Prozessqualität

Bezug zu SWT C

Aktivitäten der Softwareentwicklung

- Die vielfältigen Arbeiten, die bei der Entwicklung von Software anfallen, lassen sich unabhängig von der fachlichen Aufgabenstellung in Tätigkeitsbereiche einordnen:

- Die Tätigkeiten lassen sich noch weiter unterteilen

- Analyse [Analyse, Spezifikation der Anforderungen]
  - Verstehen der (fachlichen) Aufgabenstellung
  - Welche Aufgaben sollen von der Software übernommen werden?
  - Prüfen, Korrigieren und Dokumentieren der Anforderungen

- Entwurf
  - Berücksichtigung technischer Randbedingungen
  - Aufteilen der Gesamtfunktionalität auf Module/Komponenten
  - Festlegen von Schnittstellen
  - Strukturierung der Module/Komponenten
- Codierung/Implementierung [Codierung, Modultest]
  - Die Module werden gemäß der Spezifikation mit der gewählten Programmiersprache implementiert
  - Die Module werden auf Basis der Spezifikation getestet
- Integration [Integration, Test, Abnahme]
  - Das Gesamtsystem wird aus den einzelnen Komponenten/Modulen zusammengebaut
  - Test des Gesamtsystems
  - Abnahme durch den Kunden

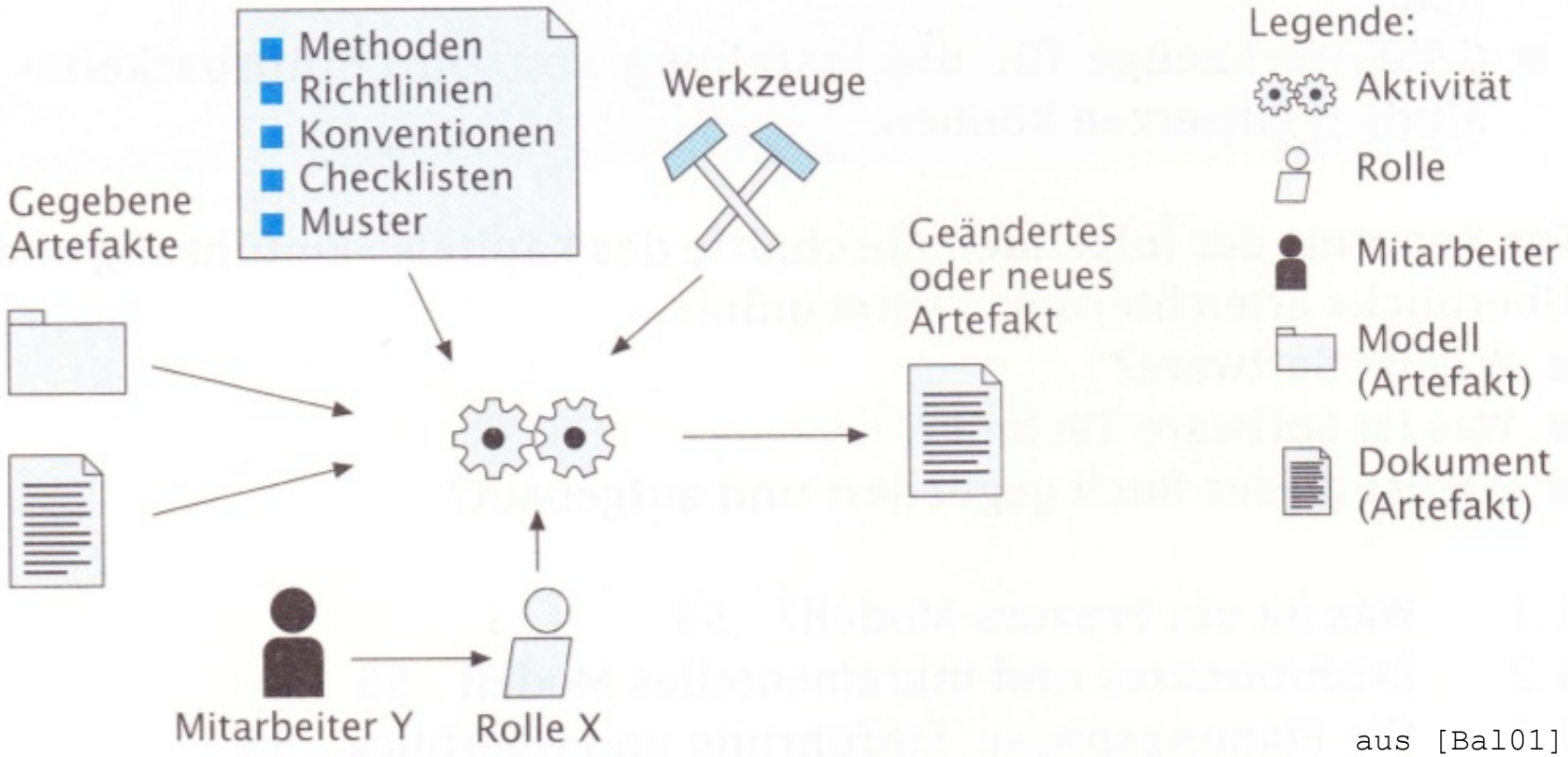
- Betrieb [Installation, Betrieb, Monitoring, Wartung, Pflege, Ersetzung]
  - Die Software wird in der Zielumgebung installiert und in Betrieb genommen
  - *Wartung*: Fehler, die sich während des Betriebs zeigen, werden beseitigt
  - *Pflege*: Es werden Änderungen/Erweiterungen der Funktionalität vorgenommen (z.B. auf Grund von gesetzlichen Anforderungen)
  - Bei Bedarf wird das System durch neue Software ersetzt. Der Übergang erfordert wieder eine präzise Planung und Umsetzung

