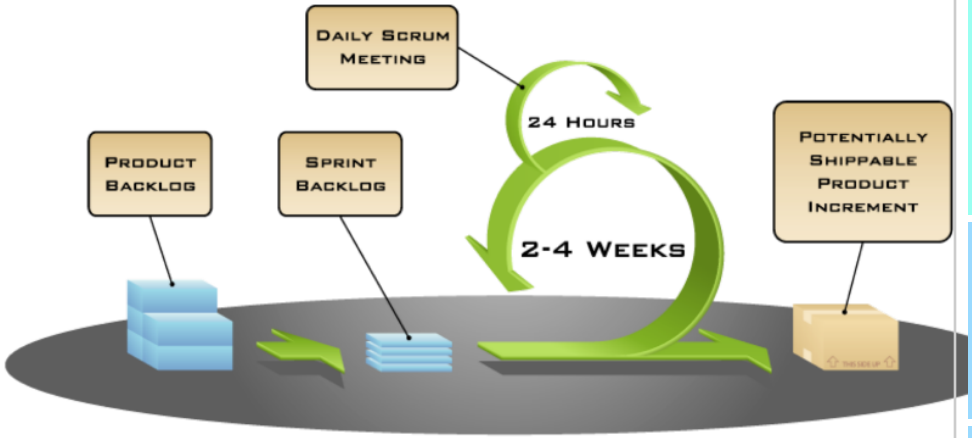
Softwareentwicklung 2

Jil Zerndt
FS 2025

Introduction to SCRUM

Intro in PM4



SCRUM

Product Backlog

- the requirements for the product
- List of all desired work on the project
- Single source of requirements for any changes to be made to the product
- Prioritized by the Product Owner

Einführung in Agile Software-Entwicklung

Traditionelle Entwicklungsmethoden

Traditionelle Entwicklungsmethoden

Traditionelle Entwicklungsmethoden wie Wasserfall und das V-Modell sind durch einen sequenziellen Prozess gekennzeichnet:

- Vordefinierte Phasen, die nacheinander abgearbeitet werden
- Umfangreiche Dokumentation in jeder Phase
- Begrenzte Möglichkeiten für Änderungen nach Festlegung der Anforderungen
- Abnahme und Auslieferung erst nach vollständiger Fertigstellung

Herausforderungen in Softwareprojekten

Softwareprojekte stehen vor einzigartigen Herausforderungen:

- Komplexe, nicht-physische Produkte
- Anforderungen ändern sich während der Entwicklung
- Hohe Fehlerrate im Vergleich zu anderen Branchen
- Technologie entwickelt sich schneller als Methoden
- Weitere Faktoren: unklare Ziele, mangelhafte Führung, schlechte Kommunikation

Projektrisiko

Die Risikowahrscheinlichkeit ist in technologischen Projekten im Vergleich zu anderen Branchen überdurchschnittlich hoch.

- Studien zeigen, dass Technologieprojekte eine Erfolgsrate von nur 30-35% haben
- Ähnliche Fehlerraten wären in vielen anderen Branchen nicht akzeptabel
- Beispiele: gescheiterte IT-Großprojekte wie Ïnsieme"(Schweiz), SAP-Projekte, Helvetia-IT-Projekt

Die Entstehung von Agile

Zentrale Fragestellung

Wie kann das Risiko in Software-Entwicklungsprojekten reduziert werden?

Agile Manifesto

Im Februar 2001 trafen sich 17 Entwickler in Snowbird, Utah und formulierten das Agile Manifesto. Diese Entwickler repräsentierten verschiedene leichtgewichtige Methoden:

- Extreme Programming (XP)
- Scrum
- DSDM (Dynamic Systems Development Method)
- Adaptive Software Development
- Crystal
- Feature-Driven Development
- Pragmatic Programming

Sie suchten nach Alternativen zu dokumentbasierten und umfangreich geregelten Entwicklungsprozessen.

Werte des Agilen Manifesto

Das Agile Manifesto definiert vier Kernwerte:

- **Individuen und Interaktion** sind wichtiger als Prozesse und Tools
- **Lauffähige Software** ist wichtiger als durchgängige Dokumentation
- **Zusammenarbeit mit dem Kunden** ist wichtiger als Vertragsverhandlungen
- **Auf Veränderungen reagieren** ist wichtiger als einem Plan folgen

"... while there is value in the items on the right, we value the items on the left more"

Agile Manifesto Prinzipien

Das Agile Manifesto umfasst 12 Prinzipien:

1. **Customer Satisfaction:** Höchste Priorität hat die Zufriedenstellung des Kunden durch frühe und regelmäßige Lieferung wertvoller Software
2. **Welcome Change:** Neue und veränderte Anforderungen sind auch spät in der Entwicklung willkommen
3. **Deliver Frequently:** Lauffähige Software wird regelmäßig geliefert (Wochen statt Monate)
4. **Working Together:** Fachexperten und Entwickler müssen täglich zusammenarbeiten
5. **Motivated Team:** Projekte werden um motivierte Individuen organisiert mit der nötigen Unterstützung und Vertrauen
6. **Face-to-Face:** Direkte Kommunikation ist die effizienteste Methode

Agile Manifesto Prinzipien (Fortsetzung)

7. **Working Software:** Lauffähige Software ist das zentrale Fortschrittsmaß
8. **Constant Pace:** Nachhaltige Entwicklung mit konstantem Arbeitstakt
9. **Good Design:** Ständiger Fokus auf technische Exzellenz und gutes Design
10. **Simplicity:** Die Kunst, Arbeit zu maximieren, die nicht getan werden muss
11. **Self-Organisation:** Die besten Architekturen und Anforderungen entstehen durch selbstorganisierte Teams
12. **Reflect & Adjust:** Regelmäßige Reflektion und Anpassung der Arbeitsweise

Die Variablen der Software-Entwicklung

Die vier Variablen der Software-Entwicklung:

- **Zeit:** Zeitplan für die Fertigstellung
- **Ressourcen:** Budget und Personal
- **Qualität:** Technische Funktionalität und Zuverlässigkeit
- **Scope:** Umfang der zu implementierenden Features

Externe Kräfte (Kunden, Vorgesetzte) können drei dieser Variablen definieren, das Entwicklungsteam definiert die vierte. In agilen Projekten ist typischerweise der Scope die variable Komponente.

Kosten für Änderungen

Zwei unterschiedliche Sichtweisen:

- **Traditionelle Sicht:** Die Kosten für Änderungen steigen exponentiell mit dem Projektfortschritt an
- **Agile Sicht:** Durch moderne Praktiken (kontinuierliche Integration, automatisierte Tests, Refactoring) bleiben die Kosten für Änderungen über den Projektverlauf flacher

Grundidee hinter Agile Methoden

Der Kerngedanke ist, dass Software-Entwicklung ein empirischer Prozess ist, nicht ein definierter:

- Komplexität erfordert einen adaptiven Ansatz
- Regelmäßiges Inspizieren und Anpassen
- Lernen und Verbessern während des Projekts
- Risikominimierung durch frühzeitige Lieferung von Mehrwert
- Reaktion auf Veränderungen statt starres Befolgen eines Plans

Die drei Gesetze der Software-Entwicklung

Humphrey's Law

"Menschen wissen nicht, was sie wollen, bevor sie es sehen."
Dieses Gesetz erklärt, warum inkrementelle Entwicklung mit regelmäßigem Feedback so wichtig ist. Es ist ein Hauptgrund, warum große Technologieunternehmen regelmäßige Releases veröffentlichen, anstatt zu versuchen, perfekte Software auf einmal zu liefern.

Ziv's Law

SSoftwareentwicklung ist unvorhersehbar und kann nie vollends verstanden werden."
Auch bekannt als das Unsicherheitsprinzip der Softwareentwicklung. Es betont, dass komplexe Systeme inhärent schwer zu verstehen und vorherzusagen sind, was agile Praktiken zur Risikominderung notwendig macht.

Conway's Law

SSoftware ist ein Spiegel der Firma und der Menschen, die sie entwerfen."
Örganisationen, die Systeme entwerfen, sind gezwungen, Entwürfe zu erstellen, die die Kommunikationsstrukturen dieser Organisationen abbilden."
Dies bedeutet, dass die Architektur eines Systems die Kommunikationsstruktur der Organisation widerspiegelt, die es erstellt hat. Die Qualität der Schnittstellen zwischen Systemkomponenten entspricht der Qualität der Kommunikation zwischen den entsprechenden Teams.

eXtreme Programming (XP)

Einführung in eXtreme Programming

Was ist eXtreme Programming?

eXtreme Programming (XP) ist eine agile Methode zur Softwareentwicklung, die 1996 von Kent Beck beim Chrysler Comprehensive Compensation System (C3) Projekt entwickelt wurde. XP zeichnet sich durch folgende Merkmale aus:

- Fokus auf Kommunikation, Einfachheit, Feedback, Mut und Respekt
- Kurze Entwicklungszyklen mit kontinuierlicher Verbesserung
- Betonung von technischer Exzellenz und gutem Design
- Reaktionsfähigkeit gegenüber wechselnden Kundenanforderungen

Ausgangslage und Problematik

XP wurde entwickelt, um typische Risiken in Entwicklungsprojekten zu adressieren:

- Zeitpläne verschieben sich
- Projekte werden abgebrochen
- Systementwicklung verläuft nicht linear (Fehlerrate steigt überproportional)
- Fachliche Missverständnisse
- Veränderungen des fachlichen Umfelds
- Implementierung der falschen Features
- Personalfluktuation

XP etabliert Prinzipien und Praktiken, um mit diesen Risiken umgehen zu können.

Die Werte von XP

Kommunikation (Communication)

- Jeder ist Teil des Teams mit täglicher Face-to-Face-Kommunikation
- Gemeinsame Arbeit in allen Aspekten des Projekts
- Zusammen wird die bestmögliche Lösung für Probleme gefunden

Einfachheit (Simplicity)

- Nur das Notwendige und Angeforderte implementieren
- Den Wert im Verhältnis zum Aufwand maximieren
- Kleine, einfache Schritte zum Ziel machen und Fehler sofort lösen
- Lösungen schaffen, die mit vernünftigem Aufwand langfristig wartbar sind

Feedback

- Commitments einer Iteration ernst nehmen und lauffähige Software liefern
- Software früh und oft zeigen, sorgfältig zuhören und notwendige Änderungen vornehmen
- Über das Projekt sprechen und den Prozess entsprechend anpassen

Mut (Courage)

- Die Wahrheit über Fortschritt und Schätzungen kommunizieren
- Keine Ausreden für Scheitern dokumentieren, sondern für Erfolg planen
- Keine Angst haben, da niemand allein arbeitet
- Sich an Veränderungen anpassen, wann immer sie auftreten

Respekt

- Jedes Teammitglied wird als wertvoll angesehen und respektiert
- Alle tragen etwas bei, und sei es nur Enthusiasmus
- Entwickler respektieren die Expertise des Kunden und umgekehrt
- Vorgesetzte respektieren das Recht des Teams, Verantwortung zu übernehmen

Die Prinzipien von XP

Fundamentale Prinzipien

- **Rapid Feedback:** Schnellstmögliche Rückmeldung
- **Assume Simplicity:** Davon ausgehen, dass eine einfache Lösung existiert
- **Incremental Change:** Änderungen erfolgen inkrementell
- **Embracing Change:** Änderungen werden erwartet und vorgesehen
- **Quality Work:** Bestmögliche Lösung anstreben

Weitere Prinzipien

- Teach learning (Lernen lehren)
- Small initial investment (Kleine anfängliche Investition)
- Play to win (Auf Sieg spielen)
- Concrete experiments (Konkrete Experimente)
- Open, honest communication (Offene, ehrliche Kommunikation)
- Work with people's instincts (Mit den Instinkten der Menschen arbeiten)
- Accepted responsibility (Akzeptierte Verantwortung)
- Local adaptation (Lokale Anpassung)
- Travel light (Leicht reisen)
- Honest measurement (Ehrliche Messung)

Learning to Drive

Ein Leitmotiv von XP, formuliert von Kent Beck:
"Wir müssen die Entwicklung von Software durch viele kleine Anpassungen steuern, nicht durch wenige große Anpassungen, ähnlich wie beim Autofahren. Das bedeutet, dass wir das Feedback brauchen, um zu wissen, wann wir leicht abweichen, wir brauchen viele Gelegenheiten für Korrekturen, und wir müssen in der Lage sein, diese Korrekturen zu vernünftigen Kosten vorzunehmen."

