder haben denselben Effekt
- Sind auf die Entwicklung von Fähigkeiten ausgerichtet, die schwierig zu erlernen und leicht zu verlernen sind

5.4.2 Prinzipien

- **Erste Regel:** Design kann nicht ohne Code besprochen werden, Code kann nicht ohne Teste gezeigt werden
- Kommen Sie mit Ihren Erfahrungen
- Wieder lernen
- Entschleunigen
- Sich in die Suche nach einem Meister stürzen
- Sich einem Meister unterwerfen
- Einen Untergebenen anleiten

5.5 Coding Katas

"Entwickler sollten immer wieder an kleinen, nicht berufsbezogenen Code-Basen üben, damit sie ihren Beruf wie Musiker beherrschen" ~Dave Thomas

Kata Typen:

- Function Katas (FizzBuzz, Roman Numbers)
- Class Katas (Bowling, Stack)
- Library Katas (App Login)
- Application Katas (Tic Tac Toe, LOC Count)
- Architecture Katas (URL Shortener, Quizduell)
- Agile Katas
- ...

5.5.1 Was wird geübt

Kata ist ein japanisches Wort. Kata ist ein detailliert choreografiertes Bewegungsmuster, das für das Üben alleine gemacht ist, aber auch in Gruppen und im Unisono beim Training geübt wird!

In Karate: Eine Kata ist eine Übung, bei der Sie eine Form viele, viele Male wiederholen, während Sie bei jeder Wiederholung kleine Verbesserungen erzielen!

5.5.2 Merkmale einer Code-Kata

Definition: Eine Kata ist ein definierter Lösungsweg einer Code-Übung, der dafür gemacht ist, viele Male alleine, zu zweit oder in der Gruppe geübt zu werden und dabei kleine Verbesserungen zu erzielen.

Dauer: Die meisten Übungen sind recht kurz (30 Minuten bis 1 Stunde), so dass man sie als Routine in den Alltag einbauen kann!

Inhalte: Einige beinhalten Programmierung und können auf viele verschiedene Arten kodiert werden. Einige sind offen und beinhalten das Nachdenken über die Themen, die hinter der Programmierung stehen, z. B. Architekturkatas

Fokus: Der Punkt der Kata ist nicht, eine richtige Antwort zu finden. Es geht um die Dinge, die Sie auf dem Weg dorthin lernen. Das Ziel ist die Übung, nicht die Lösung!

Beispiel für ein FizzBuzz Kata: [Folien Seite 33](#)

6 Pyramid of Agile Competencies

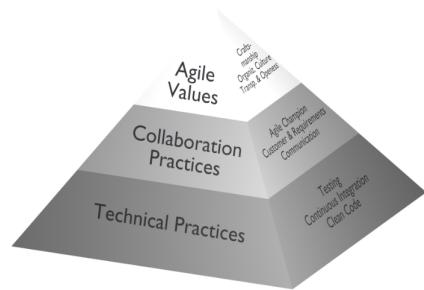
Lernziel

Sie kennen die «Pyramid of Agile Competencies» und können die drei Ebenen erklären.

6.1 Die drei Ebenen der Pyramid of Agile Competencies

6.1.1 Agile Values

Transparenz und Offenheit werden in der agilen Softwareentwicklung gross geschrieben. Das Team ergreift verschiedene Massnahmen, um sowohl die Organisation als auch den Kunden über den Fortschritt zu informieren und schnelles Feedback zu erhalten.



Organizational Culture

Im Prinzip gibt es drei Möglichkeiten:

1. Agiles Team, Organisation und Unternehmen
2. Agiles Team, nicht-agiles Unternehmen
3. Agiles Team, nicht-agile Organisation und Unternehmen

6.1.2 Collaboration Practices

Customer and Requirements: Eine intensive und häufige Kommunikation mit dem Kunden ist von grösster Bedeutung

Agile Champion: Catalyst Leadership style. Es soll in allen Projekten mindestens eine Person geben, die sich für Agilität eingesetzt hat.

Collaboration and Communication: Intensive und offene Kommunikation zwischen allen Beteiligten wird als eines der Schlüsselemente für erfolgreiche agile Projekte angesehen.

Drei wichtige Kommunikationsszenarien:

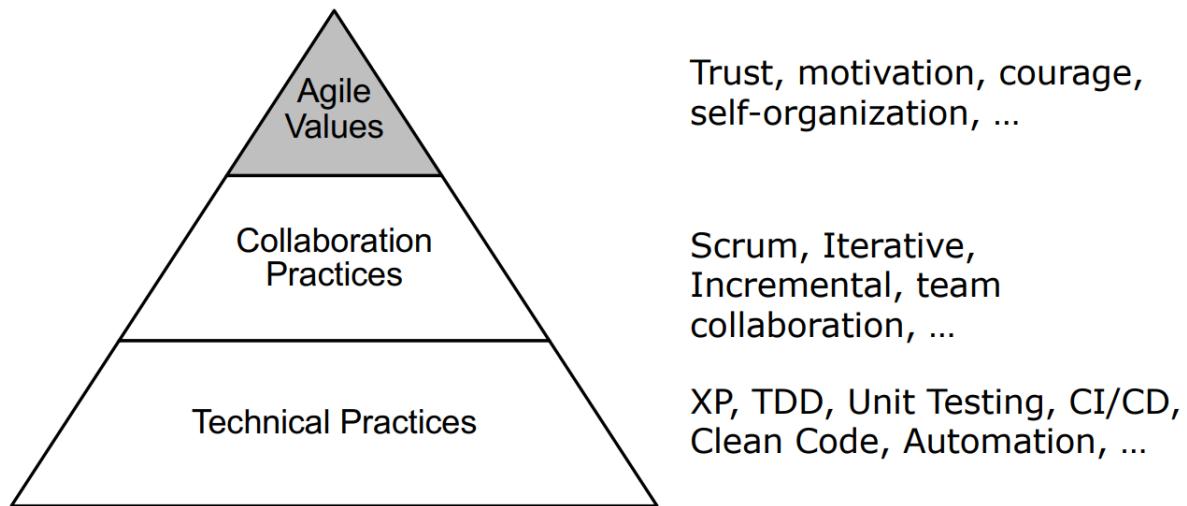
1. Die Teammitglieder selbst müssen untereinander intensiv kommunizieren
2. Das Team als Ganzes muss mit dem Kunden und den Endusern kommunizieren
3. Das Team muss eine gute Kommunikation mit dem Management aufbauen

6.1.3 Technical Practices

Testing: Automatisierte Tests auf Unit-Ebene sind gut etabliert und wird als ein absolutes Muss für die Bereitstellung einer guten Softwarequalität angesehen.

Continuous Integration wird als ein absolutes Muss angesehen, um Software mit hoher Frequenz liefern zu können.

Clean Code: Es wird als immer wichtiger angesehen, von Anfang an auf das Schreiben von gutem Code zu achten.



7 User Stories

Lernziel

Sie beschreiben was die Ziele, die Anwendung und wie das Format einer Userstory aufgebaut ist. Sie kennen die Zusammenhänge zwischen Epics, Themes und Stories. Sie kennen die Eigenschaften einer guten Userstory.

7.1 Ziel einer User Story

User Stories kombinieren die Stärken der schriftlichen und mündlichen Kommunikation. Durch User Stories, wird nicht ein Feature definiert, sondern eine Interaktion von einem User (oder auch dem System) mit der Anwendung.

7.2 Anwendung

7.2.1 User Rollen erfassen

Identifizierung und Auswahl einer definierten Menge von User Rollen durch:

1. Brainstorming einer initialen Menge von User Rollen
2. Organisation dieser initialen Menge
3. Konsolidierung der Rollen
4. Definition der Rollen

Kunde, Entwicklung sowie alle Beteiligten, die etwas vom Zielpublikum der Software wollen, müssen beteiligt werden

1. Jeder krallt sich ein paar Karten
2. Schreibt Rollennamen auf jede Karte
3. So schnell wie möglich ohne Beurteilung
4. Legt die Karten auf den Tisch
5. Spricht die Rollen laut aus, wenn man die Karte auf den Tisch legt

7.2.2 User Stories erfassen

Techniken zur Erfassung von User Stories:

Fragebogen

- Gute Technik, um mehr über Stories herauszufinden
- Hilft bei der Priorisierung bei grosser Nutzerbasis
- **Aber:** Nicht effektiv bei der Erfassung

Beobachtung

- Beobachtung mit oder ohne Wissen des Users
- Durch vorführen lassen

Interviews

- In vielen Fällen das Standard-Vorgehen
- Die Auswahl der zu interviewenden Personen ist zentral
- So viele Personen wie möglich interviewen
- Die meisten User sind nicht Experte im Verstehen der eigenen Bedürfnisse
→ **Keine** Fragen dieser Art: "Was wollen Sie eigentlich?"

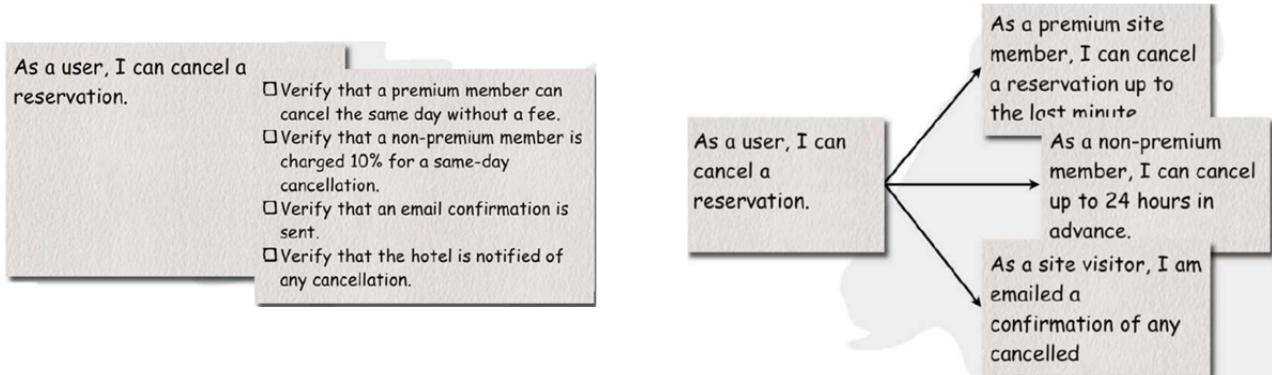
"Story-Writin" Workshops

- Kunde, Entwicklung sowie andere Beteiligte
- Brainstorming, um User Stories zu generieren
- Ziel: So viele wie möglich ohne priorisierung
- Manche sind "implementation ready" andere "Epics"

7.3 Format einer User Story

Eine User Story ist eine knappe und präzise Beschreibung eines Stücks Funktionalität, welche dem User der Software einen Nutzen stiftet. User Story Karten haben 3 Teile:

- **Card:** Eine schriftliche Beschreibung der User Story für Planungszwecke und als Erinnerung
- **Conversation:** Ein Bereich, der weitere Informationen über die User Story und Abstimmungsdetails enthält
- **Confirmation:** Ein Bereich, der die Tests enthält, die sicherstellen, dass die User Story vollständig ist und so wie erwartet abläuft



User Stories müssen nicht allumfassende und vollständige Anforderungen darstellen, sie sind mit schriftlicher Dokumentation anzureichern, falls notwendig. Anreichern mit:

- Geschäftsregeln
- Glossar
- Beispiele für Input und erwarteter Output
- Abläufe
- ...