- Eine Reihe von Tätigkeiten finden während der gesamten Entwicklungszeit statt:
  - Planung/Management
  - Dokumentation
  - Qualitätssicherung

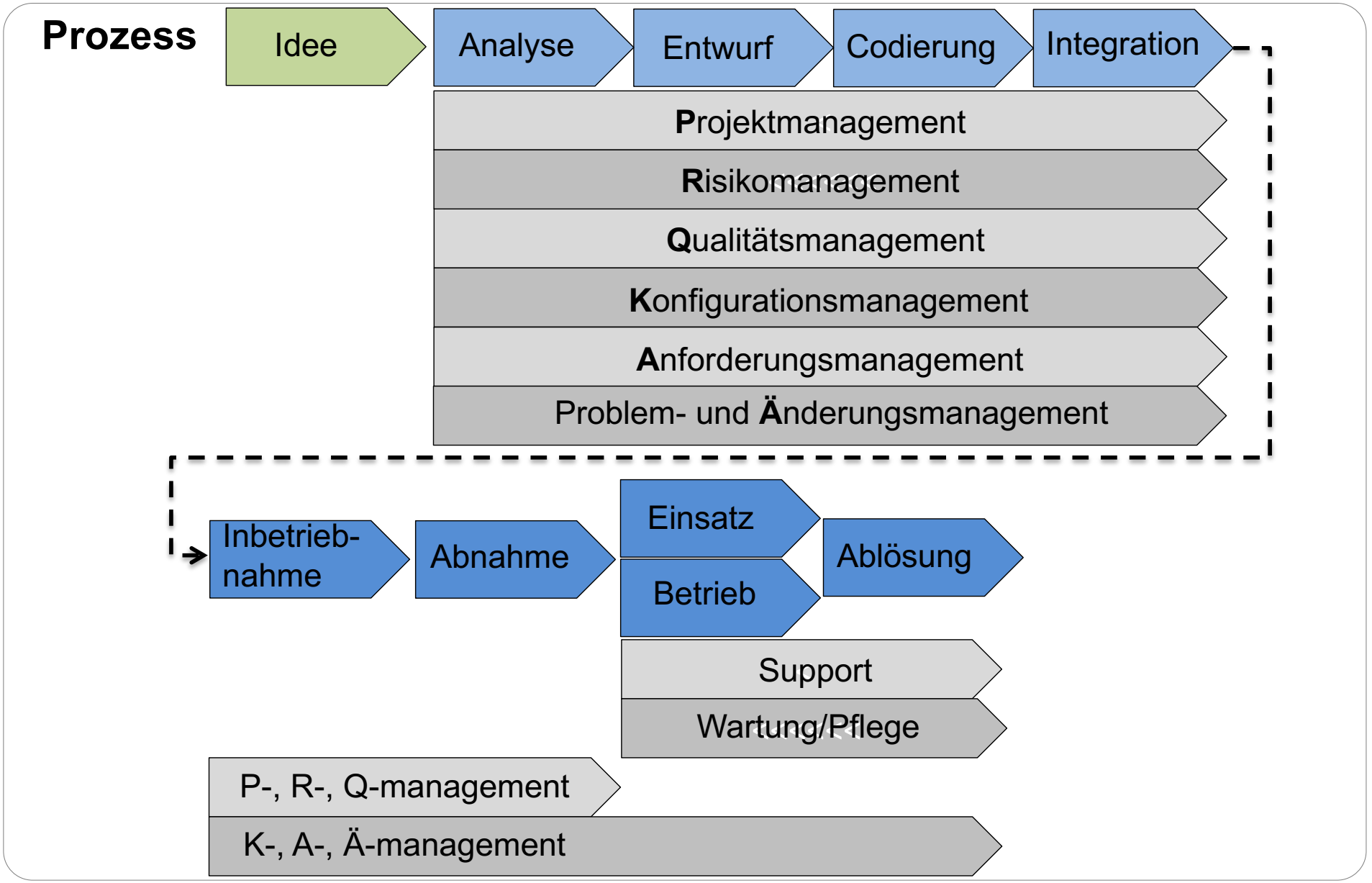
- Es ist festzulegen, in welcher Reihenfolge und wie häufig die einzelnen Tätigkeiten durchlaufen werden
- Hierzu existieren verschiedene **Vorgehensmodelle**