XP Praktiken

The Planning Game

Balance zwischen geschäftlichen und technischen Überlegungen:

- Business entscheidet über: Scope, Priorität, Zusammensetzung der Releases, Release-Daten
- Techniker entscheiden über: Schätzungen, Impacts, Prozesse, Feinplanung

Small Releases

- Jedes Release sollte so klein wie möglich sein und die wichtigsten Geschäftsanforderungen enthalten
- Das Release muss als Ganzes Sinn machen (keine halbfertigen Features)
- Lieber monatlich als halbjährlich liefern

Metaphor

- Jeder im Team muss ein "gemeinsames Verständnis"für das System haben
- Ein "gemeinsamer Wortschatz"wird etabliert
- Dies gilt für technische und nichttechnische Personen
- Definiert, was die Grundelemente des Systems sind und wie sie in Beziehung stehen

Simple Design

Das richtige Design für ein Softwaresystem:

- Alle Tests bestehen
- Keine redundante Logik enthalten
- Die kleinstmögliche Anzahl von Klassen und Methoden haben
- SSetzen Sie ein, was Sie brauchen, wenn Sie es brauchen"(YAGNI-Prinzip)
- Emergent, wachsendes Design; kein Överdesigning"

Testing

- Programmfunktionen ohne automatisierte Tests existieren nicht
- Tests werden Teil des Systems
- Tests ermöglichen dem System, Änderungen zu akzeptieren
- Entwicklungszyklus: Zuhören (Voraussetzungen) → Test (zuerst schreiben) → Code (am einfachsten) → Design (Refactoring)

Refactoring

- Bei der Implementierung einer Funktion prüfen, ob vorhandener Code verbessert werden kann
- Automatisierte Tests bieten ein Sicherheitsnetz für angstfreies Refactoring
- Kontinuierliche Verbesserung der Codequalität

Pair Programming

- Produktionscode wird von zwei Personen geschrieben, die gemeinsam an einem Bildschirm arbeiten
- Zwei Rollen: Der aktive Programmierer konzentriert sich auf die aktuelle Methode, der andere denkt über den weiteren Kontext nach
- Paare wechseln häufig

Collective Ownership

- Jeder, der eine Verbesserungsmöglichkeit sieht, kann jederzeit jeden Teil des Codes ändern
- Alle übernehmen Verantwortung für das gesamte System
- Nicht jeder kennt jeden Teil gleich gut, aber jeder weiß etwas über jeden Teil

Continuous Integration

- Code wird mindestens einmal täglich integriert und getestet
- Der Build-Prozess läuft automatisiert auf einem dedizierten Computer
- Automatisierte Tests ermöglichen es, Probleme frühzeitig zu erkennen

40 Hours Week

- Nachhaltige Entwicklung mit gleichmäßig verteiltem Aufwand
- Längere Überstunden wirken sich negativ auf die Produktivität aus
- Ziel: Morgens frisch, abends müde und zufrieden sein
- Abstand vom Computer kann zu Äha!Momenten führen

On-Site Customer

- Ein echter Kunde ist physisch im Team präsent, um Fragen zu beantworten
- Echter Kunde = tatsächlicher Benutzer des Systems
- Muss nicht zu 100% am Projekt arbeiten, aber erreichbar sein
- Hilft bei der Priorisierung

Coding Standards

- Kollektives Eigentum und kontinuierliches Refactoring erfordern einheitliche Codierungspraktiken im Team
- Standards werden gemeinsam definiert und eingehalten

Weitere XP-Praktiken (2. Version)

Test-Driven Development (TDD)

Zyklische Vorgehensweise:

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

Slack

- Einplanen von kleineren Aufgaben, die bei Bedarf gestrichen werden können
- Realistische, erreichbare Ziele setzen
- Puffer für unvorhergesehene Probleme einbauen

Spike

- Eine zeitlich begrenzte Untersuchung eines Problems oder einer Technologie
- Dient dem Wissensaufbau, wenn das Team nicht genug weiß, um eine Story zu schätzen
- Timeboxing ist wichtig

Incremental Design

- Kontinuierliche Verbesserung in kleinen oder kleinsten Schritten
- Design entsteht inkrementell während der Entwicklung
- Vermeidet umfangreiches Upfront-Design

Self-Organized Team

- Das Unternehmen legt die Prioritäten fest
- Das Team organisiert sich selbst, um den besten Weg zur Umsetzung zu finden
- Keine Mikromanagement oder externe Arbeitszuweisung

Kosten von Änderungen in XP

Traditionelle vs. XP-Sicht auf Änderungskosten

- **Traditionelle Sicht:** Kosten steigen exponentiell mit dem Projektfortschritt
- **XP-Sicht:** Durch XP-Praktiken (automatisierte Tests, Refactoring, kontinuierliche Integration) bleibt die Kostenkurve flacher

Diese flachere Kostenkurve ermöglicht es, Änderungen auch in späteren Projektphasen wirtschaftlich umzusetzen.

XP-Fazit

- XP ist keine SSilberkugel es gibt keinen magischen Prozess, der in jedem Projekt identisch funktioniert
- XP beschreibt zentrale Werte und Prinzipien, die auf den konkreten Kontext angewendet werden müssen
- Agile Teams müssen ihre Arbeit kontinuierlich reflektieren
- XP erscheint auf den ersten Blick weniger formal als traditionelle Methoden, erfordert aber eine hohe Disziplin

Test-Driven Development (TDD) in der Praxis

Vorbereitung

- Verstehen der Anforderung
- Identifizieren der zu testenden Funktionalität
- Testumgebung einrichten

TDD-Zyklus (Red-Green-Refactor)

- **Red:** Schreibe einen Test, der fehlschlägt
- **Green:** Implementiere den einfachsten Code, der den Test bestehen lässt
- **Refactor:** Verbessere den Code, ohne die Funktionalität zu ändern

Fortgeschrittene Anwendung

- Tests als Dokumentation verwenden
- Testabdeckung überwachen
- Testsuiten automatisiert ausführen
- Integration in CI/CD-Pipeline

TDD-Beispiel: Einfache Additionsfunktion

```java
// 1. Test schreiben (Red)
@Test
public void testAddition() {
    Calculator calc = new Calculator();
    assertEquals(5, calc.add(2, 3));
}

// 2. Implementierung (Green)
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}

// 3. Refactoring (falls noetig)
// In diesem einfachen Beispiel ist kein Refactoring noetig
```

Agile Planung und Schätzung

Der Planungsprozess

Der Sinn der Planung

Schätzung und Planung sind kritisch für den Erfolg eines Softwareprojekts, auch in agilen Methoden:

- Hilft bei Ressourcenallokation: "Wer arbeitet wieviel während welcher Zeit im Projekt?"
- Ermöglicht Fortschrittskontrolle: Ïst das Projekt auf dem richtigen Weg?"
- Unterstützt bei der Terminplanung: "Wann werden wir fertig sein?"

Missverständnisse über agile Planung

Häufige Missverständnisse:

- Ägile Teams brauchen keine Planung"
- "Wenn wir genug agil sind, brauchen wir keinen Plan, denn wir reagieren ja schnell genug"

In der Praxis erstellen agile Teams Pläne auf zwei Ebenen:

- Grobe langfristige Planung für die strategische Ausrichtung
- Detaillierte kurzfristige Arbeitsplanung für die nächsten Wochen oder Monate

Der Trichter der Unsicherheit

Projekte beginnen mit hoher Unsicherheit, die im Laufe der Zeit abnimmt. Der "Trichter der Unsicherheitßeigt, wie sich die Genauigkeit von Schätzungen mit fortschreitendem Projektfortschritt verbessert:

- Zu Projektbeginn: Schätzungen können um +/- 400% abweichen
- Nach detaillierter Anforderungsanalyse: +/- 50%
- Nach Design: +/- 25%
- Während der Implementierung: +/- 10%

Warum Planen trotz Unsicherheit?

Gründe für Planung trotz der Herausforderungen:

- Organisationen benötigen Schätzungen für Budget, Marketing, Rollout usw.
- Planung unterstützt die Suche nach Wert: "Was sollen wir bauen?"
- Ein guter iterativer Planungsprozess:
  - Reduziert Risiken
  - Verringert Unsicherheit
  - Unterstützt bessere Entscheidungsfindung
  - Schafft Vertrauen
  - Verbessert die Kommunikation

Agile Planung

Merkmale der agilen Planung:

- Fokus liegt auf dem Prozess der Planung, nicht auf dem Plan selbst
- "Pläne sind Dokumente – Planung ist eine Aktivität"
- Agile Pläne ändern sich oft: Während des Projekts lernen wir ständig Neues
- Kundenanforderungen können sich ändern
- Die Umgebung kann komplexer sein als erwartet

Agile Herangehensweise

Der agile Ansatz für die Planung

Zentrale Idee:

- Ein Projekt erzeugt neue Fähigkeiten und neues Wissen in schneller Abfolge
- Neue Fähigkeiten werden als Produkt geliefert
- Neues Wissen ist die Basis, um das Produkt bestmöglich umzusetzen:
  - Wissen über das Produkt: Was soll gebaut werden?
  - Wissen über das Projekt: Team, Technologie, Risiken etc.

Mehrere Planungsebenen

Agile Teams planen auf mindestens drei Ebenen:

- **Release-Ebene:** Umfasst mehrere Iterationen (3-9 Monate)
- **Iterations-Ebene:** Typischerweise 2-4 Wochen
- **Tages-Ebene:** Tägliche Planung und Anpassung

Dies ermöglicht eine Balance zwischen langfristiger Vision und kurzfristiger Anpassungsfähigkeit.

Condition of Satisfaction

- Jedes Projekt wird mit einer gewissen Menge von Zielen initialisiert
- Zusätzlich zu den Features gibt es Ziele bezüglich Zeitplan, Budget und Qualität
- Diese Ziele stellen für den Kunden oder Product Owner die "Conditions of Satisfaction"dar
- Sie bilden den Rahmen für Release- und Iterationsplanung

Typische "Condition of Satisfaction"für eine User Story Äls Benutzer möchte ich eine Reservierung stornieren können:

- Stornierung bis 24h im Voraus führt zu vollständiger Kostenrückerstattung
- Bei weniger als 24h Vorankündigung fällt eine Gebühr an
- Ein Stornierungscode wird generiert und per E-Mail zugesandt
- Die stornierte Reservierung wird im System als storniert markiert

User Stories

User Story

Eine User Story ist eine knappe und präzise Beschreibung eines Funktionalitätselements, das einem Benutzer der Software einen Nutzen stiftet:

- Format: Äls [Benutzerrolle] möchte ich [Ziel], damit ich [Nutzen]"
- Beispiel: Äls Verkäufer möchte ich, dass Verkaufschancen Zusatzinformationen zu Produkten beinhalten, damit ich den Nutzen der Produkte dem Kunden erklären und besser Zusatzprodukte verkaufen kann"

User Story Karten

User Story Karten haben drei Teile:

- **Card:** Eine schriftliche Beschreibung für Planungszwecke
- **Conversation:** Weitere Informationen und Abstimmungsdetails
- **Confirmation:** Tests, die sicherstellen, dass die Story vollständig ist

User Rollen

Warum User Rollen wichtig sind:

- Erweitern den Scope — nicht von einem einzigen User ausgehen
- Erlauben die Differenzierung von Nutzern nach:
  - Hintergrund
  - Vertrautheit im Umgang
  - Nutzungsfrequenz
  - Verwendetem Zielgerät
  - Nutzungszweck

Eigenschaften guter User Stories (INVEST)