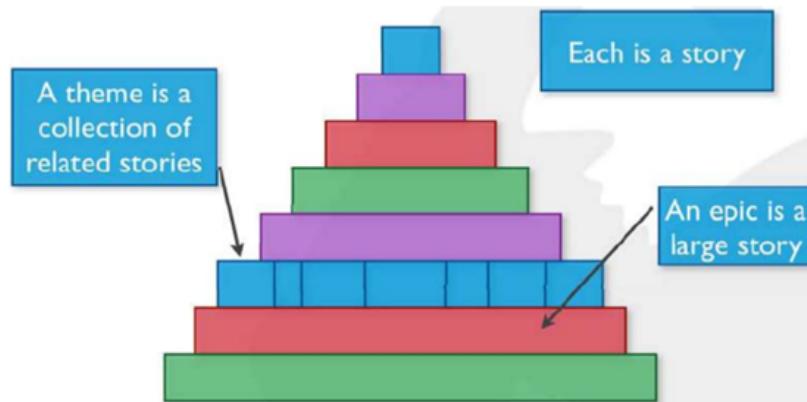
7.3.1 User Rollen

Verschiedene User Rollen erweitern den Umfang, da man nicht von einem einzigen User ausgeht. Dadurch wird es möglich, User zu variieren, beispielsweise durch:

- Hintergrund
- Vertrautheit im Umgang
- Nutzungs frequenz
- Hauptsächlich verwendetes Zielgerät
- Nutzungszweck

7.4 Epic, Themes und Stories

Wie im unteren Bild zu sehen ist, sind Epics grosse User Stories. Themes sind Sammlungen von zusammenhängenden User Stories. User Stories sind oben (7.3) beschrieben. Jedoch sind alle Varianten eine Story.



7.5 Eigenschaften einer guten User Story

Gute User Stories weisen 6 Eigenschaften auf:

- **Independent:** Abhängigkeiten vermeiden
- **Negotiable:** Verbindlichkeit zwischen User und Development
- **Valuable:** Für den Kunden oder die User
- **Estimateable:** Für die Umsetzung zentral
- **Small:** Normalerweise ein Satz
- **Testable:** Durch einen Testfall geprüft

INVEST steht für diese Eigenschaften.

7.5.1 Independent / Unabhängig

Abhängigkeiten führen zu Problemen in Bezug auf die Priorisierung und Schätzung.

→ Im Idealfall kann eine User Story für die Umsetzung ausgewählt werden, ohne dass eine Vielzahl anderer User Stories mitberücksichtigt werden muss.

Beispiel von **abhängiger** User Stories:

Als Firma kann für ein Inserat mit einer Masterkarte bezahlt werden

Als Firma kann für ein Inserat mit einer American Express Karte bezahlt werden

Als Firma kann für ein Inserat mit einer Visa Karte bezahlt werden

7.5.2 Negotiable / Verhandelbarkeit

- User Stories sind keine Verträge, aber sie sind zentrale Artefakte für die Entwicklung und für den Kunden.
- Flexibilität soll also berücksichtigt werden (sonst weglassen)
- Unnötige Details oder Präzisionen weglassen
- Details können im Rahmen von Tests niedergeschrieben werden

Beispiel Tests:

- Test mit Visa, Mastercard und American Express
- Test mir richtigen, falschen und fehlender Nummer
- Test mit abgelaufener Karte
- Test mit > 100 CHF und < 100 CHF

7.5.3 Valuable / Werthaltigkeit

- User Stories sollten für den Kunden von Wert sein und nicht für die Entwicklung
- User Stories, die aus der Entwicklung kommen, sollten umgeschrieben werden, um die Werthaltigkeit für den Kunden zu reflektieren

Vermeiden

Alle Verbindungen zur DB müssen über einen Connection Pool gehandelt werden

Errorhandling & Logging hat über eine separate Klasse zu erfolgen

Ersatz

Bis zu 50 Nutzer sollten mit einer 5 User-Lizenz der DB arbeiten können

Fehler werden einheitlich angezeigt dargestellt und konsistent abgespeichert

7.5.4 Estimatable / Schätzbarkeit

- Da die Planung auf User Stories basiert, sollten sie unbedingt schätzbar sein
- Warum können User Stories schwierig zu schätzen sein?

Gründe	Lösungen
Der Entwicklung fehlt das Fachwissen	Klärung mit dem Kunden
Der Entwicklung fehlt das technische Know-How	Spike!
Die User Story ist zu umfangreich	User Story aufteilen!

7.5.5 Small / Klein

- User Stories sollten nicht so umfangreich sein, dass es schwierig wird, deren Umsetzung zu planen, zuzuordnen oder zu priorisieren
- Sind die User Stories klein, so ist es sehr einfacher zu skalieren

7.5.6 Testable / Testbar

- User Stories müssen testbar sein
- Erfolgreich getestet ist dasselbe wie erfolgreich entwickelt
- Test sollten automatisiert werden

8 Estimation and Planning

Lernziel

Sie kennen die folgenden Begriffe und können diese erklären: User Stories, User Roles , Epics, Themes, Story Points, Velocity, Planning Poker, Conditions of Satisfactions, Levels of planning, Product Backlog, Priorisierung (der User Stories), Techniques for Estimating, Relative Schätzung.

Sie kennen die folgenden Begriffe und können diese erklären: Planning for value, Cost, Financial value, Risk, New Knowledge, Kano-Model of Customer Satisfaction. Sie kennen die Planungsebenen im agilen Kontext und können die Zuständigkeiten beschreiben.

8.1 Begriffserklärungen

8.1.1 User Stories

Wird im Kapitel **User Stories** beschrieben.

8.1.2 User Roles

Wird im Kapitel **User Rollen** beschrieben.

8.1.3 Epics

Wird im Kapitel **Epic, Themes und Stories** beschrieben.

8.1.4 Themes

Wird im Kapitel **Epic, Themes und Stories** beschrieben.

8.1.5 Story Points

Was sind Story Points?

- Die Anzahl der Story Points bestimmen die Grösse der Story
- Es gibt keine definierte Formel, um die Grösse einer Story zu definieren
- Schätzen den Aufwand, der notwendig ist, um ein bestimmtes Feature zu realisieren
- Sind eine relative Messung der Komplexität einer User Story

8.1.6 Velocity

Was ist Velocity?

- Ist ein Mass für die Fortschrittsrate eines Teams

- Die Velocity wird berechnet, indem die Anzahl der Story-Points, die jeder User Story zugewiesen wurden und die das Team während der Iteration abgeschlossen hat, addiert wird.
- Es wird angenommen, dass das Team pro Iteration eine ähnliche Anzahl von Story-Points abschliessen wird

8.1.7 Planning Poker

Planning Poker ist ein "Spiel", um die User Stories iterativ zu schätzen.

- Jeder Schätzer erhält einen Kartenstapel, auf jeder Karte steht eine gültige Schätzung
- Der Kunde/Produkteigentümer liest eine Geschichte vor und bespricht sie kurz
- Jeder Schätzer wählt eine Karte aus, die seine Schätzung darstellt und legt sie verkehrt auf den Tisch
- Die Karten werden umgedreht, damit alle sie sehen können
- Besprechung der Unterschiede (vor allem bei Ausreisern)
- Neuschätzung bis zur Übereinstimmung der Schätzung

8.1.8 Conditions of Satisfactions

Definition

- Vor dem Planungsstart müssen die Kriterien bekannt sein, die bewerten, ob das Projekt erfolgreich ist oder nicht
- Für die meisten Projekte ist der Business Case relevant (→ wieviel Geld wird gespart / zusätzlich generiert)
- **Indikatoren sind normalerweise:** Zeitplan, Scope und Ressourcen
 - Aus Sicht des Product Owner sind die Conditions of Satisfaction durch eine Kombination der Indikatoren gegeben
 - Ein datumsgetriebenes (Date-Driven) Projekt ist eines, welches Releases zu fixen Zeitpunkten jedoch bei variabler Funktionalität liefert
 - Ein funktionsetriebenes (Feature-Driven) Projekt ist eines, welches die Funktionalität höher bewertet als den Zeitplan

8.1.9 Levels of planning

Die Planung wird zu einem Prozess der Festlegung und Überarbeitung von Zielen, die zu einem längerfristigen Ziel führen.



Agile teams
plan at the
innermost three
levels

8.1.10 Product Backlog

- Anforderungen
- Liste aller auszuführenden Arbeiten im Projekt
- Idealerweise so dargestellt, dass jeweils der Wertbeitrag für den Nutzenden des Produktes nachgewiesen wird
- Priorisiert
- Re-Priorisiert am Anfang jedes Sprints

8.1.11 Priorisierung

Welche Features sollten entwickelt werden?

Die Faktoren für die Priorisierung der User Stories sind:

- Der finanzielle Wert eines Features
- Die Kosten für die Entwicklung (und den Unterhalt) des neuen Features
- Der Umfang und die Signifikanz des Know-How-Gewinns durch die Entwicklung des Features
- Die Höhe des Risikos, das durch die Entwicklung des Features beseitigt wurde

8.1.12 Techniques for Estimating

Schätzungen werden am besten in Zusammenarbeit mit dem Team erstellt. → **Planning Poker**

Gemäss Studien kann man Dinge am besten einschätzen, wenn sie innerhalb einer Größenordnung liegen.
Mögliche Größenordnungen:

- **Fibonacci:** 1, 2, 3, 5 und 8
- 2^n : 1, 2, 4 und 8,

8.1.13 Relative Schätzung

Story Points sind eine relative Messung der Komplexität einer User Story. Relative Schätzgrößen sind besser als absolute Schätzgrößen.