- Jede **Tätigkeit/Aktivität** wird von einem **Mitarbeiter** ausgeführt. Der Mitarbeiter nimmt dabei eine **Rolle** ein
- Jede Tätigkeit liefert **Ergebnisse**. Dabei kann es sich z.B. um Texte, Diagramme oder Quellcodes handeln. Die Ergebnisse werden allgemein auch als **Artefakte** bezeichnet

- **Prozessmodelle** legen neben der Vorgabe einer Reihenfolge der Tätigkeiten zusätzlich die folgenden Punkte fest:
  - personelle Organisation (Rollen)
  - Aufbau der Artefakte (z.B. Gliederung der Dokumentation)
  - Verantwortlichkeiten für Aktivitäten und Artefakte


Methoden
Richtlinien
Konventionen
Checklisten
Muster
Werkzeuge
Gegebene Artefakte
Geändertes oder neues Artefakt
Mitarbeiter Y
Rolle X
Legende:
Aktivität
Rolle
Mitarbeiter
Modell (Artefakt)
Dokument (Artefakt)

aus [Bal01]


Prozess
Idee
Analyse
Entwurf
Codierung
Integration
Projektmanagement
Risikomanagement
Qualitätsmanagement
Konfigurationsmanagement
Anforderungsmanagement
Problem- und Änderungsmanagement
Inbetrieb-nahme
Abnahme
Einsatz
Betrieb
Ablösung
Support
Wartung/Pflege
P-, R-, Q-management
K-, A-, Ä-management

Prozessqualität

- Ein guter Prozess führt nicht zwangsläufig zu guter Software, ein guter Prozess begünstigt aber die Erstellung guter Software
- Es gibt verschiedene Ansätze, um den Reifegrad eines Prozesses zu bestimmen
  - Ein hoher Reifegrad begünstigt eine hohe Prozessqualität

- Die folgenden Themen werden in der Vorlesung Softwaretechnik C behandelt
  - Softwareprojektmanagement
  - Vorgehens- und Prozessmodelle
  - Risikomanagement
  - Konfigurationsmanagement
  - Anforderungs- und Änderungsmanagement

Die Themen werden in SWT D pragmatisch aus Sicht der Software-qualitätssicherung aufgenommen

Konstruktive Maßnahmen

Konstruktive Maßnahmen

- Durch geeignete Maßnahmen soll die Entstehung von Fehlern von Anfang an vermieden werden
- Produkt- und Prozessqualität können durch konstruktive Maßnahmen beeinflusst werden
- Wir werden verschiedene konstruktive Maßnahmen betrachten