Gute User Stories sind:

- **I**ndependent: Abhängigkeiten vermeiden
- **N**egotiable: Verhandelbar zwischen User und Entwicklung
- **V**aluable: Wertvoll für den Kunden oder die Nutzenden
- **E**stimatable: Für die Umsetzung zentral
- **S**mall: Normalerweise ein Satz
- **T**estable: Durch einen Testfall zu prüfen

Story Points und Velocity

Story Points

- Die Anzahl der Story Points bestimmt die relative Größe einer User Story
- Es gibt keine definierte Formel, sondern eine relative Messung
- Story Points schätzen den Aufwand, der für die Realisierung eines Features nötig ist
- Sie berücksichtigen Komplexität, Umfang und Risiko

Velocity

- Velocity ist die Fortschrittsmessung eines Teams
- Sie wird berechnet durch die Summe der während einer Iteration umgesetzten Story Points
- Die beste Schätzung ist, dass ein Team pro Iteration eine ähnliche Anzahl Story Points realisieren kann
- Sie dient als Basis für Release-Planung

Planning Poker durchführen

Vorbereitung

- Jeder Schätzer erhält Kartenset mit gültigen Schätzwerten (z.B. Fibonacci: 1, 2, 3, 5, 8, 13, 21)
- Product Owner/Kunde bereitet User Stories vor

Ablauf für jede User Story

- Product Owner liest die Story vor und erläutert sie
- Teammitglieder stellen Fragen zur Klärung
- Jeder wählt verdeckt eine Karte für seine Schätzung
- Alle decken gleichzeitig ihre Karten auf
- Bei Unterschieden diskutieren, besonders Ausreißer erklären ihre Einschätzung
- Wiederholen der Schätzung bis zur Übereinstimmung oder Annäherung

Tipps

- Timeboxing einsetzen (5-10 Minuten pro Story)
- Bei anhaltender Uneinigkeit die größere Schätzung wählen oder Story aufteilen
- "?"Karte für ich verstehe die Story nicht"
- "∞"Karte für ßu groß für eine Iteration"

Release- und Iterationsplanung

Release-Plan

Der Release-Plan ist ein Prozess, der einen Plan für mehrere Iterationen erstellt (3-9 Monate):

- Was soll wann durch wen gebaut werden
- Vor dem Planungsstart müssen die "Conditions of Satisfaction"bekannt sein
- Schätzungen für alle User Stories, die im Vertrag enthalten sind
- Festlegung der Iterationslänge (meist 2-4 Wochen)
- Schätzung der Velocity
- Priorisierung der User Stories durch den Product Owner

User Stories auswählen und Releasedatum festlegen

Nach Schätzungen und Velocity-Ermittlung:

- **Bei funktionsgetriebenem Projekt:** Summe der Story Points aller gewünschten Features geteilt durch erwartete Velocity = Anzahl Iterationen bis zum Release
- **Bei datumsgetriebenem Projekt:** Anzahl Iterationen bis zum Wunschdatum multipliziert mit erwarteter Velocity = Maximale Story Points für das Release

Release-Plan aktualisieren

- Nach jeder Iteration sollte der Release-Plan aktualisiert werden
- Anpassung basierend auf tatsächlicher Velocity, die von der ursprünglichen Schätzung abweichen kann
- Bei Problemen sollte der Product Owner mit dem Kunden über mögliche Scope-Reduktion verhandeln

Iterations-Plan

- Detailliertere Sicht auf die Arbeit während einer Iteration
- User Stories werden in konkrete Tasks aufgeteilt
- Jeder Task wird in Stunden geschätzt
- Team wählt selbst aus, welche Tasks es übernimmt

Priorisierung und Tracking

Priorisierung der User Stories

Faktoren für die Priorisierung:

- **Finanzieller Wert:** Wie viel Geld wird verdient oder gespart?
- **Kosten:** Aufwand für Entwicklung und Wartung
- **Neues Wissen:** Lernen über Produkt oder Prozess
- **Risikominderung:** Welche Risiken werden adressiert?

Kano-Modell der Kundenzufriedenheit

Klassifizierung von Features nach ihrem Einfluss auf die Kundenzufriedenheit:

- **Must-have Features:** Grundlegende Anforderungen, deren Fehlen zu Unzufriedenheit führt
- **Linear Features:** "Je mehr, desto besser Zufriedenheit steigt proportional
- **Exciters/Delighters:** Unerwartete Features, die Begeisterung auslösen

Tracking und Kommunikation

Überwachung des Release-Plans:

- **Release Burndown Chart:** Zeigt verbleibende Arbeit über Zeit
- **Release Burndown Bar Chart:** Visualisiert Fortschritt und Änderungen
- **Parking-Lot Chart:** Zeigt, wie viel eines "Themes"bereits realisiert wurde

Überwachung des Iteration-Plans:

- **Task Board:** Visualisierung des aktuellen Status aller Tasks
- **Iteration Burndown Chart:** Tägliche Fortschrittsmessung

Ein Task Board hat typischerweise folgende Spalten:

- To Do: Noch nicht begonnene Tasks
- In Progress: Tasks, an denen gerade gearbeitet wird
- Testing/Review: Fertige Tasks, die getestet werden
- Done: Vollständig abgeschlossene Tasks

Jeder Task wird als Karte dargestellt und während der Bearbeitung von links nach rechts bewegt.

DevOps

Einführung in DevOps

DevOps

DevOps ist eine Methodik, die darauf abzielt, die Lücke zwischen Entwicklung (Dev) und Betrieb (Ops) zu schließen:

- Kombination aus Menschen, Prozessen, Produkten und Technologien
- Ziel: Schnellere und bessere Software-Lieferung durch Integration vormals getrennter Teams
- Erweiterung agiler Praktiken mit Betriebs- und Automatisierungselementen
- Kontinuierliche Überwachung von Anwendungen im Betrieb

Die Entwicklung von DevOps

Die Rolle der IT hat sich dramatisch verändert:

- 1960/70er: Großrechner - IT-Systeme beschleunigen bestehende Prozesse
- 1980er: PC-Revolution - Office-Pakete, SSchatten-IT"
- 1990/2000er: Internet - IT als Wettbewerbsvorteil
- Heute: Mobilgeräte, Cloud, KI - "Technologie wird zum Geschäft"
- Agile Entwicklungsteams und Betriebsteams benötigen die gleichen Prioritäten → Entstehung von DevOps

Von Silos zu gemeinsamen Arbeitsabläufen

Traditionelles Modell (ca. 2000er Jahre):

- Entwickler schrieben Code, verpackten die Anwendung mit Dokumentation und übergaben sie an QA
- QA testete die Anwendung und übergab sie an das Produktionsteam
- Das Betriebsteam stellte die Software bereit und verwaltete sie - mit wenig direkter Interaktion mit den Entwicklern
- Das Sicherheitsteam überprüfte den Code erst nach dem Deployment
- Bei Problemen begann der Prozess von vorn

Dieser lineare Ansatz war langsam und frustrierend für alle Beteiligten.

Unterschiedliche Perspektiven zusammenbringen

- **Entwicklung:** Betrachtet eine Anwendung von innen nach außen, definiert durch die Struktur der Anforderungen
- **Betrieb:** Betrachtet eine Anwendung von außen nach innen, als ein weiteres zu verwaltendes Element, definiert durch die Struktur der Betriebsumgebung
- DevOps vereint diese Perspektiven in einer gemeinsamen, kollaborativen Sichtweise

Der DevOps-Lebenszyklus

Die vier Phasen des DevOps-Lebenszyklus

1. **Die Idee:** Teams sammeln Anforderungen und Feedback und beginnen, die benötigten Ressourcen zu skizzieren. Stack: GitHub Issues und Project Boards.
2. **Aufbauen:** Hier erwacht DevOps zum Leben. Mit Versionskontrolle und Cloud-basierten Entwicklungsumgebungen können Entwickler kontinuierlich Änderungen vornehmen und den Code gemeinsam in Echtzeit überprüfen. Auch die kontinuierliche Integration (CI) kommt zum Einsatz. Stack: GitHub Codespaces, GitHub Actions.
3. **Ausliefern:** Nach erfolgreichen Tests verwenden Teams Tools für kontinuierliche Bereitstellung (CD), um Codeänderungen automatisch in eine Testumgebung zu übertragen. Stack: CD-Pipelines mit GitHub Actions, GitHub Packages, Azure.
4. **Lernen:** Die Betriebsteams überwachen Releases mit Tools, die die Leistung messen und die Auswirkungen von Codeänderungen verfolgen. Sie sorgen für Stabilität, sammeln Kundenfeedback und kommunizieren eng mit den Entwicklern. Stack: Monitoring-Tools wie New Relic, Splunk, etc.

DevOps als Erweiterung von Agile

DevOps wendet agile Prinzipien auf den Betrieb an:

- Kontinuierliche Lieferung entspricht dem agilen Grundsatz: Ünsere höchste Priorität ist es, den Kunden durch frühzeitige und kontinuierliche Bereitstellung wertvoller Software zufriedenzustellen"
- DevOps bringt die agile Einstellung zum Wandel dem IT-Betrieb näher
- Agile und DevOps sind nicht gegensätzlich, sondern ergänzen sich

DevOps-Metriken und Erfolg

DORA-Metriken

DevOps Research and Assessment (DORA) bietet einen Standardsatz von Metriken zur Bewertung von DevOps-Erfolg:

1. **Bereitstellungshäufigkeit:** Durchschnittliche Anzahl der täglichen Codebereitstellungen in einer bestimmten Umgebung
2. **Vorlaufzeit für Änderungen:** Die Zeitspanne zwischen Annahme und Bereitstellung einer Änderung
3. **Zeit bis zur Wiederherstellung der Dienste:** Wie lange es dauert, den Dienst nach einem Fehler wiederherzustellen
4. **Fehlerquote bei Änderungen:** Wie häufig Implementierungen fehlschlagen

Leistungsklassen nach DORA

Teams werden anhand ihrer DORA-Metriken klassifiziert:

- **Elite Performer:** Mehrere Deployments pro Tag, Vorlaufzeit $<$ 1 Stunde, Wiederherstellungszeit $<$ 1 Stunde, Fehlerrate $<$ 5%
- **High Performer:** Zwischen einmal pro Tag und einmal pro Woche, Vorlaufzeit zwischen einem Tag und einer Woche
- **Medium Performer:** Zwischen einmal pro Woche und einmal pro Monat
- **Low Performer:** Weniger als einmal pro Monat, Vorlaufzeit $>$ 6 Monate

Software-Automatisierung

Arten der Software-Automatisierung

Eine Vielzahl von Aufgaben der Softwareentwicklung können automatisiert werden:

- **On-Demand:** Ein Skript ausführen oder eine Taste drücken
- **Scheduled:** Zu bestimmten Zeiten (z.B. nächtliche Builds)
- **Triggered:** Bei bestimmten Ereignissen (z.B. Git commit/push)

Typen der Automatisierung

- **Build Automation:** Kompilieren, Software-Paketierung, Erstellen von Dokumentation
- **Test Automation:** Automatisierte Unit-, Integration- und Akzeptanztests
- **Deployment Automation:** Automatisiertes Deployment in verschiedene Umgebungen
- **Operation Automation:** Infrastruktur-Provisionierung, Monitoring, Skalierung

Warum Automatisierung wichtig ist

Probleme ohne Automatisierung:

- Ës läuft auf meinem Computer! Probleme bei der lokalen Entwicklung
- Inkonsistente Versionierung
- Unregelmäßige Unit-Tests
- Unklarer Build-Status
- Undefinierte Abhängigkeiten
- Intransparentes Deployment
- Fehleranfälligkeit bei manueller Arbeit
- Langweilige Wiederholungsarbeit

Ziele der Automatisierung

- **Produktqualität verbessern:** Automatisierte Tests, Code-Auditing, dokumentierte Build-Historie
- **Markteinführungszeit verkürzen:** Schnellere Prozesse, unmittelbares Feedback, kürzere Innovationszyklen
- **Risiken minimieren:** Fortschritt nachweisen, fehlerhafte Builds frühzeitig finden, Abhängigkeiten von Schlüsselpersonen reduzieren

Software Automation Pipeline: CI/CD

Software Automation Pipeline

Automatisierung kann in jeder Phase des Softwareprozesses eingesetzt werden:

- Jeder Schritt kann nur fortgesetzt werden, wenn er mehrere Tests erfolgreich durchläuft
- Bei Fehlern werden die Verantwortlichen sofort informiert
- Der Prozess enthält Feedback-Schleifen für kontinuierliche Verbesserung

Stufen der Automatisierung

Je nach Automatisierungsgrad unterscheidet man:

- **Build Automation:** Entwicklung einzelner Komponenten und Unit-Tests, typischerweise lokal ausgeführt
- **Continuous Integration (CI):** Automatisierte Entwicklung, Testen und Integration von Komponenten sowie Integrationstests, typischerweise auf dem CI-Server
- **Continuous Delivery (CD):** Erzeugt auch Releases, deployt in die Staging-Umgebung und führt automatische Akzeptanztests aus; bereit für die Produktion, aber manueller Schritt für das Deployment
- **Continuous Deployment:** Automatisches Deployment in die Produktion nach erfolgreichen Akzeptanztests
- **DevOps:** Automatischer Betrieb des Produktionssystems (Konfigurationsmanagement, Infrastrukturbereitstellung, Backup, Monitoring, etc.)