8.1.14 Planning for value

Gleichtes wie bei **Priorisierung**

8.1.15 Cost

- Die Kosten für die Umsetzung eines bestimmten Features ist die wichtigste Determinante für die Priorisierung
- Viele Features scheinen wunderbar zu sein, bis wir ihre Kosten erfahren
- Oftmals wird übersehen, dass sich diese Kosten über die Zeit verändern können
- Machen Sie eine grobe Umrechnung von Story Points oder idealen Tagen in Geld

8.1.16 Financial value

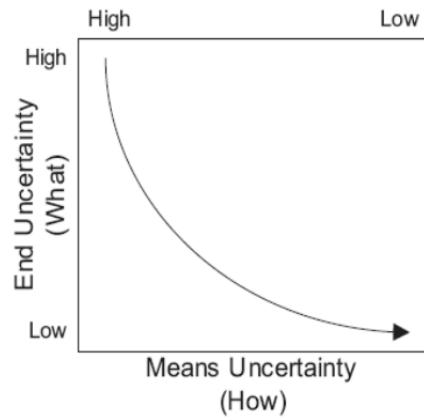
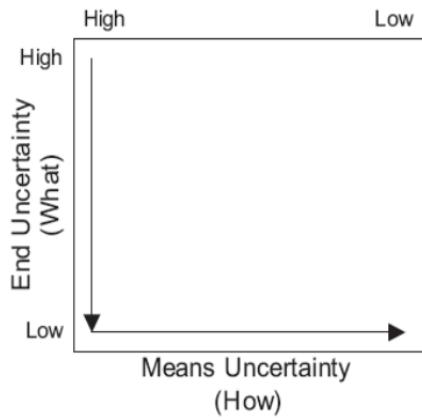
- Priorisiere den geschäftlichen Nutzen: Wie viel Geld wird das Unternehmen durch das neue Feature verdienen oder einsparen?
- Einnahmequellen: Neue, zusätzliche und verbleibende Einnahmen, betriebliche Effizienzsteigerung
- Der Zeitwert des Geldes
- Nettogegenwartswert: $NPV(i) = \sum_{t=0}^n F_t(1+i)^{-t}$
- Interner Zinssatz: $0 = PC(i*) = \sum_{t=0}^n F_t(1+i)^{-t}$

8.1.17 Risk

Gleichtes wie bei **Risiko**

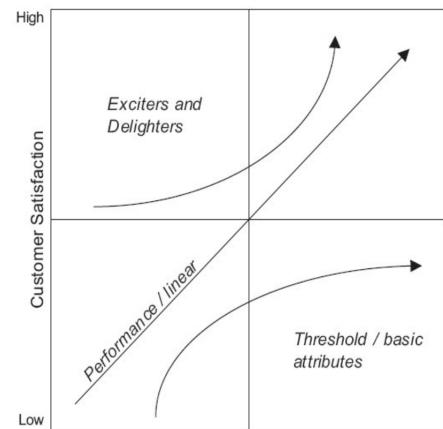
8.1.18 New Knowledge

- Bei vielen Projekten entfällt ein Grossteil der Gesamtarbeit auf die Streben nach neuem Wissen
- Es ist wichtig, dass dieser Aufwand anerkannt und als grundlegend für das Projekt angesehen wird: Zu Beginn eines Projekts wissen wir nie alles, was wir am Ende des Projekts wissen müssen
 - Wissen über das Produkt (was entwickelt werden soll)
 - Wissen über das Projekt (wie das Produkt erstellt wird)
- Die Kehrseite des Wissenserwerbs ist die Verringerung der Unsicherheit



8.1.19 Kano-Model of Customer Satisfaction

- **Must-have Features**
- **Linear Features** (Je mehr desto besser)
- **Exciters and Delighters** (Hohe Kundenzufriedenheit → Preisaufschlag)



8.2 Planungsebenen im agilen Kontext

Wird im Kapitel [Levels of planning](#) beschrieben.

9 Build Automation, CI, CD, DevOps

Lernziel

Erklären Sie die Arten von Software Automation (on-demand, scheduled, triggered), Arten von Automation und Ziele. Erklären Sie Software Automation Pipeline mit den einzelnen Schritten, sowie deren Unterschiede.

9.1 Arten von Software Automation

Eine Vielzahl von Aufgaben einer Softwareentwicklung können automatisiert werden.

- **On-Demand:** Ein Skript ausführen oder eine Taste drücken
- **Scheduled:** Zu bestimmten Zeiten → nächtliche Builds
- **Triggered:** Bei bestimmten Ereignissen → commit/push

9.2 Typen der Automatisierung

Build Automation

- Kompilieren
- Software packaging
- Erstellen von Dokumenten und/oder Release Notes

Test Automation

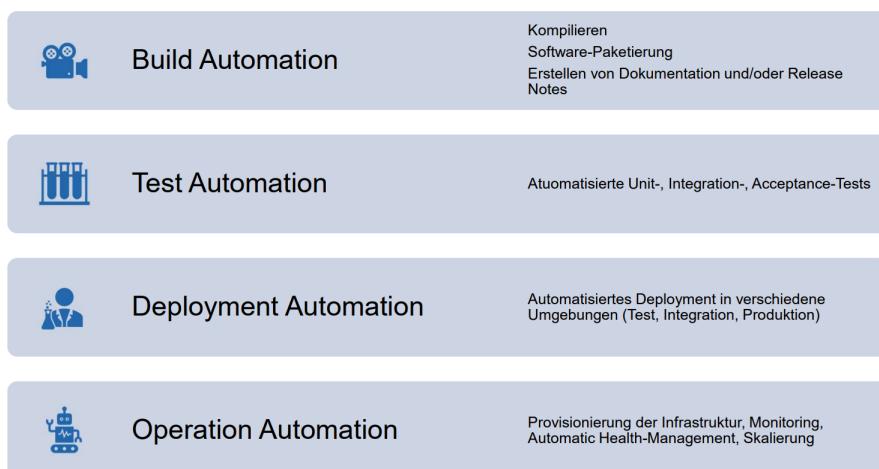
- Automatisierte Unit-, Integration-, Acceptance-Tests

Deployment Automation

- Automatisiertes Deployment in verschiedene Umgebungen (Test, Integration, Produktion)

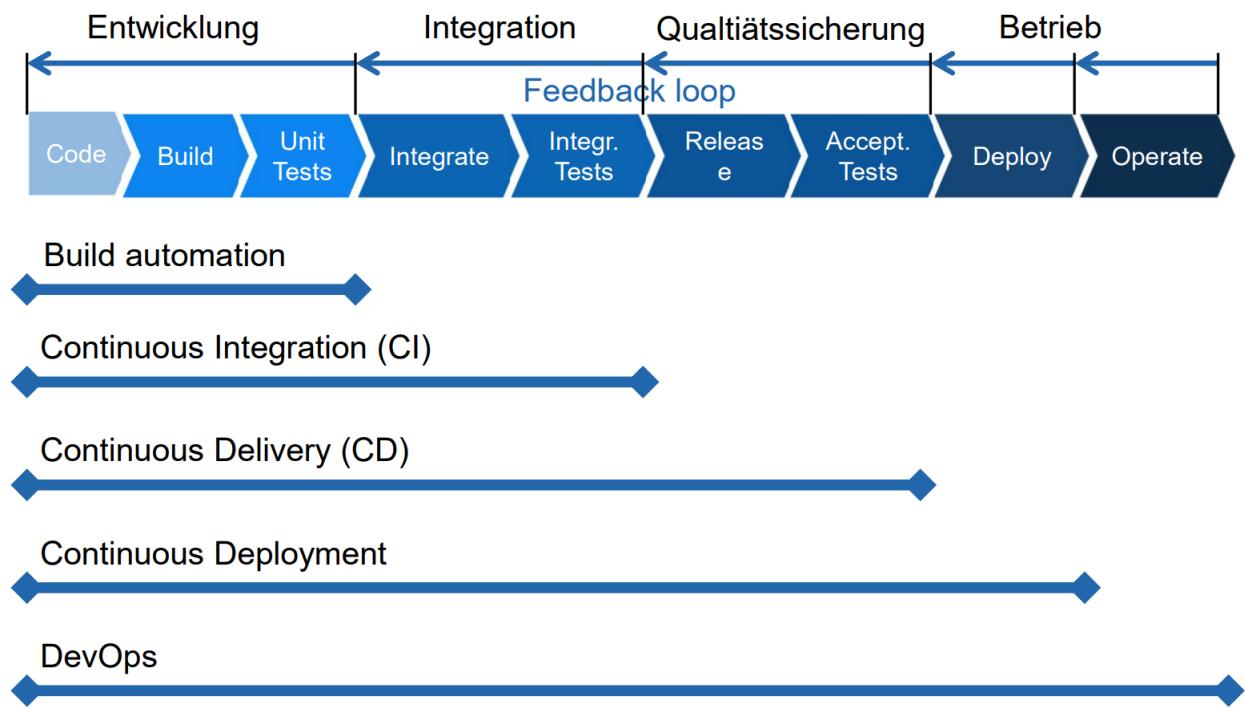
Operation Automation

- Provisionierung der Infrastruktur, Monitoring, Automatic Health-Management, Skalierung



9.3 Software Automation Pipeline

- Automatisierung kann in jeder Phase des Softwareprozesses eingesetzt werden, von der Entwicklung über die Integration und QS bis hin zum Betrieb.
- Jeder Schritt kann nur fortgesetzt werden, wenn er mehrere Tests erfolgreich durchläuft
- Andernfalls werden die verantwortlichen Personen über den Fehler informiert und der Prozess wird bis zur Behebung angehalten → Feedback loop (im Bild)



Build Automation

Wird im Kapitel [Arten von Software Automation](#) beschrieben

Continuous Integration

Wird im Kapitel [Continuous integration](#) beschrieben

Continuous Delivery

Erzeugt Releases, deployt in die Staging-Umgebung und führt automatische Akzeptanztests aus (Stresstests, Lasttests, Compliance-Tests, ...). Danach ist der Build bereit für die Produktion, aber das Deployment erfordert noch einen manuellen Schritt.

Continuous Deployment

Automatisches Deploymen in die Produktion nach erfolgreich bestandenen Akzeptanztests.

DevOps

Automatischer Betrieb des Produktionssystems (Konfigurationsmanagement, Infrastrukturbereitstellung, Backup, Monitoring, automatisches Health Management, Skalierung, ...)