1. Qualifikation und Schulungen
   - Als Beispiel betrachten wird die Programmierung
   - Das eigentliche Programmieren (Erstellen des Quellcodes) wird häufig als reines Handwerk gesehen
   - Diese Sichtweise beruht auf dem Wunsch, dass das Ergebnis der Codierung ein normgerechtes Ingenieurprodukt sein soll, und kein künstlerisches Werk (siehe [LL10])
   - Dieser Manager-Traum ist aber nicht in Erfüllung gegangen

- Gute Programme werden (immer noch) nicht mit wenigen automatisierten Handgriffen am Fließband erstellt
- Die Codierung / Implementierung umfasst eine Reihe von Aktivitäten:
  - Konzeption von Datenstrukturen und Algorithmen
  - Strukturierung des Programms
  - Erstellung des Quellcodes in der gewählten Programmiersprache
  - Dokumentation und Kommentierung der Programme
  - Testplanung
  - Test des Programms

Gute Programme werden von hochqualifizierten und motivierten Mitarbeitern erstellt

2. Attraktive Arbeitsumgebung
   - Qualifizierte Mitarbeiter fordern in der Regel interessante Aufgaben und benötigen eine gewisse gestalterische Freiheit
     - „Aufstiegsmöglichkeiten“ sind im technischen Bereich noch häufig ein ungelöstes Problem
   - Die Arbeitszeiten müssen angemessen (keine systematischen Überstunden) und flexibel sein
   - Die Ausstattung des Arbeitsplatzes muss angemessen sein

3. Verwendung eines geeigneten Prozessmodells
   - Falls das Prozessmodell nicht passend gewählt wird, können sich negative Auswirkungen ergeben
     - Reduzierung der Produktivität
     - Schlechte Einbindung der erforderlichen Maßnahmen zur Qualitätssicherung
   - Die folgenden Punkte sind bei der Auswahl eines Prozessmodells zu berücksichtigen
     - Vollständigkeit der Anforderungen
     - Stabilität der Anforderungen
     - Technologisches Know-How
     - Risikobewertung
     - Teamgröße
     - Persönlichkeitsprofile der Teammitglieder / Gruppendynamik
     - Erfahrung des Teams mit dem Prozessmodell

4. Verwendung von Werkzeugen

- Die professionelle Softwareentwicklung erfordert den Einsatz von geeigneten Werkzeugen
- Dabei sind die Werkzeuge auch selbst Software
- Die Verwendung von **Softwareentwicklungswerkzeugen** wird auch unter dem Begriff **CASE** zusammengefasst:

> ***computer-aided software engineering (CASE) –*** *The use of computers to aid in the software engineering process. May include the application of software tools to software design, requirements tracing, code production, testing, document generation, and other software engineering activities*

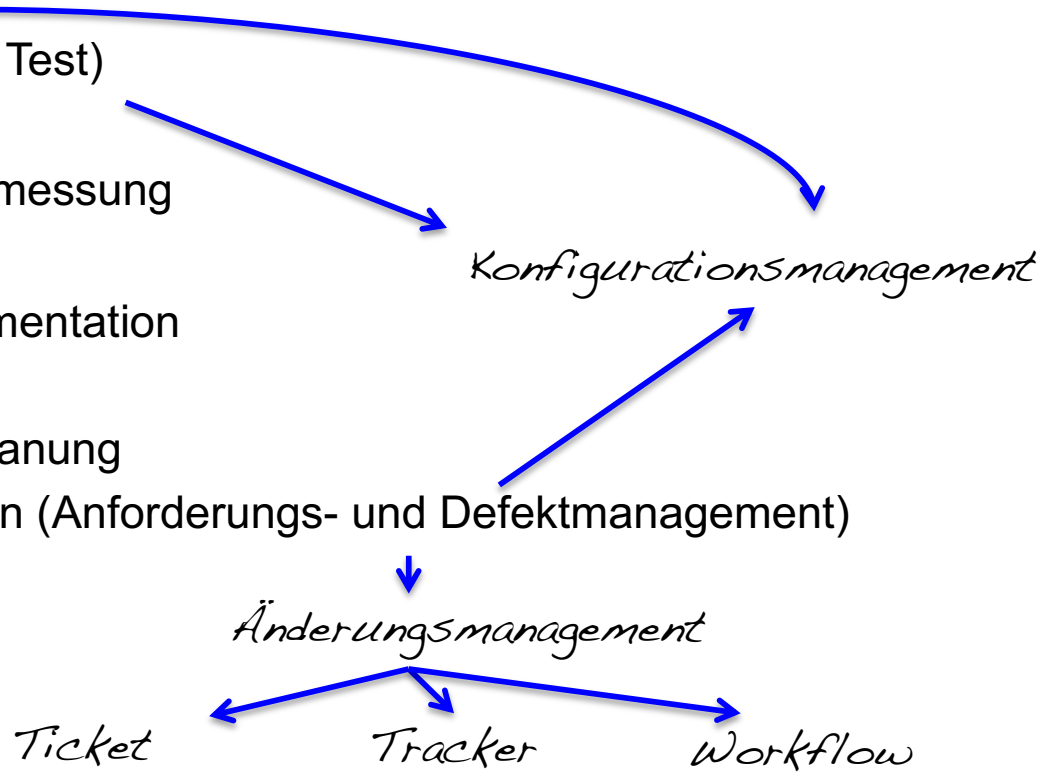
IEEE Std 610.12 (1990)