Continuous Integration

Zentrale Praktiken der Continuous Integration

- **Versionskontrolle:** Alles in einem versionskontrollierten Main-Branch (Trunk oder Main) ablegen
- **Automatisierte Builds:** Build-Prozess automatisieren und selbsttestend gestalten
- **Häufige Integration:** Jeder pusht Commits täglich in den Main-Branch
- **Schnelle Fehlerbehebung:** Fehlerhafte Builds sofort beheben
- **Kurze Build-Zeiten:** Builds unter 10 Minuten halten

Feature Flags für unfertige Funktionen

Problem

Wie integriert man Code für unfertige Features, die noch nicht für Benutzer sichtbar sein sollen?

Lösung: Feature Flags

- Feature Flags sind Konfigurationsoptionen, die bestimmte Funktionen aktivieren oder deaktivieren
- Sie ermöglichen es, unfertigen Code in den Hauptzweig zu integrieren, ohne ihn den Benutzern zu zeigen
- Der Code wird erst aktiviert, wenn das Feature fertig ist

Implementierung

- Einfaches Boolean-Flag für An/Aus-Entscheidungen
- Konfigurierbar pro Umgebung (Entwicklung, Test, Produktion)
- Kann auch für A/B-Tests oder schrittweise Einführung verwendet werden

Feature Flag Implementierung

```
// Feature Flag Konfiguration
const featureFlags = {
  newUserInterface: false,
  advancedSearch: true,
  darkMode: true
};

// Verwendung im Code
function renderUserInterface() {
  if (featureFlags.newUserInterface) {
    // Neues UI rendern
    renderNewUI();
  } else {
    // Altes UI rendern
    renderLegacyUI();
  }
}

// Feature je nach Umgebung aktivieren
if (process.env.ENVIRONMENT === 'development') {
  featureFlags.newUserInterface = true;
}
```

Kontinuierliches Deployment

Deployment-Strategien

- **Blue/Green Deployment:** Zwei identische Produktionsumgebungen (Blue und Green), wobei nur eine aktiv ist. Nach dem Deployment auf die inaktive Umgebung wird der Traffic umgeleitet.
- **Canary Deployment:** Neue Version wird nur für einen kleinen Teil der Benutzer bereitgestellt, um Probleme früh zu erkennen, bevor die Version vollständig ausgerollt wird.
- **A/B-Testing:** Ähnlich wie Canary, aber mit Fokus auf Benutzerreaktionen statt technischer Probleme.

Automatisches Rollback

- Bei Problemen im Deployment ist schnelles Rollback entscheidend
- Automated Rollback: Automatische Rückkehr zur letzten bekannten guten Version
- Monitoring-Systeme können Alarme auslösen, die ein Rollback initiieren
- Feature Flags ermöglichen ein schnelles Äusschalten"problematischer Funktionen

Virtuelle Umgebungen

Virtualisierung und Containerisierung

- **Virtualisierung:** Technologien zur Erstellung virtueller Versionen physischer Ressourcen; eine VM enthält ein vollständiges Betriebssystem
- **Containerisierung:** Leichtgewichtige Isolierung von Anwendungen; Container teilen sich das Betriebssystem des Hosts und enthalten nur die Anwendung und ihre Abhängigkeiten
- **Docker:** Populäres Tool für Containerisierung
- **Kubernetes:** System zur Orchestrierung und Verwaltung von Containern

Infrastructure as Code (IaC)

- Verwaltung und Bereitstellung von IT-Infrastrukturen mithilfe von Code statt manueller Konfiguration
- Infrastrukturdateien werden in Versionskontrollsystemen gespeichert
- Ermöglicht reproduzierbare, konsistente Umgebungen
- Entwickler werden stärker in die Konfiguration einbezogen
- Ops-Teams werden früher in den Entwicklungsprozess einbezogen
- Populäre Tools: Terraform, AWS CloudFormation, Ansible

Microservices-Architektur

- Zerlegt eine monolithische Anwendung in lose gekoppelte Dienste
- Jeder Dienst konzentriert sich auf eine Geschäftsfähigkeit
- Kommunikation über definierte APIs (meist RESTful)
- Vorteile: unabhängige Entwicklung und Skalierung, technologische Flexibilität
- Herausforderungen: Komplexität der Verteilung, Service Discovery, Fehlertoleranz

GitHub Actions und CI/CD

GitHub Actions

GitHub Actions ist eine CI/CD-Plattform, mit der man Build-, Test- und Deployment-Pipelines automatisieren kann:

- **Workflows:** YAML-Dateien, die die Pipeline definieren
- **Events:** Auslöser wie Push, Pull Request oder zeitgesteuerte Ereignisse
- **Jobs:** Aufgaben, die auf Runnern ausgeführt werden
- **Actions:** Wiederverwendbare Bausteine für Workflows
- **Runners:** Server, die die Jobs ausführen

Einfacher GitHub Actions Workflow Language = YAML

```yaml
name: CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up Node.js
      uses: actions/setup-node@v2
      with:
        node-version: '14'

    - name: Install dependencies
      run: npm ci

    - name: Run tests
      run: npm test

    - name: Build
      run: npm run build
```

Zusammenfassung

DevOps Vorteile

DevOps bietet zahlreiche Vorteile gegenüber traditionellen Entwicklungsansätzen:

- Schnellere Markteinführung durch automatisierte Prozesse
- Höhere Softwarequalität durch frühe und kontinuierliche Tests
- Verbesserte Zusammenarbeit zwischen Entwicklung und Betrieb
- Geringere Fehlerrate bei Deployments
- Schnellere Erholung von Ausfällen
- Kontinuierliche Verbesserung durch Feedback-Schleifen
- Erhöhte Skalierbarkeit und Zuverlässigkeit von Systemen

Scrum

Einführung in Scrum

Was ist Scrum?

Scrum ist ein agiler Prozessrahmen für das Projektmanagement, der sich auf die Lieferung des höchstmöglichen Wertes für das Fach konzentriert:

- Regelmäßige und wiederholte Inspektion des aktuellen Standes lauffähiger Software
- Das Fach setzt die Prioritäten, Teams organisieren sich selbst
- Alle 2-4 Wochen können alle Beteiligten lauffähige Software begutachten
- Iterative und inkrementelle Entwicklung in kurzen Zyklen (Sprints)

Herkunft von Scrum

- Der Begriff SScrumßtammt aus dem Rugby: Ein strukturiertes Gedränge, bei dem die Spieler eng zusammenarbeiten, um den Ball zu kontrollieren
- Erstmals 1986 von Hirotaka Takeuchi und Ikujiro Nonaka als Produktentwicklungsansatz im Harvard Business Review beschrieben
- 1993: Jeff Sutherland entwickelte die ersten Scrum-Ansätze bei Easel Corporation
- 1995: Ken Schwaber formalisierte den Prozess
- 1996: Gemeinsame Präsentation von Sutherland und Schwaber bei der OOPSLA-Konferenz
- 2001: Scrum wurde als eine der Methoden im Agilen Manifest verankert

Eigenschaften von Scrum

- Selbstorganisierte Teams
- Produktfortschritt in einer Serie von SSprintsïm Monats-Rhythmus
- Anforderungen werden als Artefakte in einem "Backlogërfasst
- Keine speziellen Engineering-Praktiken vorgegeben (im Gegensatz zu XP)
- Empirische Prozesskontrolle: Inspektion, Adaption und Transparenz

Scrum Rollen

Product Owner

Der Product Owner ist verantwortlich für die Maximierung des Produktwerts:

- Definiert die Features des Produkts
- Entscheidet über Datum und Inhalt eines Releases
- Ist verantwortlich für die Profitabilität (ROI) des Produkts
- Priorisiert Features basierend auf ihrem Marktwert
- Passt Features und Prioritäten bei Bedarf für jeden Sprint an
- Akzeptiert Arbeitsresultate oder weist sie zurück

Scrum Master

Der Scrum Master ist ein Dienstleister für das Team und die Organisation:

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

Entwicklungsteam

Das Entwicklungsteam ist für die Umsetzung der Produktanforderungen verantwortlich:

- Typischerweise 5-9 Personen
- Cross-Functional: Programmierer, Tester, Designer etc.
- Ideale Auslastung: 100% für das Projekt
- Selbstorganisierend in Bezug auf die Arbeit
- Zusammensetzung sollte nur zwischen den Sprints wechseln

Guter Scrum Master werden

Verantwortlichkeiten verstehen

- Scrum-Prozess am Laufen halten
- Für Machtbalance zwischen Product Owner, Team und Management sorgen
- Das Team schützen und moderieren
- Bei der Organisation helfen
- Team auf Sprint-Ziele fokussieren

Hilfestellung geben

- Sprint-Ziele erreichen helfen
- Mit Product Owner zusammenarbeiten
- Hindernisse beseitigen
- Transparenz fördern
- Team zu einem High-Performance-Team entwickeln

Selbstorganisation fördern

- Selbstorganisation ermutigen und schützen
- Team auf geschäftsorientierte Entwicklung fokussieren
- Teambildung unterstützen durch Nutzen individueller Fähigkeiten
- Feedback-Kultur fördern
- Zur Selbsthilfe befähigen

Scrum Ereignisse (Events)

Sprint

Ein Sprint ist ein zeitlich begrenzter Entwicklungszyklus:

- Zeitboxen von typischerweise 2-4 Wochen
- Konstante Dauer für einen besseren Rhythmus
- Das Produkt wird während eines Sprints designed, codiert und getestet
- Keine Änderungen am Sprint-Ziel während des Sprints
- Die Länge der Sprints hängt davon ab, wie lange Änderungen zurückgehalten werden können

Sprint Planning

Im Sprint Planning wird festgelegt, was im kommenden Sprint umgesetzt werden soll:

- Das Team wählt Items aus dem Product Backlog aus, die es innerhalb eines Sprints umsetzen kann
- Ein Sprint Backlog wird definiert
- Tasks werden identifiziert und geschätzt (typischerweise 1-16 Stunden)
- Ein Sprint Goal wird formuliert - ein kurzes Statement zur Zielformulierung
- Die Planung erfolgt gemeinschaftlich, nicht allein durch den Scrum Master

Daily Scrum

Das Daily Scrum ist ein tägliches 15-minütiges Meeting des Entwicklungsteams:

- Täglich zur gleichen Zeit, am selben Ort
- Stehend durchgeführt (Stand-up)
- Jedes Teammitglied beantwortet drei Fragen:
  1. Was habe ich gestern getan?
  2. Was werde ich heute tun?
  3. Gibt es Hindernisse?
- Keine Problemlösungen während des Meetings
- Alle können teilnehmen, aber nur das Team, der Scrum Master und der Product Owner dürfen sprechen

Sprint Review

Im Sprint Review wird das Inkrement präsentiert und Feedback eingeholt:

- Das Team präsentiert die Ergebnisse des Sprints als Demo
- Informell, keine PowerPoint-Präsentationen
- Maximal 2 Stunden Vorbereitungszeit
- Alle Stakeholder nehmen teil
- Feedback fließt in die Planung des nächsten Sprints ein
- Bei einem 4-Wochen-Sprint dauert das Review etwa 4 Stunden

Sprint Retrospektive

In der Sprint Retrospektive reflektiert das Team über seine Arbeitsweise:

- Periodische Prüfung dessen, was gut läuft und was nicht
- Typischerweise 15-30 Minuten nach dem Sprint Review
- Alle nehmen teil: Scrum Master, Product Owner, Team
- Fokus auf kontinuierliche Verbesserung des Prozesses
- Konkrete Verbesserungsmaßnahmen für den nächsten Sprint werden identifiziert

Eine typische Sprint Retrospektive verwendet das SStart-Stop-ContinueFormat:

- **Start:** Was sollten wir beginnen zu tun?
- **Stop:** Was sollten wir aufhören zu tun?
- **Continue:** Was sollten wir weiterhin tun?