10 Scrum

Lernziel

Sie kennen die Charakteristiken von SCRUM inkl. allen «Roles resp. Scrum team», «Ceremonies resp. Scrum Events», «Scrum Artifacts», sowie «Scrum Values» und können diese erklären. Sie kennen die folgenden Begriffe und können diese erklären: Sprint, Sprint goal, Retrospective, Task board, Burndown chart, Definition of Done, Definition of Ready, Daily Scrum, Increment.

Sie haben den SCRUM Guide gelesen: [Scrum Guide](#)

10.1 Scrum Charakteristiken

- Selbstorganisierte Teams
- Produktfortschritt als Serie von "Sprints" in Monats-Rhythmus
- Anforderungen werden als Artefakte in einem "Backlog" erfasst
- Keine speziellen Engineering-Vorgehen definiert
- Nutzt sich entwickelnde Regeln zur Schaffung eines agilen Umfelds für die Durchführung von Projekten
- Einer der agilen Prozesse

10.2 Scrum Rollen

Es gibt verschiedene Rollen bei Scrum

Product Owner

- Definiert die Features des Produkts
- Entscheidet über das Datum und den Inhalt eines Releases
- Ist Verantwortlich für die Profitabilität des Produkts
- Priorisiert Features aufgrund ihres Marktwertes
- Passt Features und Prioritäten wenn nötig für jeden Sprint an
- Akzeptiert Arbeitsresultate oder weist sie zurück

Scrum Master

- Verantwortlich für die Durchsetzung der Scrum-Werte und Praktiken
- Entfernt Hindernisse
- Sorgt dafür, dass das Team vollständig arbeitsfähig und produktiv ist
- Ermöglicht enge Zusammenarbeit zwischen allen Beteiligten
- Schirmt das Team gegen äußere Einflüsse ab

Aufgaben eines Scrum Masters:

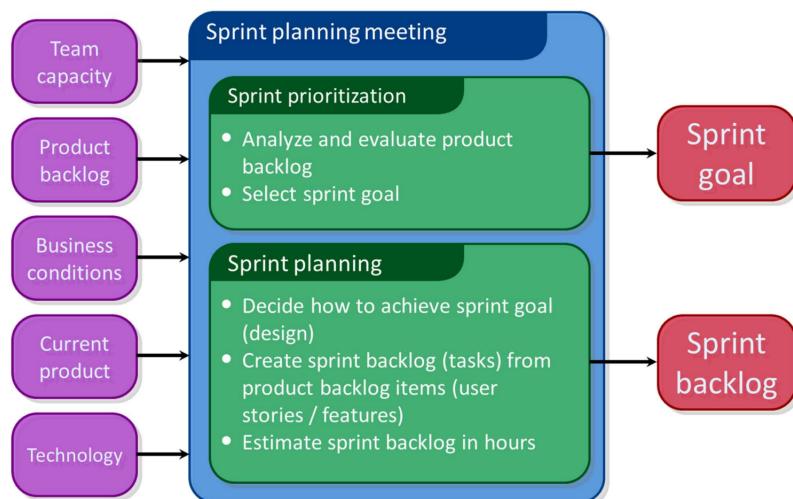
- Keeps Scrum process running
- Ensures a proper power balance between Project Owner, Team, Management
- Protects the Team
- Moderates in the Team
- Helps to organize
- Helps to keep the Team focused on the current Sprint
- Helps to achieve Sprint goals
- Works with Project Owner
- Educates Project Owner, Team, Management and Organization
- Solves impediments
- Encourages and helps to achieve transparency
- Strives to develop a Team into a High Performance Team
- Encourages and protects self-organization
- Educates and focuses a Team toward business-driven development
- Supports Team building and Team development by utilizing the abilities and skills of individuals, and fostering a Feedback culture
- Helps to self-help
- Ensures and supports Empowerment of the Team
- Addresses needs efficiently and effectively
- Detect hidden problems and strives to solve them
- Helps Team to learn from its experiences

Team

- Typischerweise 5–9 Personen
- Cross-Functional (Programmieren, Tester, Designer, etc...)
- Sollten zu 100% im Team sein
- Self-Organized
- Zusammensetzung sollte nur zwischen den Sprints ändern

10.3 Scrum Ceremonies / Events

Sprint planning



Das Sprint Goal ist ein kurzes Statement zur Zielformulierung

- Team wählt Items, die sie innerhalb eines Sprints umsetzen können, aus dem Backlog aus
- Ein Sprint Backlog wird definiert
- Identifikation und Schätzung der Tasks
- Gemeinsam, nicht alleine durch den Scrum Master
- High-Level Design überlegen

Sprint review

- Team präsentiert die Resultate des Sprints
- Als Demo
- Keine Folien
- Alle nehmen teil
- Alle Einladen

Sprint retrospective

- Periodische Prüfung was läuft und was nicht
- Typischerweise 15–30 Minute
- Nach jedem Sprint
- Alle nehmen Teil (Scrum Master, Product Owner, Team, Kunden)

In einer Retrospektive überlegt das ganze Team, was sie gerne Starten, Stoppen und weitermachen möchten.

Daily scrum meeting

- Parameter
 - Täglich
 - 15 Minuten
 - Stand-Up
- Keine Problemlösungen
- Alle Beteiligten sind eingeladen
- Nur die Team-Mitglieder, der Scrum Master und der Product Owner dürfen sprechen
- Sollte unnötige Meetings vermeiden helfen

Jeder Antwortet auf diese drei Fragen

1. Was hast du gestern gemacht?
2. Was willst du heute tun?
3. Ist alles auf gutem Weg?

10.4 Scrum Artifacts

Product backlog

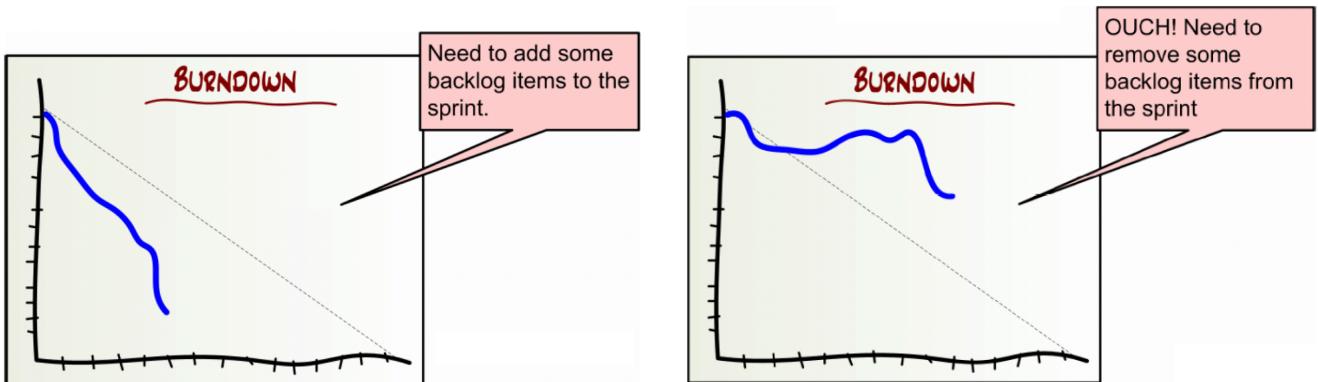
Wird im Kapitel [Product Backlog](#) beschrieben.

Sprint backlog

- Individuen wählen die Arbeit selbst aus
- Die Schätzung des noch verbleibenden Aufwandes erfolgt täglich
- Jedes Team-Mitglied kann den Sprint Backlog anpassen
- Die Arbeit des Sprints entsteht
- Falls die Arbeit unklar ist sollte ein entsprechend grösseres Backlog Item für die spätere Klärung geschaffen werden

Sprint burndown charts

Durch den Burndown chart sieht man, wie man beim aktuellen Sprint unterwegs ist.



10.5 Scrum values

11 SAFe (Scaled Agile Framework)

Lernziel

Sie kennen die Werte, Prinzipien und Ebenen von SAFe und wissen, was ein Release Train ist und welches die spezifischen SAFe Rollen sind.

11.1 Werte von SAFe

Alignment

Die Mission wird kommuniziert, indem die Portfoliostrategie und die Lösungsvision festgelegt wird und die Wertschöpfung während der Planung bestimmt und angepasst wird, um sicherzustellen, dass die Nachfrage mit der Kapazität übereinstimmt.

Built-in quality

Es wird eine Umgebung geschaffen, in der integrierte Qualität zum Standard wird.

Transparency

Förderung der Visualisierung aller relevanten Arbeiten und schaffen Umfeld, in dem "...die Fakten immer freundlich sind, jedes bisschen Beweis, das man in irgendeinem Bereich erlangen kann, führt einen so viel näher an das, was wahr ist"

Program execution

Führungskräfte nehmen als Business Owner an der PI-Planung und -Ausführung teil.

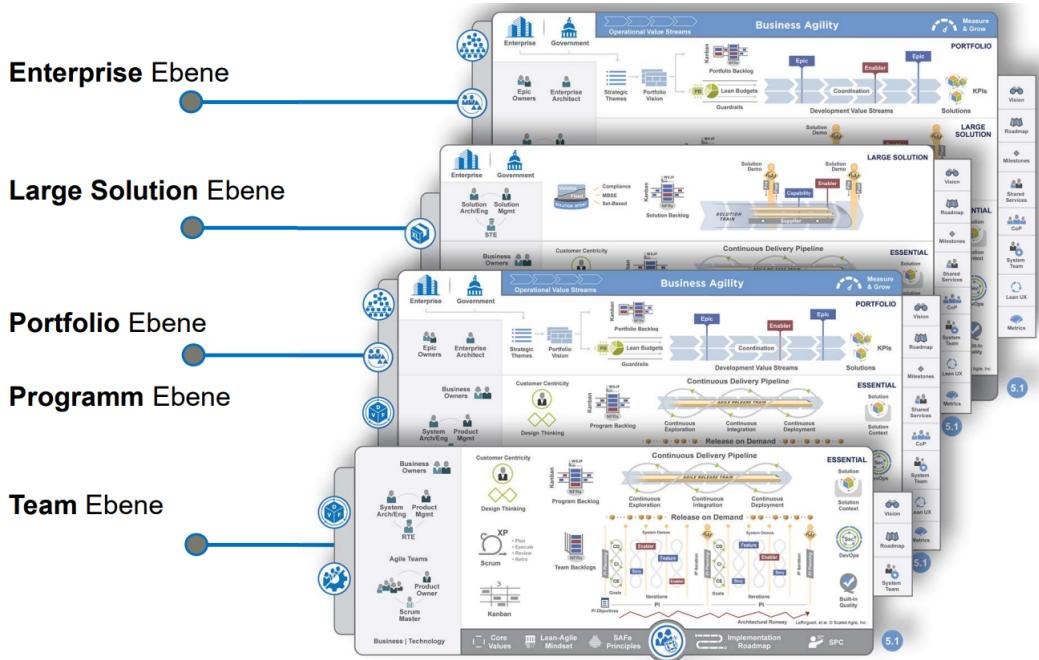
11.2 Prinzipien von SAFe

1. Nehmen Sie eine wirtschaftliche Sicht ein
2. Wenden Sie Systemdenken an
3. Nehmen Sie Variabilität an, halten Sie Möglichkeiten offen
4. Machen Sie inkrementell mit schnellen, integrierten Lernzyklen Fortschritte
5. Bauen Sie Meilensteine auf objektiven Bewertungen von funktionierenden Systemen auf
6. Visualisieren und begrenzen Sie die Produktionszeit, reduzieren Sie die Batch-Size und halten Sie die Wartezeiten kurz
7. Etablieren Sie einen Arbeitsrhythmus und synchronisieren Sie diesen durch bereichsübergreifende Planung
8. Setzen Sie die Motivation frei, die Ihren Wissensableitern innewohnt
9. Dezentralisieren Sie die Entscheidungsfindung
10. Organisieren Sie sich rund um die Wertschöpfung