- Mit der Verwendung von Werkzeugen kann die Effizienz und die Effektivität der Softwareentwicklung gesteigert werden
- Damit ein Werkzeug mit Gewinn eingesetzt werden kann, müssen die folgenden Bedingungen erfüllt sein:
  - Das Werkzeug muss zur Aufgabe passen
  - Das Werkzeug muss sinnvoll eingesetzt werden
- Werkzeuge sind Teil einer Methode
- Werkzeuge unterstützen unterschiedliche Operationen

- Die Operationen lassen sich wie folgt klassifizieren:
  1. Editieren
  2. Transformieren
  3. Verwalten / Versionieren
  4. Suchen
  5. Nachvollziehen
  6. Messen
  7. Testen
  8. Verbinden
  9. Dokumentieren
  10. Verfolgen / Überwachen

- Werkzeug-Grundausstattung in einem professionellen Projekt
  - Methodenspezifischer Editor (Syntaxhighlighting, Autovervollständigung)
  - Compiler/Linker (inkl. Standardbibliotheken)
  - Versionsverwaltung
  - Automatisierung (Build, Test)
  - Überdeckungsprüfung
  - Statische Prüfung / Vermessung
  - Code Review
  - Kommunikation / Dokumentation
  - Aufwandserfassung
  - Projekt- / Ressourcenplanung
  - Verfolgen- / Überwachen (Anforderungs- und Defektmanagement)
  - ...

Integration über IDE



5. Verwendung einer geeigneten Programmiersprache
   - Die verwendete Programmiersprache hat Einfluss auf die Effizienz der Programmerstellung und die Qualität des Programms
   - Die Programmiersprache sollte zur Problemdomäne passen
     - Bessere Abstraktion
     - Klarere Struktur
     - Weniger Quellcode
     - Höhere Produktivität
   - Die Sprache sollte Eigenschaften besitzen, die sich positiv auf die Qualität der Programmierer und die Qualität der Programme auswirken (siehe[LL10]):
     - Strukturelemente zur Konstruktion modularer Programmeinheiten
     - Typsystem mit strenger Typprüfung
     - Trennung von Schnittstelle und Implementierung
     - Syntax, die zur Lesbarkeit des Codes beiträgt
     - Automatische Zeigerverwaltung
     - Ausnahmebehandlung
     - Gute Unterstützung durch Werkzeuge

6. Einhalten von Richtlinien

- Software wird in der Regel in Teams erstellt
- Dabei treten u.a. die folgenden Probleme auf
  - Zu viel Individualität erschwert die gemeinsame Arbeit an Artefakten (unterschiedliche Strukturen und Notationen)
  - Unter Zeitdruck wird vergessen wichtige Informationen zu dokumentieren
  - Hilfreiche Darstellungsformen sind nicht allen Teammitgliedern bekannt
- Durch die Vorgabe von Richtlinien will man die folgenden Punkte erreichen
  - Vereinheitlichung, um die Effizienz der Teamarbeit zu steigern
  - Fehlerreduktion, in dem problematische Darstellungs- und Notationsformen vermieden werden

Konkrete Richtlinien werden noch im weiteren Verlauf der Vorlesung betrachtet

- Richtlinien können für unterschiedliche Artefakte existieren
  - Gliederungsschema für ein Pflichtenheft
  - Verzeichnisstrukturen für Projektdateien
  - Aufbau eines Glossareintrags
  - ...
- Richtlinien können differenziert werden

a) Notationskonventionen
   - Schreibweise von Bezeichnern
   - Layout des Quelltextes
   - Aufbau von Kontrollstrukturen
   - ...