Das Team sammelt Punkte zu jeder Kategorie und entscheidet gemeinsam, welche Maßnahmen im nächsten Sprint umgesetzt werden sollen.

Scrum Artefakte

Product Backlog

Das Product Backlog ist eine priorisierte Liste aller gewünschten Produktfunktionen:

- Enthält alle Anforderungen und auszuführenden Arbeiten im Projekt
- Idealerweise so formuliert, dass der Wertbeitrag für den Benutzer erkennbar ist
- Wird vom Product Owner priorisiert
- Ist dynamisch und wird kontinuierlich weiterentwickelt
- Wird zu Beginn jedes Sprints neu priorisiert

Sprint Backlog

Das Sprint Backlog enthält die für den aktuellen Sprint ausgewählten Backlog-Items:

- Individuen wählen ihre Arbeit selbst aus (keine Zuweisung)
- Die Schätzung des verbleibenden Aufwands wird täglich aktualisiert
- Jedes Teammitglied kann das Sprint Backlog anpassen
- Bei unklaren Anforderungen werden größere Backlog-Items für die spätere Klärung geschaffen

Inkrement

Das Inkrement ist die Summe aller Product Backlog-Items, die während eines Sprints fertiggestellt wurden:

- Muss "Doneßein, d.h. den Akzeptanzkriterien entsprechen
- Muss eine potentiell auslieferbare Version des Produkts darstellen
- Wird im Sprint Review präsentiert

Task Board

Das Task Board visualisiert den aktuellen Stand aller Aufgaben im Sprint:

- Typische Spalten: "To Do", Ïn Progress", "Done"
- Jeder Task wird als Karte dargestellt
- Teammitglieder verschieben ihre Tasks je nach Fortschritt
- Bietet Transparenz über den aktuellen Sprint-Status
- Zeigt mögliche Blockaden und Engpässe auf

Sprint Burndown Chart

Das Sprint Burndown Chart zeigt den verbleibenden Aufwand im Sprint:

- X-Achse: Tage im Sprint
- Y-Achse: Verbleibender Aufwand (in Stunden oder Story Points)
- Ideale Linie zeigt gleichmäßigen Fortschritt
- Tatsächliche Linie zeigt realen Fortschritt
- Warnsignale: Flache Linie (kein Fortschritt) oder steigende Linie (mehr Arbeit entdeckt)

Definition of Done und Ready

Definition of Done (DoD)

Die Definition of Done legt fest, wann eine User Story als abgeschlossen gilt:

- Gegenseitig akzeptiertes Übereinkommen aller Beteiligten
- Konform mit den Governance-Vorgaben der Organisation
- Wird für User Stories und Sprints definiert
- Verhindert technische Schulden und Illusionen über den Projektfortschritt
- Wird vom gesamten Team inkl. Product Owner definiert
- Wird vor jedem Sprint überprüft und bei Bedarf angepasst

Typische Definition of Done für eine User Story:

- Unit-Tests bestehen mit mindestens 85% Abdeckung
- Ausreichend negative Tests wurden geschrieben
- Code wurde überprüft (oder als Pair Programming erstellt)
- Coding-Standards sind erfüllt
- CI/CD ist implementiert
- Code wurde refaktoriert
- UAT-Tests werden bestanden
- Nicht-funktionale Tests werden bestanden
- Erforderliche Dokumentation ist fertiggestellt

Definition of Ready

Die Definition of Ready legt fest, wann eine User Story bereit für die Aufnahme in einen Sprint ist:

- Die User Story ist klar definiert
- Akzeptanzkriterien sind festgelegt
- Abhängigkeiten sind identifiziert
- Die Schätzung durch das Team liegt vor
- UX-Artefakte wurden akzeptiert
- Nicht-funktionale Anforderungen sind definiert
- Die abnehmende Person ist benannt
- Das Team hat eine klare Vorstellung davon, was zu zeigen ist

Skalierung von Scrum

Scrum of Scrums

Bei größeren Projekten mit mehreren Scrum-Teams wird Scrum of Scrums eingesetzt:

- Vertreter jedes Teams treffen sich regelmäßig
- Koordination und Abstimmung zwischen Teams
- Identifikation von teamübergreifenden Abhängigkeiten
- Lösung von Konflikten und Blockaden

Scaled Agile Framework (SAFe)

SAFe ist ein Framework für die Skalierung agiler Methoden auf Unternehmensebene:

- Verschiedene Konfigurationen je nach Unternehmensgröße
- Integriert Scrum mit Lean und DevOps
- Organisiert Teams in Ägile Release Trains"(ARTs)
- Definiert zusätzliche Rollen und Events für die Koordination
- Berücksichtigt Portfoliomanagement und Unternehmensstrategie

Scrum-Werte

Die fünf Scrum-Werte

- **Commitment:** Das Team verpflichtet sich, seine Ziele zu erreichen
- **Fokus:** Konzentration auf die Sprint-Arbeit und die Scrum-Ziele
- **Offenheit:** Transparenz über Arbeit und Herausforderungen
- **Respekt:** Gegenseitiger Respekt für die Fähigkeiten und Unabhängigkeit
- **Mut:** Mut, das Richtige zu tun und schwierige Probleme anzugehen

Scrum-Erfolgsformel

Für erfolgreiche Scrum-Implementierungen:

- Transparenz: Alle relevanten Aspekte des Prozesses müssen für alle Beteiligten sichtbar sein
- Inspektion: Regelmäßige Überprüfung von Artefakten und Fortschritt
- Anpassung: Prozess oder Material bei Abweichungen anpassen
- Selbstorganisation: Teams entscheiden selbst, wie sie Arbeit erledigen
- Timeboxing: Strikte Zeitbegrenzungen für alle Aktivitäten
- Inkrementelle Lieferung: Regelmäßige Bereitstellung von Produktinkrementen

Software Craft und Pyramid of Agile Competencies

Software Craft(smanship)

Software Craftsmanship

Software Craftsmanship ist eine Bewegung in der Softwareentwicklung, die sich mit der Arbeitsweise und Wahrnehmung des Berufsstandes der Softwareentwickler beschäftigt:

- Ziel ist es, Softwareentwicklung als eigenständige Profession und als Handwerk (nicht nur als Ingenieursdisziplin) wahrzunehmen
- Fokus auf Professionalität, kontinuierliches Lernen und Qualität
- Wertschätzung für technische Exzellenz und Clean Code
- Reaktion auf die Vernachlässigung technischer Praktiken im agilen Umfeld

Die Entstehung der Software Craftsmanship Bewegung

- 2001: Agiles Manifest fokussiert stark auf den Projektprozess, weniger auf technische Aspekte
- 2008: Software Craftsmanship Summit mit Micah Martin
- 2009: Entstehung des Software Craftsmanship Manifesto
- 2009: Erste Software Craftsmanship Konferenzen in USA und UK
- Ab 2010: Gründung lokaler Communities weltweit
- Heute: SoCraTes (Software Craftsmanship and Testing) Konferenzen in vielen Ländern

Manifesto for Software Craftsmanship

Als Ergänzung zum Agilen Manifest betont das Software Craftsmanship Manifesto:

- Nicht nur funktionierende Software, sondern auch **handwerklich gut gemachte Software**
- Nicht nur auf Veränderung reagieren, sondern auch **kontinuierlich Wert hinzufügen**
- Nicht nur Individuen und Interaktionen, sondern auch eine **Gemeinschaft von Profis**
- Nicht nur Zusammenarbeit mit dem Kunden, sondern auch **produktive Partnerschaften**

Motto der Craftsmanship-Bewegung

"Wir sind es leid, Mist zu schreiben."

- Wir 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 man uns zwingt, unprofessionell zu handeln

Üben wie ein Software Crafter

Craftsmanship Prinzipien

Die vier Kernprinzipien der Software Craftsmanship:

- **Individuals & Interactions:** Voneinander lernen
- **Clean Code:** Codequalität und Lesbarkeit priorisieren
- **Lifelong Learning:** Kontinuierliche Weiterbildung
- **Continuous Improvement:** Ständige Verbesserung durch Übung

Möglichkeiten zum Üben

Software Crafter nutzen verschiedene Formate zum Üben und Verbessern ihrer Fähigkeiten:

- Code Katas
- Coding Dojos
- Code Retreats
- Clean Code Developer
- Code Koans
- Pair Programming
- Mob Programming

Coding Dojo

Ein Haufen Coder kommt zusammen, programmiert, lernt und hat Spaß Emily Bache
Ein Coding Dojo ist eine sichere Übungsumgebung für Entwickler:

- Keine Manager, keine Deadlines
- Fokus auf das Lernen, nicht auf das Ergebnis
- Spezielle Übungsformen zur Entwicklung schwieriger Fähigkeiten
- Gemeinsames Programmieren und Reflektieren

Dojo-Prinzipien

- Design kann nicht ohne Code besprochen werden, Code kann nicht ohne Tests gezeigt werden
- Mit eigenen Erfahrungen kommen
- Bereitschaft, neu zu lernen
- Entschleunigen
- Sich auf die Suche nach einem Meister machen
- Sich einem Meister unterwerfen
- Einen Untergebenen anleiten

Code Kata

- "Kataïst ein japanisches Wort für ein detailliert choreografiertes Bewegungsmuster
- In der Softwareentwicklung: Eine definierte Programmieraufgabe, die wiederholt wird, um bestimmte Fähigkeiten zu üben
- Dave Thomas: Ëntwickler sollten immer wieder an kleinen, nicht berufsbezogenen Code-Basen üben, damit sie ihren Beruf wie Musiker beherrschen"

Kata-Typen

- **Function Katas:** Einfache Algorithmen (FizzBuzz, Roman Numbers)
- **Class Katas:** Objektdesign (Bowling, Stack)
- **Library Katas:** Programmieren gegen Schnittstellen (App Login)
- **Application Katas:** Komplette Anwendungen (Tic Tac Toe)
- **Architecture Katas:** Architekturentwürfe (URL Shortener)
- **Agile Katas:** Übungen für agile Praktiken

FizzBuzz Kata

```java
/* FizzBuzz Kata:
 * Schreibe ein Programm, das die Zahlen von 1 bis 100 ausgibt, aber:
 * - fuer Vielfache von 3 gibt es "Fizz" aus
 * - fuer Vielfache von 5 gibt es "Buzz" aus
 * - fuer Vielfache von 3 und 5 gibt es "FizzBuzz" aus
 */
public class FizzBuzz {
    public static void main(String[] args) {
        for (int i = 1; i <= 100; i++) {
            if (i % 3 == 0 && i % 5 == 0) {
                System.out.println("FizzBuzz");
            } else if (i % 3 == 0) {
                System.out.println("Fizz");
            } else if (i % 5 == 0) {
                System.out.println("Buzz");
            } else {
                System.out.println(i);
            }
        }
    }
}
```

Merkmale einer Code-Kata

- **Definition:** Ein definierter Lösungsweg einer Code-Übung, der viele Male wiederholt wird
- **Dauer:** Meist kurz (30-60 Minuten), um regelmäßiges Üben zu ermöglichen
- **Fokus:** Nicht die richtige Antwort zu finden, sondern der Lernprozess
- **TDD:** Test-Driven Development wird meist als Standard-Pattern verwendet
- **Wiederholung:** Die gleiche Übung wird mehrfach durchgeführt, um bei jeder Wiederholung kleine Verbesserungen zu erzielen

Pyramid of Agile Competencies

Pyramid of Agile Competencies

Die Pyramid of Agile Competencies ist ein didaktisches Konzept, das die für agile Softwareentwicklung erforderlichen Kompetenzen auf drei Ebenen darstellt:

- **Agile Values** (Basis der Pyramide)
- **Collaboration Practices** (mittlere Ebene)
- **Technical Practices** (Spitze der Pyramide)

Alle drei Ebenen sind für erfolgreiche agile Teams notwendig.