11.3 Ebenen von SAFe

- Enterprise Ebene
- Large Solution Ebene
- Portfolio Ebene

- Programm Ebene
- Team Ebene



11.4 Release Train

Agile Teams arbeiten im Kontext eines SAFe Agile Release Train (ART), einem langlebigen Team von Agilen Teams, das eine gemeinsame Vision und Richtung vorgibt und letztendlich für die Bereitstellung von Lösungsergebnissen verantwortlich ist.

11.5 SAFe Rollen

Der **Product Manager** verantwortet das übergeordnete Product Backlog, das dann Programm Backlog heißt.

Der **System Architect** ist für die Systemgestaltung zuständig

Der **Release Train Engineer** kümmert sich als eine Art teamübergreifender Scrum Master für die Zusammenarbeit und Abhängigkeiten zwischen Teams.

Der **Business Owner** behält den Blick auf ROI und Wertmaximierung sowie der Ausbalancierung von Kosten und Nutzen in der Produktentwicklung auf Systemebene, somit entlastet er den Product Manager etwas in dieser Hinsicht.

UND die **Scrum Rollen (Product Owner, Scrum Master, Team)**

12 Software Architecture

Lernziel

Sie kennen die folgenden Begriffe und können diese erklären: Software Architektur, Software Architekt, «allgemeine Systemeigenschaften», Moduleigenschaften, Schwierigkeiten des Software Designs Architektur Style: Sie kennen die Vor- und Nachteile aller behandelten Architektur-Stile und können diese erklären. Sie kennen die drei grossen Pattern (Transactions Script, Domain Model und Table Module) und deren Anwendungsgebiet und Sie verstehen die 4 Level des C4 Modells.

Hinweis: Die Bücher «Software Architecture for Developers: Volume 1 & 2» müssen sie nicht im Detail gelesen haben. Das C4-Modell wird vorausgesetzt.

12.1 Software Architektur

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.

Es gibt mehrere Typen von Architektur:

- Infrastructure
- Security
- Technical
- Solution
- Network
- Data
- Hardware
- Enterprise
- Application
- System
- Integration
- IT
- Database
- Information
- Process
- Business
- Software

Application Architecture

Bei der Anwendungsarchitektur geht es darum, ins Innere der Anwendung zu schauen, um zu verstehen, wie sie entworfen und aufgebaut ist. Zentral ist dabei die Fragestellung, wie die Anwendung in einzelne Building-Blocks zerlegt werden kann:

- Komponenten
- Schichten
- Packages
- Namespace

→ Es geht um die Organisation des Codes

System Architecture

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

12.1.1 Vorteile guter Software Architektur

- Kommunikationsbasis zwischen den an einem Softwareprojekt beteiligten Personen ist die treibende Kraft des Systemdesigns
- Setzt den Rahmen für die Wiederverwendung der einzelnen Software-Artefakte. "Träger" von Qualitätsmerkmalen und nichtfunktionalen Eigenschaften
- Unveränderlich über mehrere Softwareprojekte mit minimal unterschiedlichen Systemanforderungen, was den Entwicklungsprozess für eine Reihe ähnlicher Projekte vereinfacht
- Basis für eine umfassende Betrachtung einer Software und erlaubt die Analyse bestimmter Systemeigenschaften bereits zu einem sehr frühen Entwicklungszeitpunkt des Systems

12.2 Software Architekt

Wir sehen als wesentliches Eigenschaft guter Architekten an, dass er die unter den gegebenen Umständen bestmöglichen Systeme konstruiert und deren Entwicklung begleitet. Systeme, die verständlich, langlebig, wartbar, funktional, performant und sicher sind. Systeme, die robust auf Fehler reagieren und ihre jeweiligen Stakeholder positiv überraschen, statt sie zu verärgern. Kurzum: Gute Architekten liefern gute Qualität.

12.2.1 Die Aufgabe des Architekten

- **Entscheiden**
Unsicherheit als Antrieb nutzen, architektonische Kompromisse
 - **Dokumentieren**
Kommunikation ist King, klarheit und Führung ihre Diener.
Halten Sie Ihre Begründung fest
 - **Machbarkeit prüfen**
Eine Zeile funktionierenden Codes ist 500 Zeilen Spezifikation wert.
Probieren Sie, bevor Sie wählen.
 - **Programmieren**
Architekten müssen praktisch tätig sein.
Ein Architekt ist vor allem ein Entwickler.
Wenn man es entwirft, sollte man es auch programmieren können.
Ein Architekt implementiert immer/auch
 - **Kommunizieren**
Stand up, Klartext reden.
Lernen Sie eine neue Sprache.
 - **Verhandeln**
Suchen Sie den Wert der gewünschten Fähigkeiten.
- Sie verhandeln mehr, als Sie denken.
 - **Vereinfachen**
Wesentliche Komplexität vereinfachen, zufällige Komplexität reduzieren.
Vereinfachung vor Allgemeinheit, Verwendung vor Wiederverwendung.
Sicherstellen, dass die einfachen Dinge einfach sind.
 - **Vereinheitlichen**
Verringerung der Entropie.
 - **Zuhören**
Die Bedenken der Stakeholder anhören.
 - **Beobachten**
Kontrolliere nicht, sondern beobachte.
Get the 1000-foot view. → Beobachten aus entfernung
 - **Denken (über die Zukunft)**
Alles wird letztendlich scheitern.
Konzentrieren Sie sich auf Anwendungssupport und Wartung.
Ihr System ist veraltet, entwickeln Sie es entsprechend.
Sie können keine zukunftssicheren Lösungen

- anbieten.
 - **Führen**
- Geben Sie den Entwicklern Unabhängigkeit.
Es gibt kein "Ich" in der Architektur.

12.3 Allgemeine Systemeigenschaften

Warum ist der Bau von Software so schwierig?

1. Komplexität
2. Konformität
3. Formbarkeit
4. Unsichtbarkeit

12.3.1 Komplexität

Software ist im Verhältnis zu seiner Grösse wahrscheinlich das Komplexeste, was von Menschenhand überhaupt erschaffen wird. Wird durch Abstraktion ein vereinfachtes Modell eines komplexen Phänomens erzeugt, so ist für das Modell wesentlich, dass in ihm die wichtigsten Eigenschaften des Phänomens abgebildet werden können. Die anderen Eigenschaften können weggelassen werden. Dies ist jedoch bei Software nicht der Fall. Es können keine Eigenschaften weggelassen werden, um die Komplexität in einfachere Modelle abzubilden.

12.3.2 Konformität

Das Umfeld eines Systems, also die Schnittstellen des Systems zu anderen Systemen variieren in dem masse, wie die Komplexität dieser umgebenden Systeme variiert.

12.3.3 Formbarkeit

Software ist einem ständigen Wandel unterworfen. Da Software so leicht geändert werden kann, werden auch sehr viele Änderungen vorgenommen.

12.3.4 Unsichtbarkeit

Software ist unsichtbar und kann nicht direkt visualisiert werden. Es gibt keine geometrische Darstellung von Software. Die Visualisierung von Software erfodert verschiedene Ansichten.

12.4 Architektur Stile

Independent Components: Unabhängig ablaufende Elemente, die über Nachrichten miteinander interagieren

Call-and-Return: Definiert durch einen fixen Kommunikations-Mechanismus zwischen aufrufendem und aufgerufenen Element

Virtual Machine: Erlaubt die Realisierung portabler und interpretierbarer Systeme

Data Flow: System ist eine Abfolge von Datenbezogenen Transformationen

Date Centered: Zentrale Aufgabe ist Zugriff und Aktualisierung von Daten eines Repositories.

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	Expertenmodelle	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 Kontrollsystme	Skalierbar	Anwendung eingeschränkt

12.5 Software Pattern

12.5.1 Transactions Script

Das Transaktionsskript-Pattern organisiert und unterteilt die Geschäftslogik in einzelne Prozeduren, so dass jede Prozedur eine einzelne Anfrage der Darstellungslayer abdeckt.