Agile Values

Die Basis der Pyramide bilden gemeinsame Werte und Einstellungen:

- **Transparenz und Offenheit:** Werden in der agilen Softwareentwicklung großgeschrieben, um sowohl die Organisation als auch den Kunden über Fortschritte zu informieren und schnelles Feedback zu erhalten
- **Organizational Culture:** Es gibt drei Möglichkeiten:
  - Agiles Team, agile Organisation und agiles Unternehmen
  - Agiles Team und agile Organisation, nicht-agiles Unternehmen
  - Agiles Team, nicht-agile Organisation und nicht-agiles Unternehmen
- **Software Craftsmanship:** Mehr als nur Technik - eine Einstellung und Haltung zur Softwareentwicklung

Collaboration Practices

Die mittlere Ebene umfasst Praktiken zur Zusammenarbeit:

- **Customer and Requirements:** Ïntensive und häufige Kommunikation mit dem Kunden ist von größter Bedeutung"
- **Agile Champion:** Eine Person, die Agilität im Team fördert und vorantreibt
  - Führt und inspiriert Agilität
  - Hilft zu definieren, welche Veränderungen notwendig sind
  - Überzeugt andere, die Veränderung zu unterstützen
  - Hilft zu zeigen, dass Veränderung stattfindet und gute Ergebnisse liefert
  - Verhindert "Cowboy-Agileünd Rückfälle zu früheren Ansätzen
- **Collaboration and Communication:** Intensive und offene Kommunikation zwischen allen Beteiligten ist ein Schlüsselelement für erfolgreiche agile Projekte
  - Kommunikation zwischen Teammitgliedern
  - Kommunikation zwischen Team und Kunde/Endnutzer
  - Kommunikation zwischen Team und Management

Technical Practices

Die Spitze der Pyramide bilden technische Praktiken:

- **Testing:** Automatisierte Tests auf Unit-Ebene sind gut etabliert und werden als absolutes Muss für eine gute Softwarequalität angesehen
- **Continuous Integration:** Wird als absolutes Muss angesehen, um Software mit hoher Frequenz liefern zu können
- **Clean Code:** Kontinuierliche Aufmerksamkeit für guten Code von Anfang an wird als immer wichtiger angesehen

Integration der Pyramid of Agile Competencies im Team

Agile Values etablieren

- Transparente Kommunikation fördern
- Respektvolle Feedback-Kultur aufbauen
- Kontinuierliches Lernen als Wert verankern
- Gemeinsame Verantwortung für Qualität entwickeln

Collaboration Practices implementieren

- Regelmäßige Kundenkommunikation strukturieren
- Cross-funktionale Zusammenarbeit fördern
- Agile Champions identifizieren und unterstützen
- Effektive Kommunikationskanäle etablieren

Technical Practices stärken

- Test-Driven Development als Standard einführen
- Continuous Integration und Delivery automatisieren
- Code Reviews und Pair Programming etablieren
- Regelmäßige Refactoring-Sessions durchführen
- Technische Schulden aktiv managen

Agile Software-Entwicklung in der Praxis

Moderne Unternehmen und Software Craft

Unternehmen, die Software Craft ernst nehmen:

- Bieten Coaching on the Job
- Organisieren interne Un-Konferenzen
- Richten Coding Dojos oder Code Retreats aus
- Unterstützen Lernen mit eigenem Budget (für Bücher, Konferenzen, Schulungen)
- Fördern die Software-Craft-Bewegung
- Ermöglichen Zeit für technische Verbesserungen und Refactoring

Community of Practice

Vorteile der Teilnahme an einer Software Craft Community:

- Miteinander und voneinander lernen
- Außenperspektive zu eigenen Projekten erhalten
- Fragen stellen und Ideen herausfordern lassen
- Gleichgesinnte mit ähnlicher Leidenschaft treffen
- Berufliche Weiterentwicklung

Clean Code als Grundlage

Clean Code basiert auf den Ideen von Robert C. Martin und ist ein wesentlicher Bestandteil von Software Craftsmanship:

- Verständlicher, lesbarer Code
- Einfache Designprinzipien
- Kontinuierliche Verbesserung durch Refactoring
- Sinnvolle Benennung von Variablen, Methoden und Klassen
- Kleine, fokussierte Funktionen mit einer Verantwortlichkeit
- Automatisierte Tests als Dokumentation und Sicherheitsnetz

Integration der drei Ebenen

Für erfolgreiche agile Teams ist die Balance und Integration aller drei Ebenen der Pyramid of Agile Competencies entscheidend:

- Agile Values ohne technische Praktiken führen zu schlechter Codequalität
- Technical Practices ohne Collaboration Practices führen zu Lösungen, die nicht den Kundenbedürfnissen entsprechen
- Collaboration Practices ohne Agile Values führen zu Oberflächlichkeit und mangelnder Nachhaltigkeit
- Nur durch die Kombination aller drei Ebenen entsteht echte Agilität

Advanced Scrum & SAFe

Sprint Review

Der Sprint Review

Der Sprint Review ist ein kollaboratives Arbeitsmeeting, das am Ende des Sprints stattfindet:

- Keine reine Demo, sondern ein interaktives Meeting
- Das Scrum-Team zeigt die Arbeitsergebnisse des Sprints
- Das Feedback steht im Zentrum
- Das Feedback wird als Input für den nächsten Sprint verwendet
- Es wird lauffähige Software gezeigt (keine PowerPoint-Präsentationen)
- Zeitbox: 4 Stunden für einen 4-Wochen-Sprint (proportional weniger für kürzere Sprints)
- Das gesamte Scrum-Team und Stakeholder nehmen teil

Mechanismen eines Sprint Reviews

Der Sprint Review folgt einer strukturierten Vorgehensweise:

- Begrüßung und Einführung durch den Scrum Master
- Vorstellung des Sprint-Ziels durch den Product Owner
- Demonstration der fertigen Funktionalität durch das Entwicklungsteam
- Sammlung von Feedback und Diskussion mit allen Teilnehmern
- Überprüfung des Product Backlogs und Ausblick auf die nächsten Sprints
- Anpassung der Release-Planung basierend auf dem aktuellen Status

Sprint Review Checkliste

Vorbereitung

- Sprint-Ziel und User Stories überprüfen
- Produktvision, Roadmap und Release-Plan bereithalten
- Story Map für den Kontext vorbereiten
- Kapazität und tatsächlichen Fortschritt (Story Points) dokumentieren
- Demonstrationsfähige Funktionalitäten identifizieren

Durchführung

- Für jede User Story:
  - Priorisierung erläutern
  - Verständnisfragen klären
  - Umsetzung demonstrieren
- Backlog-Verifikation durchführen
- Auf Konflikte, Definition of Done und fehlende Backlog-Einträge achten
- Risiken besprechen

Abschluss

- Commitment für den nächsten Sprint einholen
- Tooling und Prozessverbesserungen diskutieren
- Nächste Schritte und offene Punkte festhalten

Sprint Retrospektive

Die Sprint Retrospektive

Die Sprint Retrospektive ist eine Gelegenheit für das Scrum-Team, sich selbst zu inspizieren und zu verbessern:

- Diskussion über den Scrum-Prozess, das Verhalten des Teams und eingesetzte Tools
- Erweiterung der "Definition of Done"
- Ziel: Umsetzbare Verbesserungen identifizieren, die das Team im nächsten Sprint umsetzen kann
- Findet nach dem Sprint Review und vor dem nächsten Sprint Planning statt
- Zeitbox: 3 Stunden für einen 4-Wochen-Sprint (proportional weniger für kürzere Sprints)
- Das vollständige Scrum-Team nimmt teil

Leitmotiv der Retrospektive

"Wir gehen davon aus, dass alle Beteiligten den bestmöglichen Einsatz im gegebenen Rahmen (Wissensstand, Ressourcen, Fähigkeiten) geleistet haben, unabhängig davon, was im Rahmen einer Retrospektive entdeckt worden ist."

- Schafft eine sichere Umgebung für offene Diskussionen
- Fokus auf systemische Probleme statt auf Schuldzuweisungen
- Ermöglicht konstruktives Feedback

Typische Fragestellungen in Retrospektiven

- Was lief gut während des Sprints?
- Was lief nicht so gut?
- Was können wir anders oder besser machen?
- Welche konkreten Maßnahmen nehmen wir uns für den nächsten Sprint vor?
- Alternative Formate:
  - Start-Stop-Continue: Was sollten wir beginnen/aufhören/weitermachen?
  - Glad-Sad-Mad: Was macht uns froh/traurig/wütend?
  - 4Ls: Liked, Learned, Lacked, Longed For
  - Segelboot: Wind (Antrieb), Anker (Bremsen), Felsen (Risiken), Sonne (Positives)

Beispiel für ein Start-Stop-Continue Retrospektive-Ergebnis:

Start:

- Tägliches Pair Programming für komplexe Aufgaben
- Test-Driven Development konsequent anwenden
- Mehr technische Sessions zur Wissensverteilung

Stop:

- Zu viele parallele Aufgaben pro Entwickler
- Lange Meetings ohne klare Agenda
- Ungetesteten Code integrieren

Continue:

- Daily Standup pünktlich und fokussiert durchführen
- Code Reviews vor Integration
- Regelmäßige Abstimmung mit Product Owner

Definition of Done (DoD)

Definition of Done

Die "Definition of Done"beschreibt Vollständigkeit im Sinne eines gegenseitig akzeptierten Übereinkommens aller Beteiligten, das konform zu den Governance-Vorgaben der Organisation ist.

- Bezieht sich auf User Stories und Sprints
- Verhindert technische Schulden
- Schafft klare Erwartungen bezüglich Produktqualität
- Wird vom gesamten Team (inklusive Product Owner) definiert
- Wird schriftlich festgehalten und kontinuierlich verbessert

Konsequenzen bei fehlender DoD

Ohne klare Definition of Done drohen:

- Technische Schulden
- Nichterledigte Arbeit, die sich auftürmt
- Illusionen bezüglich des Projektfortschritts (verfälschte Velocity)
- Unvorhersehbares Lieferdatum
- Team över-commitmentïn Bezug auf die Arbeit, die während eines Sprints erledigt werden kann
- Überraschende (unfertige) Ergebnisse während des Sprint Reviews

Beispiel für eine Definition of Done für eine User Story:

1. Unit-Tests bestehen und Abdeckung entspricht dem Standard (85% oder mehr)
2. Ausreichend negative Unit-Tests wurden geschrieben (mehr negative als positive)
3. Code ist überprüft (oder paarweise programmiert)
4. Coding-Standards sind erfüllt
5. CI/CD ist implementiert (automatisierter Build, Einsatz und Test)
6. Code ist refaktorisiert
7. UAT-Tests werden bestanden (Testfallanforderungen)
8. Nicht-funktionale Tests werden bestanden (Skalierbarkeit, Zuverlässigkeit, Sicherheit)
9. Erforderliche Dokumentation ist fertiggestellt
10. Performance-Tests zeigen akzeptable Reaktionszeiten

Definition of Ready

Ergänzend zur Definition of Done definiert die "Definition of Ready", wann eine User Story bereit für die Aufnahme in einen Sprint ist:

1. Die User Story ist definiert
2. Die Akzeptanzkriterien sind definiert
3. Die Abhängigkeiten der User Story sind identifiziert
4. Die Schätzung durch das Team liegt vor
5. Das Team hat die Artefakte des Nutzererlebnisses akzeptiert
6. Performance und andere nichtfunktionale Eigenschaften sind definiert, falls notwendig
7. Die Person, die die User Story abnimmt, ist definiert
8. Das Team hat eine klare Vorstellung, was gezeigt werden muss

Scaled Agile Framework (SAFe)

Was ist SAFe?

Das Scaled Agile Framework (SAFe) ist ein Framework zur Skalierung agiler Methoden auf Unternehmensebene:

- Kombiniert Scrum, Kanban, XP und Lean-Prinzipien
- Eignet sich für komplexe, große Organisationen mit vielen Teams
- Koordiniert die Arbeit mehrerer agiler Teams
- Verbindet Geschäftsstrategie mit Softwareentwicklung
- Unterstützt die agile Transformation auf allen Organisationsebenen

Warum SAFe?

Ïmmer mehr große Unternehmen und Branchen werden mit Software betrieben und als Online-Dienste angeboten - von der Filmindustrie über die Landwirtschaft bis hin zur nationalen Verteidigung"
Da Software als Produktbestandteil immer wichtiger wird, müssen Organisationen angepasst werden:

- Effizienz und Stabilität einer bewährten Aufbauorganisation kombinieren mit
- der Innovationsgeschwindigkeit einer agilen Vorgehensweise

Duales System für agile Organisationen

SAFe fördert ein duales Betriebssystem, das zwei Netzwerktypen kombiniert:

- **Wertschöpfungs-Netzwerk:** Agile, kundenorientierte Strukturen mit schnellen Entscheidungswegen
- **Funktionale Hierarchie:** Effiziente, stabile Organisationsstruktur

Die Werte von SAFe

SAFe basiert auf vier Grundwerten:

1. **Alignment:** Die Mission wird kommuniziert, indem die Portfoliostrategie und Lösungsvision festgelegt und die Wertschöpfung abgestimmt wird
2. **Built-in quality:** Es wird eine Umgebung geschaffen, in der integrierte Qualität zum Standard wird
3. **Transparency:** Förderung der Visualisierung aller relevanten Arbeiten und Schaffung eines Umfelds, in dem "die Fakten immer freundlich sind"
4. **Program execution:** Führungskräfte nehmen als Business Owner an der PI-Planung und -Ausführung teil

Die 10 SAFe-Prinzipien

1. Nehmen Sie eine wirtschaftliche Sicht ein
2. Wenden Sie Systemdenken an
3. Nehmen Sie Variabilität an; halten Sie Möglichkeiten offen
4. Machen Sie inkrementelle Fortschritte mit schnellen, integrierten Lernzyklen
5. Bauen Sie Meilensteine auf objektiven Bewertungen funktionierender Systeme auf
6. Visualisieren und begrenzen Sie die Produktionszeit (Work-in-Process), reduzieren Sie die Chargengröße und halten Sie die Wartezeiten kurz
7. Etablieren Sie einen Arbeitsrhythmus und synchronisieren Sie diesen durch bereichsübergreifende Planung
8. Setzen Sie die intrinsische Motivation der Wissensarbeiter frei
9. Dezentralisieren Sie die Entscheidungsfindung
10. Organisieren Sie sich rund um die Wertschöpfung

Die Ebenen von SAFe

SAFe definiert verschiedene Konfigurationen mit unterschiedlichen Ebenen:

- **Team-Ebene:** Agile Teams (Scrum oder Kanban)
- **Programm-Ebene:** Agile Release Trains (ARTs)
- **Large Solution-Ebene:** Koordination mehrerer ARTs
- **Portfolio-Ebene:** Strategische Ausrichtung und Investitionsentscheidungen
- **Enterprise-Ebene:** Unternehmensweite Agilität

Agile Release Train (ART)

Ein Agile Release Train (ART) ist ein zentrales Element in SAFe:

- Ein langlebiges Team aus agilen Teams
- Arbeitet mit einer gemeinsamen Vision und Richtung
- Besteht typischerweise aus 5-12 agilen Teams (50-125 Personen)
- Entwickelt und liefert Lösungen inkrementell
- Arbeitet in synchronisierten Zeitboxen (Program Increments, PI)
- Organisiert rund um Wertströme statt funktionaler Abteilungen

SAFe-Rollen

SAFe definiert zusätzliche Rollen zu den bekannten Scrum-Rollen:

- **Product Manager:** Verantwortet das übergeordnete Product Backlog (Program Backlog)
- **System Architect:** Zuständig für die Systemgestaltung auf höherer Ebene
- **Release Train Engineer (RTE):** Ähnlich einem Scrum Master auf ART-Ebene, kümmert sich um Zusammenarbeit und Abhängigkeiten zwischen Teams
- **Business Owner:** Behält den Blick auf ROI und Wertmaximierung auf Systemebene
- Zusätzlich die bekannten Scrum-Rollen (Product Owner, Scrum Master, Teams)

Ein typisches SAFe-Szenario könnte wie folgt aussehen:
Ein Unternehmen entwickelt eine große Subscription Billing Platform mit mehreren integrierten Komponenten. Die Lösung wird als SAFe-Implementierung organisiert:

- 8 Scrum-Teams bilden einen Agile Release Train
- Ein Product Manager definiert die übergreifende Produkt-Roadmap
- Release Train Engineer koordiniert teamübergreifende Abhängigkeiten
- System Architect sorgt für konsistente technische Visionen
- PI-Planung alle 10 Wochen mit allen Teams gemeinsam
- Synchronisierte 2-Wochen-Sprints für alle Teams
- Regelmäßige System Demos zeigen den integrierten Fortschritt

Architecture Patterns

Einführung in Architektur-Muster

Was sind Architektur-Patterns?

Architektur-Patterns (Architekturmuster) sind bewährte Lösungen für wiederkehrende Architekturprobleme in der Softwareentwicklung:

- Dokumentierte, erprobte Lösungsansätze für typische Probleme
- Beschreiben die Struktur, das Verhalten und die Interaktion von Komponenten
- Höhere Abstraktionsebene als Design Patterns (GoF)
- Befassen sich mit der übergreifenden Systemorganisation
- Helfen bei grundlegenden Architekturentscheidungen

Warum Architektur-Patterns wichtig sind

Architektur-Patterns bieten zahlreiche Vorteile:

- Bewährte Lösungen für komplexe Probleme
- Gemeinsame Sprache für Architekturentscheidungen
- Reduzieren technische Risiken durch erprobte Ansätze
- Verbessern die Wartbarkeit und Erweiterbarkeit von Systemen
- Erleichtern die Einarbeitung neuer Teammitglieder
- Unterstützen die Dokumentation der Systemarchitektur

CQRS (Command Query Responsibility Segregation)

CQRS - Command Query Responsibility Segregation

CQRS ist ein Architekturmuster, das Lese- und Schreiboperationen in einem System trennt:

- **Commands:** Ändern den Systemzustand, liefern kein Ergebnis zurück
- **Queries:** Liefern Ergebnisse, ändern nicht den Systemzustand
- Separate Modelle für Lese- und Schreiboperationen
- Ermöglicht unterschiedliche Optimierungen für beide Operationstypen
- Kann mit Event Sourcing kombiniert werden

Aufbau von CQRS

Ein CQRS-System besteht aus folgenden Hauptkomponenten:

- **Command-Seite:**
  - Command-Objekte (repräsentieren Benutzerabsichten)
  - Command-Handler (verarbeiten Commands)
  - Write-Model (Domänenmodell, optimiert für Konsistenz)
  - Write-Datenbank (häufig relationale DB)
- **Query-Seite:**
  - Query-Objekte (repräsentieren Leseanfragen)
  - Query-Handler (verarbeiten Queries)
  - Read-Model (denormalisierte Views, optimiert für Leseoperationen)
  - Read-Datenbank (häufig NoSQL oder spezialisierte Leseoptimierung)
- Optional: Synchronisationsmechanismus (um Änderungen vom Write- zum Read-Model zu propagieren)

Vorteile von CQRS

- Ermöglicht unabhängige Skalierung von Lese- und Schreiboperationen
- Optimierung für spezifische Anwendungsfälle (z.B. komplexe Berichte)
- Bessere Leistung durch spezialisierte Datenmodelle
- Vereinfachte Domänenmodelle durch Trennung der Verantwortlichkeiten
- Verbesserte Sicherheit durch getrennte Berechtigungsmodelle möglich
- Unterstützung von komplexen Geschäftsregeln auf der Schreibseite

Nachteile von CQRS

- Erhöhte Komplexität durch doppelte Modelle und Synchronisation
- Eventual Consistency kann für Benutzer verwirrend sein
- Höherer Entwicklungsaufwand, besonders in der Anfangsphase
- Größere Lernkurve für Entwickler
- Nicht für alle Anwendungen sinnvoll (Overengineering für einfache CRUD-Systeme)

CQRS Implementierungsbeispiel

```java
// Command-Seite
public class CreateOrderCommand {
    private final String customerId;
    private final List<OrderItem> items;

    // Constructor, getters...
}

public class CreateOrderHandler {
    private final OrderRepository repository;

    public void handle(CreateOrderCommand command) {
        // Geschaeftslogik, Validierung
        Order order = new Order(command.getCustomerId(), command.getItems());
        repository.save(order);
        eventBus.publish(new OrderCreatedEvent(order.getId()));
    }
}

// Query-Seite
public class GetOrderQuery {
    private final String orderId;
    // Constructor, getters...
}

public class GetOrderHandler {
    private final OrderReadRepository readRepository;

    public OrderDto handle(GetOrderQuery query) {
        return readRepository.findById(query.getOrderId());
    }
}
```

Event Sourcing

Event Sourcing

Event Sourcing ist ein Architekturmuster, bei dem der Zustand eines Systems als Sequenz von Ereignissen modelliert wird:

- Der aktuelle Zustand wird durch Anwenden aller historischen Ereignisse rekonstruiert
- Ereignisse (Events) sind unveränderlich und werden nur hinzugefügt
- Jede Änderung am System wird als Event aufgezeichnet
- Bietet vollständige Nachvollziehbarkeit und Audit-Trail
- Häufig in Kombination mit CQRS eingesetzt

Kernkonzepte von Event Sourcing

- **Event:** Aufzeichnung einer Zustandsänderung (z.B. OrderPlaced, ItemAdded)
- **Event Store:** Persistente Speicherung aller Events
- **Aggregate:** Konsistenzgrenzen für Entities (z.B. Order, Customer)
- **Snapshots:** Momentaufnahmen des Zustands zur Performance-Optimierung
- **Event Handlers:** Verarbeiten Events und aktualisieren Read-Modelle
- **Projections:** Erzeugen verschiedene Ansichten der Daten aus Events

Vorteile von Event Sourcing

- Komplette Historie aller Änderungen (Audit-Trail)
- Zeitreise-Fähigkeit (Zustand zu jedem historischen Zeitpunkt rekonstruierbar)
- Ermöglicht Debugging durch Wiedergabe von Events
- Vereinfacht das Testen von Geschäftslogik
- Erlaubt nachträgliche Erstellung neuer Projektionen
- Verbesserte Skalierbarkeit durch Vermeidung von Update-Konflikten
- Robustheit gegen Datenverlust durch append-only Natur

Nachteile von Event Sourcing

- Erhöhte Komplexität, besonders bei der ersten Umsetzung
- Erhöhte Anforderungen an die Infrastruktur (Event Store)
- Eventual Consistency kann für Benutzer verwirrend sein
- Versioning von Events kann herausfordernd sein
- Performance-Überlegungen bei großen Event-Streams
- Erhöhter initialer Entwicklungsaufwand