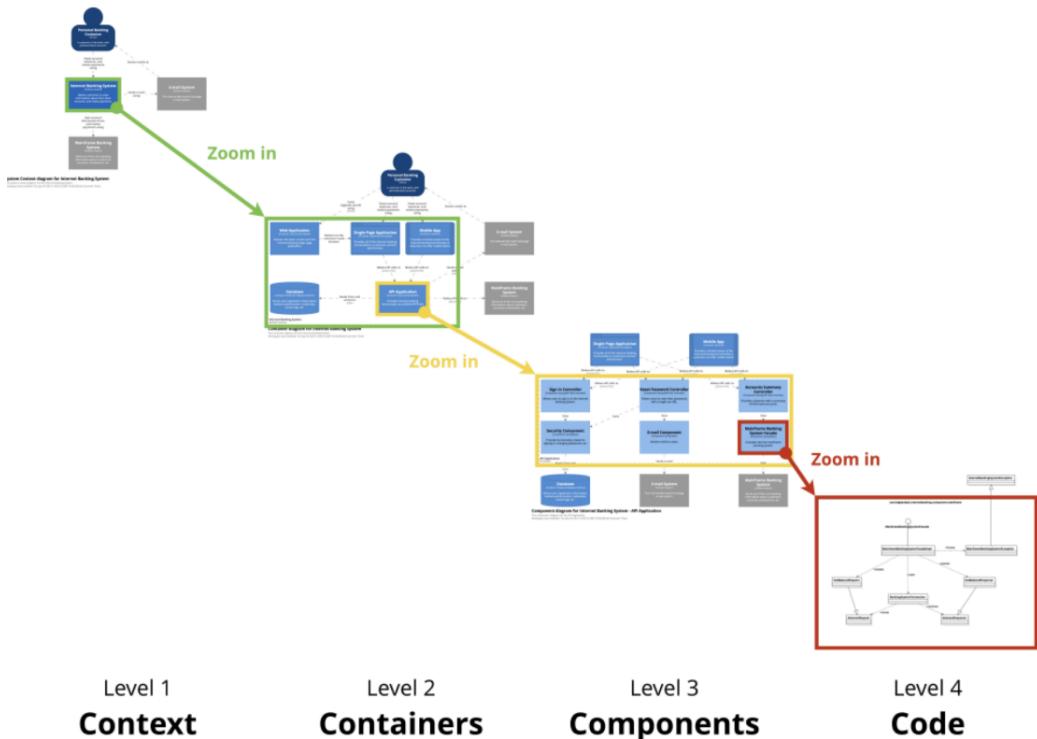
12.5.2 Domain Model

Das Domänenmodell-Pattern beschreibt ein Objektmodell der Problemdomäne, das Verhalten und Daten enthält.

12.5.3 Table Module

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

12.6 C4 Modell



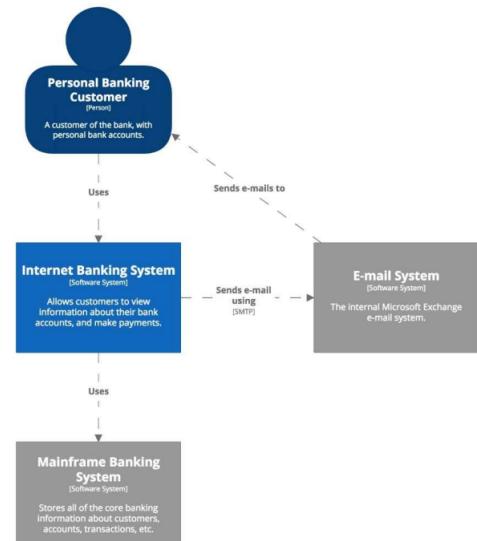
Das C4-Modell besteht aus 4 Leveln

12.6.1 Level 1: System Context diagram

Ein Systemkontextdiagramm kann ein nützlicher Ausgangspunkt für das Zeichnen und Dokumentieren eines Softwaresystems sein, der es ermöglicht, einen Schritt zurückzutreten und das Gesamtbild zu betrachten.

Ein Systemkontextdiagramm hilft, folgende Fragen zu Beantworten:

1. Wo ist das Software-System, das gebaut wird?
2. Von wem wird es verwendet?
3. Wie passt es in die bestehende Umgebung?



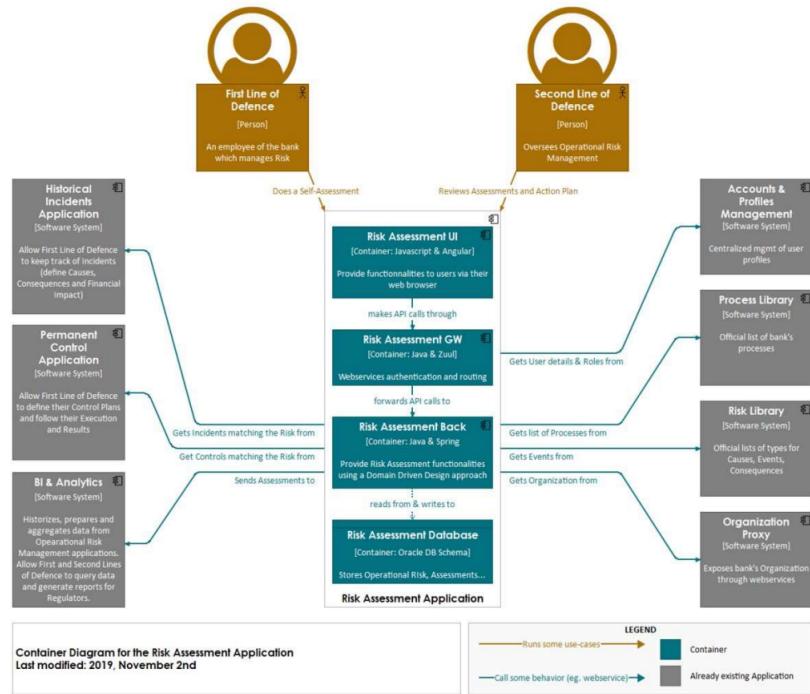
12.6.2 Level 2: Container diagram

Das Container-Diagramm zeigt die High-Level-Form der Software-Architektur und wie Verantwortlichkeiten darin verteilt sind.

Ein Container-Diagramm hilft, folgende Fragen zu beantworten:

1. Wie ist die Gesamtform des Softwaresystems?

2. Was sind die High-Level-Technologien entscheidungen?
3. Wie sind die Verantwortlichkeiten im System verteilt?
4. Wie kommunizieren die Container miteinander?
5. Wo muss der Entwickler Code schreiben, um Features zu implementieren?

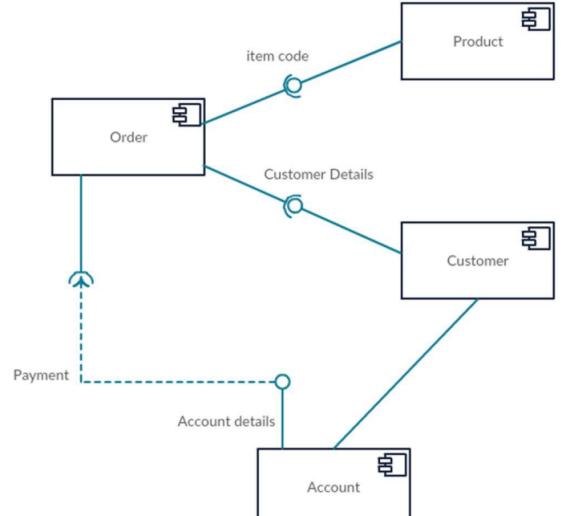


12.6.3 Level 3: Components diagram

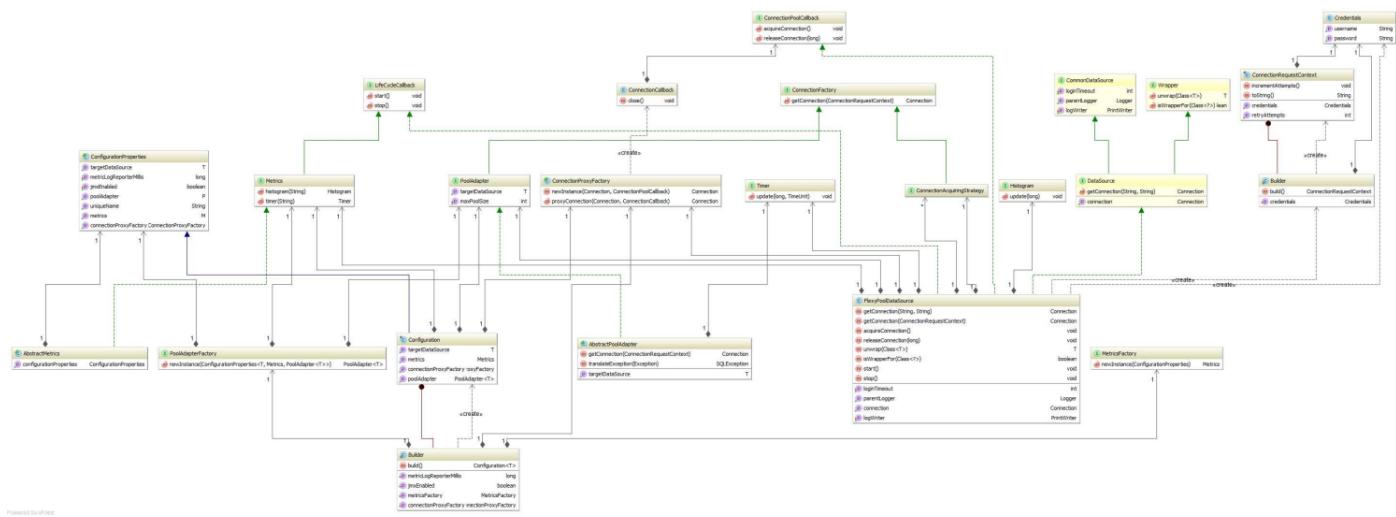
Das Component-Diagramm zeigt den Aufbau der einzelnen Elementen des Container-Diagramms.

Ein Komponenten-Diagramm hilft, die folgenden Fragen zu beantworten:

1. Aus welchen Komponenten setzt sich jeder Container zusammen?
2. Haben alle Komponenten ein Zuhause (d. h. sie befinden sich in einem Container)?
3. Ist es klar, wie die Software auf hoher Ebene funktioniert?



12.6.4 Level 4: Code-level diagrams



13 SOLID Principles

Lernziel

Sie kennen alle «SOLID Principles» und deren «Anwendungsfälle» und können weitere Laws wie «Conway's Law», «Brooks's Law», «Parkinson's Law», «Pareto's Fallacy», «Sturgeon's Revelation», «The Iceberg Fallacy», «The Peter Principle», «Eagleson's Law», sowie «Greenspun's 10th Rule of Programming», erklären.

Links zu den Power-Points der SOLID Vorträge

Alle [SOLID-Vorträge](#)

ACHTUNG: An SEP darf kein Internet verwendet werden → SOLID-Unterlagen downloaden

Single responsibility principle ([SRP](#))

Open closed principle ([OCP](#))

Liskov substitution principle und Interface segregation principle ([LSP&ISP](#))

Dependency Inversion principle ([DIP](#))

Das PDF mit den genauen Beschreibungen der Prinzipien sind [hier](#) zu finden.

Die Prinzipien sind auf folgenden Seiten des PDFs:

- SRP: Ab Seite 44 (61)
- OCP: Ab Seite 52 (69)
- LSP: Ab Seite 60 (77)
- ISP: Ab Seite 66 (83)
- DIP: Ab Seite 70 (87)

Zu den Gesetzten

Es gibt ein weiteres PDF (Laws.pdf).

14 Cynefin Framework / Codefin

Lernziel

Sie haben das Cynefin Video auf Moodle gesehen und können den Nutzen von Cynefin im Kontext der Software Entwicklung erklären. Sie kennen die folgenden Begriffe und können sie erklären: Cynefin, Framework, Exaptation, 4 + 1 Domains (Simple, Complicated, Complex, Chaotic, Disorder), Causality, Correlation, Constraint.

Sie haben das Codefin Video auf Moodle gesehen und können die praktische Anwendung in der Software Entwicklung erklären.

14.1 Nutzen von Cynefin

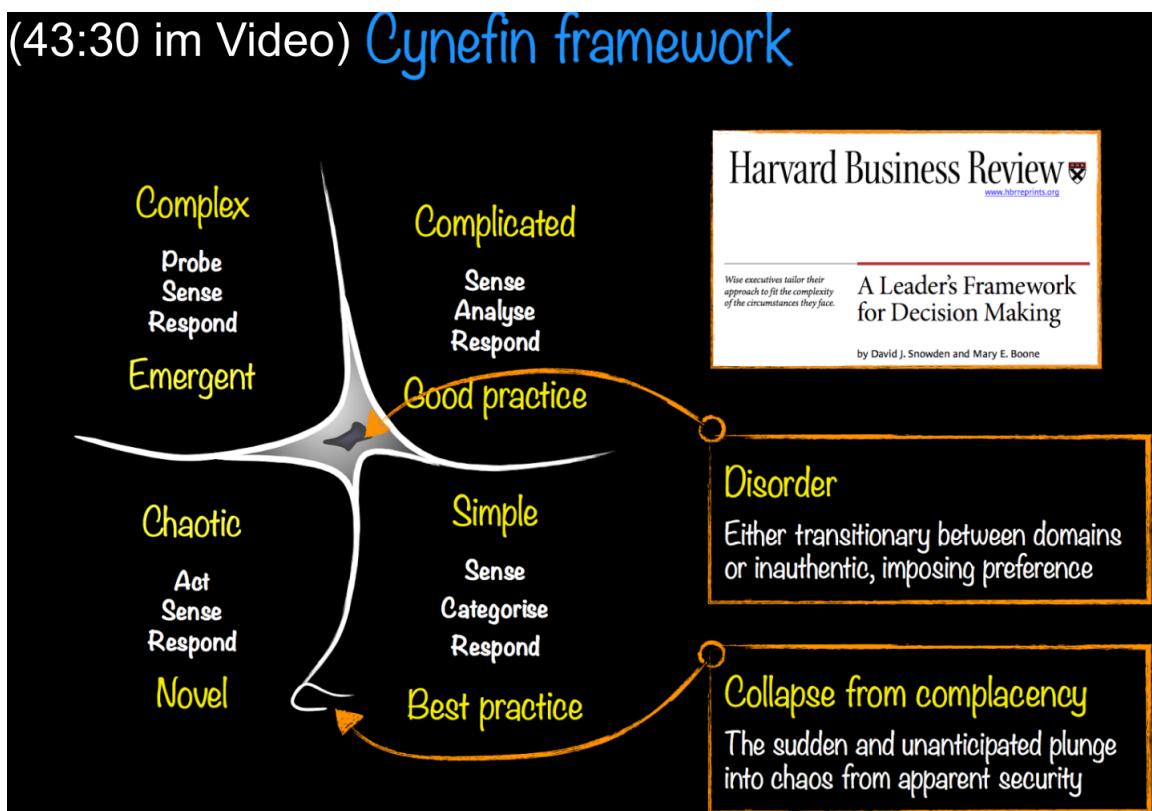
Video zu [Cynefin](#) (bis 49:50) muss geschaut werden!!!

14.2 Cynefin

Das Cynefin Framework ist ein Modell, das helfen soll, zu verstehen, welches Verhalten in bestimmten Kontexten zum Erfolg führt.

→ Es basiert auf der Komplexitätstheorie (siehe THIN)

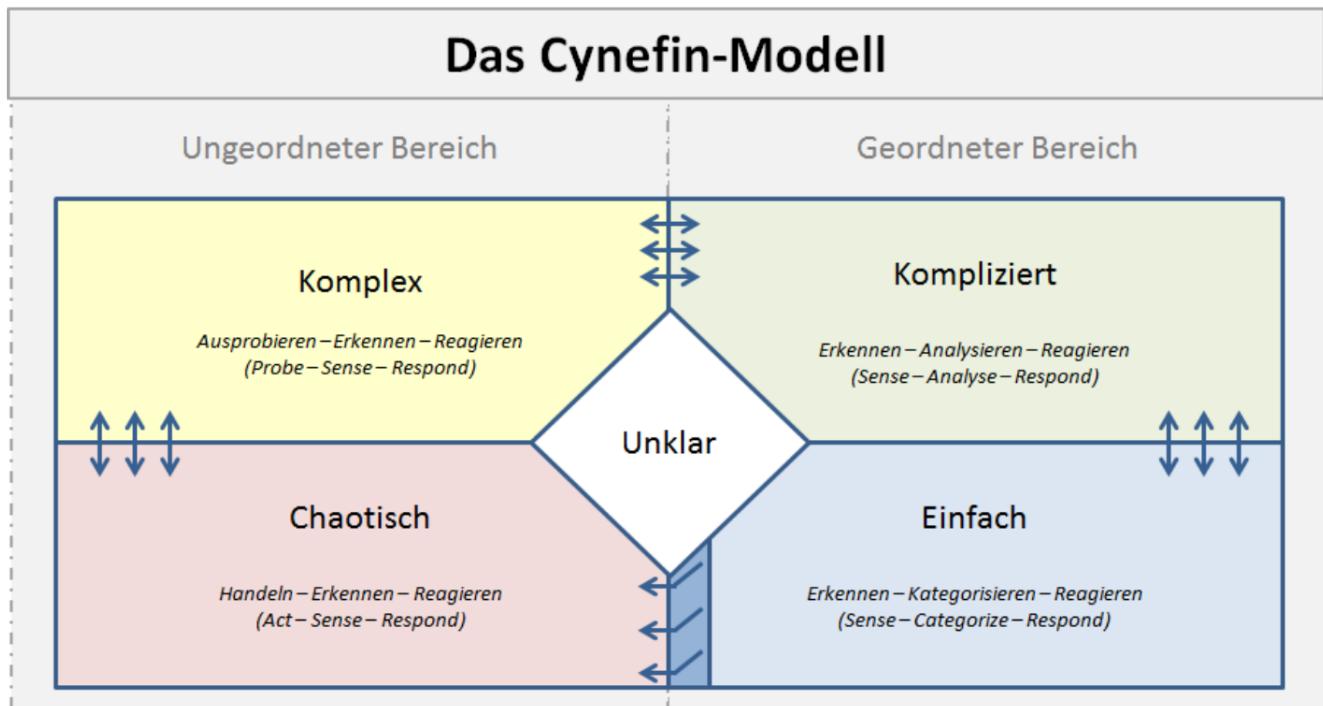
14.3 Framework



14.4 Exaptation

Der Prozess, durch den Merkmale Funktionen erwerben, für die sie ursprünglich nicht angepasst oder ausgewählt wurden.

14.5 4+1 Domains (Simple, Complicated, Complex, Chaotic, Disorder)



15 Kanban / Lean Software Development

Lernziel

Sie wissen, wie Agile, Lean Software Development, Scrum, XP und Kanban zusammenhängen und sie kennen deren Herkunft. Sie kennen die Principles, Practices und Values von Kanban. Sie können erklären, wie ein Kanban Board und das CFD funktionieren. Sie kennen die folgenden Begriffe und können diese erklären: Pull System, Limited WIP, «Visualize the Workflow» (Kanban Board), Cumulative Flow Diagramm, Lead time resp. Cycle time, «Kaizen», Waste in Software Development. Sie kennen die «Principles of Lean» und können diese erklären. Sie kennen in groben Zügen die Unterschiede und Gemeinsamkeiten von Scrum und Kanban.

15.1 Zusammenhänge

Lean und **Agile** bestehen aus einer Menge kompatibler Werte und Prinzipien, die beschreiben, wie Produktentwicklung erfolgreich umgesetzt werden kann.

Scrum, XP und Kanban sind drei konkrete Möglichkeiten, diese Prinzipien in die Praxis umzusetzen. Mit anderen Worten, sie sind drei sich leicht überschneidende Varianten der schlanken und agilen Softwareentwicklung.

Diese Techniken können als **Prozesswerkzeuge** betrachtet werden. Die drei Toolkits haben erhebliche Überschneidungen, z.B. empfehlen alle drei die Verwendung von physischen Task-Boards, um zu visualisieren, was vor sich geht.

15.2 Prinzipien von Kanban

Es gibt sechs Grundprinzipien von Kanban, die sich in zwei Gruppen unterteilen lassen:

Die Prinzipien des **Changemanagement**.

Die Prinzipien der **Leistungserbringung**.



15.3 Praktiken von Kanban

Die Allgemeinen Praktiken von Kanban definieren wesentliche Aktivitäten für diejenigen, die Kanban-Systeme verwalten.

Workflow Visualisieren: Darstellung der Work-Items und des Workflows auf einer Kartenwand oder einem elektronischen Board.

Limitierung des Work-in-Progress: Vereinbarte Grenzen festlegen, wie viele Work-Items gleichzeitig in Arbeit sind.

Messen und Verwalten des Flusses: Tracking der Arbeitsaufgaben, um zu sehen, ob der Arbeitsfortschritt in einem gleichmässigen Tempo erfolgt.

Prozessrichtlinien explizit gestalten: Abstimmung und Veröffentlichung der Richtlinien, wie die Arbeit gehandhabt wird.

Modelle zur Bewertung von Verbesserungsmöglichkeiten: Anpassung des Prozesses unter Verwendung von Ideen aus dem Systemdenken.

15.4 Werte von Kanban

Die Kanban-Methode ist Wert geleitet. Sie ist motiviert von der Überzeugung, dass der Respekt vor allen Personen, die zu einem kollaborativen Unternehmen beitragen, nicht nur für den Erfolg des Unternehmens notwendig ist, sondern auch dafür, dass es sich überhaupt lohnt.

Transparenz: Die Überzeugung, dass die offene Weitergabe von Informationen den Fluss von Geschäftswerten verbessert.