Event Sourcing Implementierungsbeispiel

```java
// Event
public class OrderPlacedEvent {
    private final String orderId;
    private final String customerId;
    private final LocalDateTime timestamp;
    private final List<OrderItem> items;

    // Constructor, getters...
}

// Aggregate
public class Order {
    private String id;
    private String customerId;
    private List<OrderItem> items = new ArrayList<>();
    private OrderStatus status;

    // Apply-Methoden fuer Events
    public void apply(OrderPlacedEvent event) {
        this.id = event.getOrderId();
        this.customerId = event.getCustomerId();
        this.items.addAll(event.getItems());
        this.status = OrderStatus.PLACED;
    }

    // Command-Handling
    public OrderPlacedEvent placeOrder(String customerId, List<OrderItem> items) {
        // Validierung
        OrderPlacedEvent event = new OrderPlacedEvent(
            UUID.randomUUID().toString(),
            customerId,
            LocalDateTime.now(),
            items
        );
        apply(event);
        return event;
    }
}

// Event Store
public class EventStore {
    private final Map<String, List<Event>> eventStreams = new HashMap<>();

    public void saveEvents(String aggregateId, List<Event> events) {
        List<Event> stream = eventStreams.getOrDefault(aggregateId, new
             ArrayList<>());
        stream.addAll(events);
        eventStreams.put(aggregateId, stream);
    }

    public List<Event> getEventsForAggregate(String aggregateId) {
        return eventStreams.getOrDefault(aggregateId, new ArrayList<>());
    }
}
```

Strangler Pattern / Online-Migration

Strangler Pattern

Das Strangler Pattern (auch Strangler Fig Pattern) ist ein Ansatz zur schrittweisen Migration eines Legacy-Systems zu einer neuen Architektur:

- Benannt nach der "Würgefeige"(Strangler Fig), die um einen Wirtsbaum wächst und ihn schließlich ersetzt
- Neue Funktionalität wird parallel zum alten System entwickelt
- Bestehende Funktionen werden schrittweise migriert
- Die Umstellung erfolgt inkrementell, nicht als Big-Bang-Ansatz
- Das Legacy-System wird nach und nach ërwürgt"(ersetzt)

Implementierung des Strangler Patterns

Typische Implementierungsstrategie:

- **Facade:** Einrichten einer Fassade vor dem Legacy-System
- **Interception:** Abfangen von Anfragen und Weiterleitung an das neue oder alte System
- **Koexistenz:** Altes und neues System arbeiten parallel
- **Graduelle Migration:** Feature für Feature wird migriert
- **Auslaufen:** Das alte System wird schrittweise abgeschaltet

Vorteile des Strangler Patterns

- Reduziertes Risiko im Vergleich zu einem kompletten Neubau
- Frühes Wertschöpfungspotenzial durch inkrementelle Lieferung
- Möglichkeit, früh Feedback zu erhalten und zu reagieren
- Kontinuierliche Bereitstellung von Geschäftswert
- Lernen während der Migration
- Bessere Ressourcenplanung möglich

Herausforderungen des Strangler Patterns

- Komplexere Übergangszustände während der Migration
- Höherer Koordinationsaufwand zwischen Systemen
- Notwendigkeit für Kompatibilitätsschichten
- Daten-Synchronisationsprobleme
- Längere Gesamtmigrationsdauer
- Risiko unvollständiger Migration (ëwiges"Projekt)

Durchführung einer Strangler-Migration

Analyse und Planung

- Legacy-System analysieren und verstehen
- Abhängigkeiten und Integrationspunkte identifizieren
- Migrationsreihenfolge basierend auf Geschäftswert und technischer Machbarkeit festlegen
- Fassadenschicht planen (API Gateway, Reverse Proxy etc.)

Implementierung

- Fassadenschicht vor dem Legacy-System einrichten
- Routingregeln definieren (neu vs. alt)
- Mit isolierten, weniger kritischen Funktionen beginnen
- Schrittweise weitere Funktionen migrieren
- Dual-Write-Mechanismen für Datenmigration einsetzen

Konsolidierung

- Erfolgskriterien für jede migrierte Funktion definieren
- Traffic schrittweise zum neuen System umleiten
- Alt-System-Komponenten nach erfolgreicher Migration abschalten
- Ressourcen für das Legacy-System reduzieren
- Migration dokumentieren und Lessons Learned festhalten

Circuit Breaker / Bulkhead / Retry

Circuit Breaker Pattern

Das Circuit Breaker Pattern verhindert, dass eine Anwendung versucht, eine Operation auszuführen, die wahrscheinlich fehlschlagen wird:

- Basiert auf der Idee elektrischer Sicherungen
- Überwacht Fehler in Aufrufen externer Systeme
- Bei zu vielen Fehlern "öffnetßich der Circuit Breaker und blockiert weitere Aufrufe
- Nach einer Wartezeit wird ein "HalboffenerSZustand eingenommen, um probeweise Aufrufe zuzulassen
- Bei erfolgreichen Aufrufen schließt sich der Circuit Breaker wieder

Zustände eines Circuit Breakers

Ein Circuit Breaker hat typischerweise drei Zustände:

- **Geschlossen (Closed):** Aufrufe werden normal durchgeführt, Fehler werden gezählt
- **Offen (Open):** Aufrufe werden sofort abgewiesen, ohne den externen Service zu kontaktieren
- **Halb-offen (Half-Open):** Einige Testaufrufe werden durchgelassen, um zu prüfen, ob der Dienst wieder verfügbar ist

Bulkhead Pattern

Das Bulkhead Pattern isoliert Elemente eines Systems, sodass der Ausfall eines Elements nicht zum Ausfall des gesamten Systems führt:

- Benannt nach den wasserdichten Schotten in Schiffen
- Ressourcen werden in isolierte Pools aufgeteilt
- Jeder Dienst/Komponente erhält eigene Ressourcen (z.B. Threadpools, Verbindungspools)
- Verhindert Ressourcenerschöpfung des Gesamtsystems
- Begrenzt die Auswirkungen von Fehlern auf einzelne Komponenten

Retry Pattern

Das Retry Pattern ermöglicht einer Anwendung, vorübergehende Fehler zu behandeln, indem Operationen automatisch wiederholt werden:

- Geeignet für transiente Fehler (z.B. Netzwerkunterbrechungen)
- Verschiedene Retry-Strategien möglich:
  - Sofortiges Retry
  - Exponentielles Backoff (zunehmende Wartezeiten)
  - Jitter (zufällige Variationen der Wartezeiten)
- Begrenzte Anzahl von Wiederholungen, um endlose Schleifen zu vermeiden

Kombination der Patterns

Diese Resilience Patterns werden oft kombiniert:

- Circuit Breaker schützt vor anhaltenden Fehlern
- Bulkhead isoliert Auswirkungen von Fehlern
- Retry behandelt vorübergehende Fehler
- Zusammen bilden sie eine umfassende Resilienzstrategie
- Weitere ergänzende Patterns: Timeout, Fallback, Cache

Circuit Breaker Implementierung mit Resilience4j

```java
// Circuit Breaker Konfiguration
CircuitBreakerConfig config = CircuitBreakerConfig.custom()
    .failureRateThreshold(50)     // 50% Fehlerrate oeffnet den Circuit Breaker
    .waitDurationInOpenState(Duration.ofMillis(1000))  // 1s im Open-State
    .ringBufferSizeInHalfOpenState(10)  // Anzahl Aufrufe im Half-Open State
    .ringBufferSizeInClosedState(100)   // Anzahl Aufrufe im Closed State
    .build();

// Circuit Breaker erstellen
CircuitBreaker circuitBreaker = CircuitBreaker.of("backendService", config);

// Funktion mit Circuit Breaker absichern
Supplier<String> decoratedSupplier = CircuitBreaker
    .decorateSupplier(circuitBreaker, () -> backendService.doSomething());

// Aufruf der geschuetzten Funktion
try {
    String result = decoratedSupplier.get();
} catch (Exception e) {
    // Handle exception (Circuit open or service error)
}
```

Vergleich von Architekturstilen

Monolith

Eine Monolith-Architektur ist eine traditionelle Softwarearchitektur, bei der alle Komponenten in einer einzigen Anwendung integriert sind:

- **Vorteile:**
  - Einfachere Entwicklung und Deployment
  - Weniger Komplexität bei Transaktionen und Konsistenz
  - Einfacheres Debugging und Testen
  - Geringere Latenz bei internen Aufrufen
- **Nachteile:**
  - Schwierigere Skalierung einzelner Komponenten
  - Langsamere Entwicklungszyklen bei großen Anwendungen
  - Höhere Kopplung zwischen Komponenten
  - Technologieabhängigkeit (Plattform, Sprache)

Modulith

Ein Modulith ist ein gut strukturierter Monolith mit internen Modulen:

- **Vorteile:**
  - Behält Einfachheit des Monoliths bei
  - Verbesserte Struktur und Wartbarkeit
  - Klarere Domänengrenzen innerhalb der Anwendung
  - Kann später leichter in Microservices aufgeteilt werden
- **Nachteile:**
  - Weiterhin Einschränkungen bei unabhängiger Skalierbarkeit
  - Gemeinsame Deployment-Einheit kann weiterhin problematisch sein
  - Erfordert Disziplin zur Einhaltung der Modulgrenzen

Microservices

Microservices sind eine Architektur, bei der eine Anwendung als Sammlung kleiner, unabhängiger Dienste entwickelt wird:

- **Vorteile:**
  - Unabhängige Entwicklung und Deployment
  - Bessere Skalierbarkeit einzelner Komponenten
  - Technologische Flexibilität
  - Resilienz durch Isolation von Fehlern
  - Ermöglicht Conway's Law positiv zu nutzen
- **Nachteile:**
  - Erhöhte Komplexität der Infrastruktur
  - Herausforderungen bei verteilten Transaktionen
  - Potential für erhöhte Latenz
  - Anspruchsvolleres Monitoring und Debugging
  - Höhere Betriebskosten

Self-contained Systems (SCS)

Self-contained Systems sind unabhängige Systeme, die zusammen eine größere Anwendung bilden:

- **Vorteile:**
  - Autonomie einzelner Teams
  - Gröber als Microservices, daher weniger Overhead
  - Reduzierte Abhängigkeiten zwischen Teams
  - Gute Balance zwischen Monolith und Microservices
- **Nachteile:**
  - Mögliche Duplizierung von Funktionalität
  - Herausforderungen bei systemübergreifender Konsistenz
  - Komplexere Benutzeroberflächen-Integration

Serverless

Serverless ist ein Cloud-Computing-Modell, bei dem der Cloud-Anbieter die Infrastruktur dynamisch verwaltet:

- **Vorteile:**
  - Keine Server-Verwaltung notwendig
  - Automatische Skalierung
  - Pay-per-Use-Preismodell
  - Schnelle Markteinführung
  - Fokus auf Geschäftslogik statt Infrastruktur
- **Nachteile:**
  - Cold-Start-Problematik (Latenz)
  - Vendor Lock-in
  - Beschränkungen bei Laufzeit und Ressourcen
  - Schwierigere Fehlersuche und Testing
  - Komplexeres Monitoring

Wahl der richtigen Architektur

Folgende Faktoren sollten bei der Wahl der Architektur berücksichtigt werden:

- **Organisationsstruktur:** Conway's Law beachten
- **Teamgröße:** Kleine Teams - Monolith/Modulith, große Teams - Microservices/SCS
- **Veränderungsrate:** Hohe Änderungsrate begünstigt Microservices
- **Domänenkomplexität:** Komplexe Domänen erfordern klare Grenzen
- **Skalierungsbedarf:** Unterschiedliche Skalierungsanforderungen begünstigen Microservices
- **Konsistenzanforderungen:** Hohe Konsistenz begünstigt Monolith
- **Technologie-Diversität:** Bedarf an verschiedenen Technologien begünstigt Microservices
- **Betriebsreife:** DevOps-Fähigkeiten sind Voraussetzung für Microservices