Ausgewogenheit: Das Verständnis, dass unterschiedliche Aspekte, Standpunkte und Fähigkeiten für die Effektivität ausgeglichen werden müssen.

Kollaboration: Zusammenarbeiten. Die Kanban-Methode wurde entwickelt, um die Art und Weise zu verbessern, wie Menschen zusammenarbeiten, daher steht die Zusammenarbeit im Mittelpunkt.

Kundenorientierung: Das Ziel des Systems kennen. Jedes Kanban-System fließt zu einem Punkt der Wertrealisierung – wenn der Kunde einen benötigten Artikel oder eine Dienstleistung erhält.

Fluss: Die Erkenntnis, dass Arbeit ein Fluss von Wert ist, ob kontinuierlich oder episodisch. Das Erkennen des Flusses ist ein wesentlicher Ausgangspunkt für den Einsatz von Kanban.

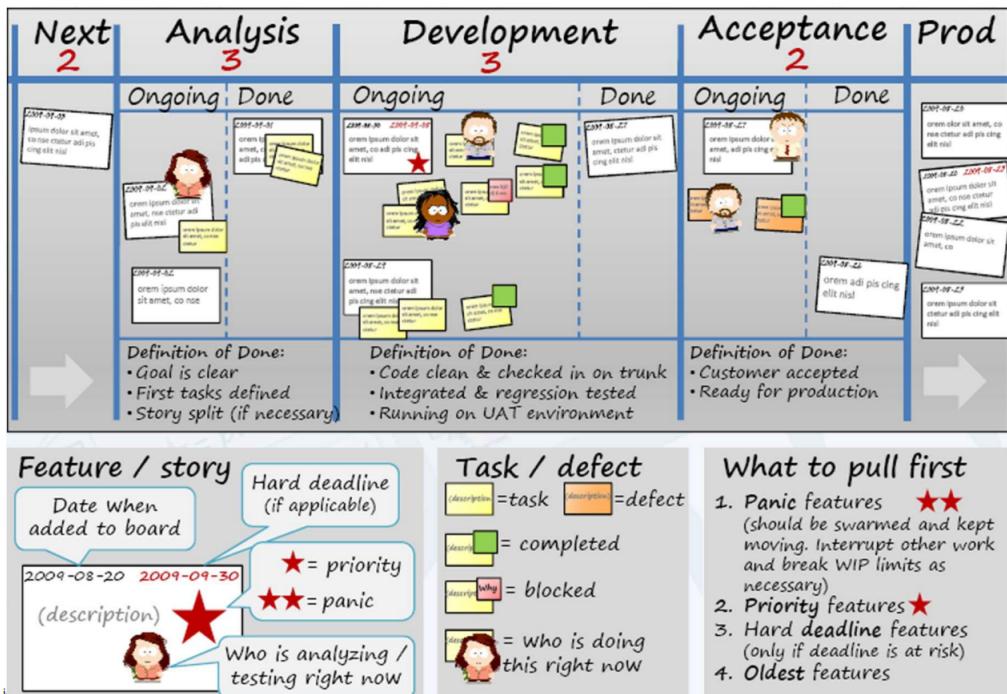
Führung: Die meisten Organisationen haben ein gewisses Mass an hierarchischer Struktur, aber in Kanban wird Führung auf allen Ebenen benötigt, um Wertlieferung und Verbesserung zu erreichen.

Verständnis: In erster Linie Selbsterkenntnis, um voranzukommen. Kanban ist eine Verbesserungsmethode, und die Kenntnis des Ausgangspunktes ist grundlegend.

Einvernehmen: Die Verpflichtung, gemeinsam auf Ziele hinzuarbeiten, wobei Meinungsverschiedenheiten oder unterschiedliche Ansätze respektiert werden.

Respekt: Wertschätzung, Verständnis und Rücksichtnahme auf Menschen.

15.5 Kanban Board



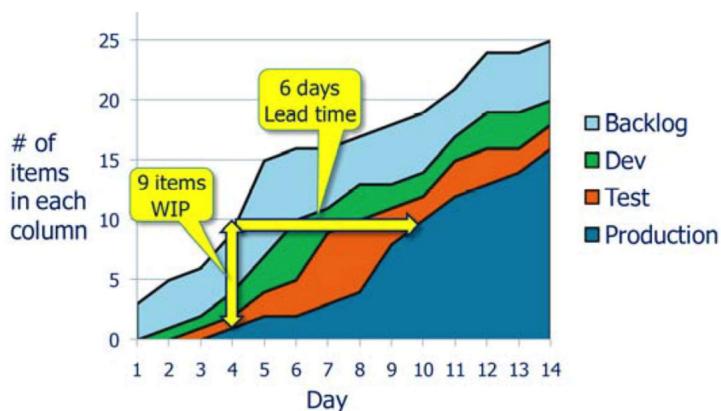
15.6 Begriffe und erklärung

Pull System

Kanban-Methode (grosses K): Evolutionäre Veränderungsmethode, die ein **kanban** (kleines k) Pull-System, Visualisierung und andere Werkzeuge nutzt, um die Einführung von Lean-Ideen in die Technologieentwicklung und den IT-Betrieb voranzutreiben.

Cumulative Flow Diagramm

Kumulative Flussdiagramme (CFDs), zeigen die kumulative Anzahl von Eingängen und Abgängen in einem Prozess oder in Teilen eines Prozesses. CFD veranschaulichen sehr schön, wie gleichmäßig der Fluss ist und wie WIP Ihre durchlaufzeit beeinflusst.



Lead Time bzw. Cycle Time

Messung der Durchlaufzeit (durchschnittliche Zeit bis zur Fertigstellung eines Items), Optimierung des Prozesses, um die Durchlaufzeit so gering und vorhersehbar wie möglich zu halten.

Kaizen

Ist ein japanisches wort, das folgendes bedeutet:

Kai: Veränderung

Zen: Gut

Also gesamthaft bedeutet es, kleine Veränderungen zum Besseren vorzunehmen

Veränderungen sind am besten, wenn sie von der Person geschaffen werden, die die Arbeit macht. Die Person, die die Arbeit macht, benutzt ihren eigenen gesunden Menschenverstand und ihre Intuition.

15.7 Lean

Lean ist der westliche Begriff für das, was die Japaner "TPS" (Toyota Production System) nennen – ein Ansatz für die Fertigung, der dazu beigetragen hat, Toyota zu einem sehr erfolgreichen Automobilhersteller zu machen.

Die mit dem **TPS** zugrunde liegenden Prinzipien, der Toyota Way, haben sich als fast überall anwendbar erwiesen, auch in der Softwareentwicklung.

Agile und Lean können als Cousins mit gemeinsamen Werten, aber unterschiedlichen Ursprüngen betrachtet werden. Lean ist aus der Fertigung entstanden. Agile aus der Software-Entwicklung und dem Agilen Manifest entstanden.

Beide Prinzipien passen gut zusammen und sind sehr breit anwendbar.

Immer mehr Softwareentwicklungsunternehmen entdecken, wie sie diese Prinzipien kombinieren können, um die gesamte Kette vom Produktkonzept bis zur Auslieferung abzudecken.

15.8 Prinzipien von Lean

- **Verschwendungen eliminieren:** Nur für etwas Zeit aufwenden, was einen echten Mehrwert für den Kunden bringt
- **Lernen verstärken:** Falls schwierige Probleme auftauchen, muss das Feedback verstärkt werden
- **So spät wie möglich entscheiden:** Verschiedene Optionen evaluieren, Entscheidungen verzögern, bis sie auf Basis von Fakten getroffen werden können
- **So schnell wie möglich Liefern:** Dem Kunden einen Mehrwert liefern, sobald er danach fragt
- **Das Team befähigen:** Die Menschen, die Mehrwert schaffen, ihr volles Potenzial nutzen lassen
- **Integrität einbauen:** Nicht versuchen, Integrität nachträglich aufzutragen – sie muss eingebaut werden
- **Das Ganze sehen:** Sich vor der Versuchung hüten, Teile auf Kosten des Ganzen zu optimieren

15.8.1 Verschwendungen eliminieren

- Alle Verschwendungen eliminieren, die keinen Wert für das Projekt darstellen
- Drei Arten von Verschwendungen
 - Verschwendungen bei der Code-Entwicklung
 - Verschwendungen in der Projektleitung

- Verschwendungen im Mitarbeiterpotenzial
- **Verschwendungen bei der Code-Entwicklung**
 - Teilweise abgeschlossene Arbeiten
 - Kann veraltet und unbrauchbar werden
 - Lösung: Iterativer Zyklus mit modularem Code
- Defekte
 - Korrektur und erneutes Testen
 - Lösung: Aktuelle Testsuite, Kundenfeedback
- **Verschwendungen in der Projektleitung**
 - Zusätzliche Prozesse
 - verschwenderische/unnötige Dokumentation
 - Lösung: Überprüfung der Dokumentation
- Code-Übergaben
 - Verlust von Wissen
 - Lösung: Keinen Code weitergeben
- Zusätzliche Funktionen
 - Der Kunde will oder braucht eine Funktion nicht
 - Lösung: Kontinuierliche Interaktion mit dem Kunden
- **Verschwendungen im Mitarbeiterpotenzial**
 - Task Switching
 - Die Gefahr des Multi-Tasking
 - Lösung: Konzentrieren Sie sich bei jeder Freigabe auf eine Aufgabe
- Warten auf Anweisungen oder Informationen
 - Entwickler sind zu teuer, um herumzusitzen!
 - Lösung: Den Entwicklern erlauben, Entscheidungen zu treffen, Zugang zu Informationen

15.9 Kanban vs. Scrum

Scrum	Kanban
Timeboxed Iterationen vorgeschrieben	Timeboxed Iterationen sind optional
Das Team verpflichtet sich zu einer bestimmten Menge an Arbeit für diese Iteration.	Verpflichtung ist optional
Verwendet Velocity als Standardmetrik für Planung und Prozessverbesserung	Verwendet die Durchlaufzeit als Standardmetrik für die Planung und Prozessverbesserung
Funktionsübergreifende Teams vorgeschrieben	Funktionsübergreifende Teams sind optional. Spezialisten-Teams erlaubt
Die Aufgaben müssen so aufgeteilt werden, dass sie innerhalb eines Sprints erledigt werden können	Es ist keine bestimmte Aufgabengröße vorgeschrieben
WIP indirekt begrenzt (pro Sprint)	WIP direkt begrenzt (pro Workflow-Zustand)
Schätzung vorgeschrieben	Schätzung optional