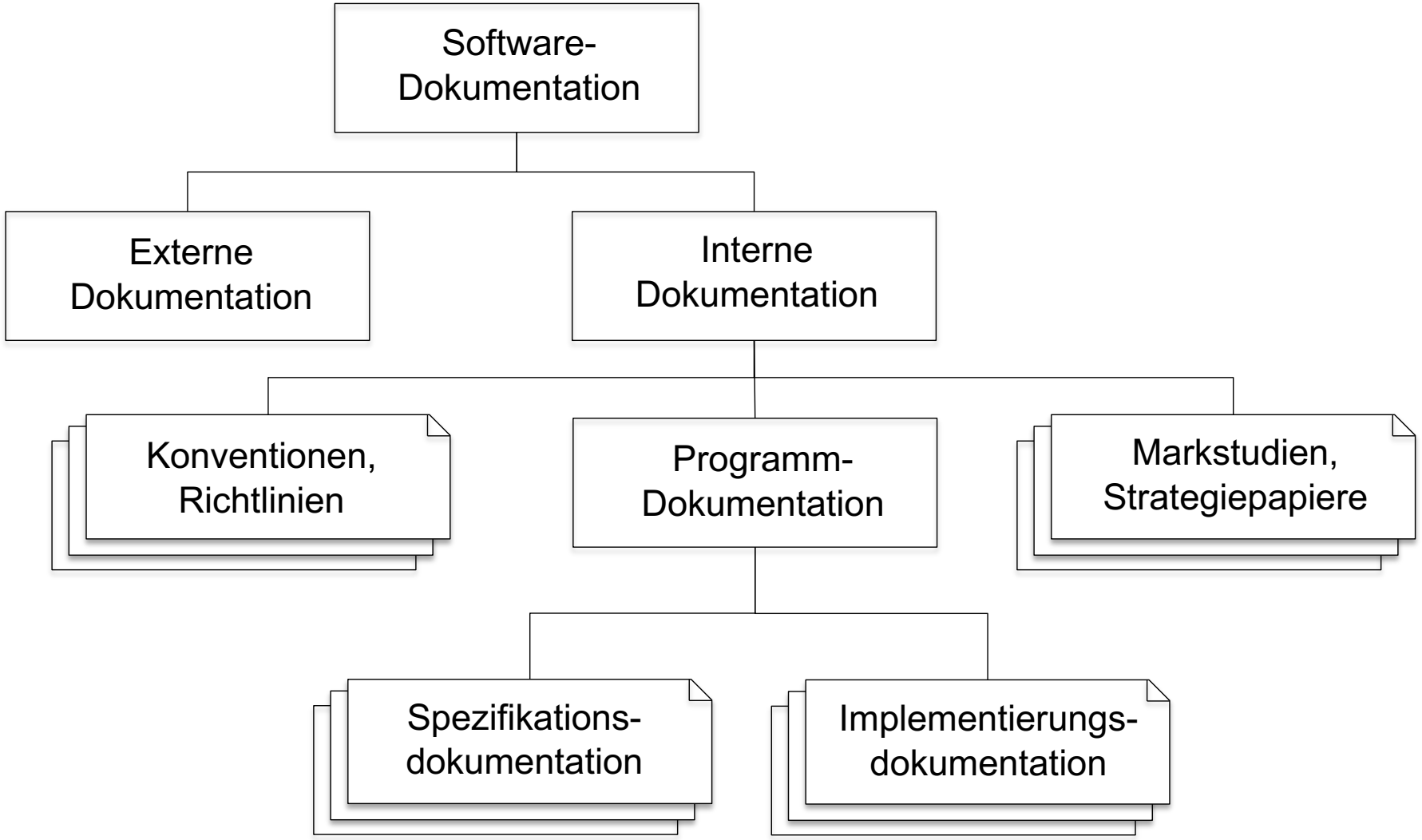
b) Sprachkonventionen
   - Regeln für die Formulierung von Anforderungen im Rahmen der Spezifikation
   - Umgang mit den semantischen Besonderheiten einer Programmiersprache
   - ...

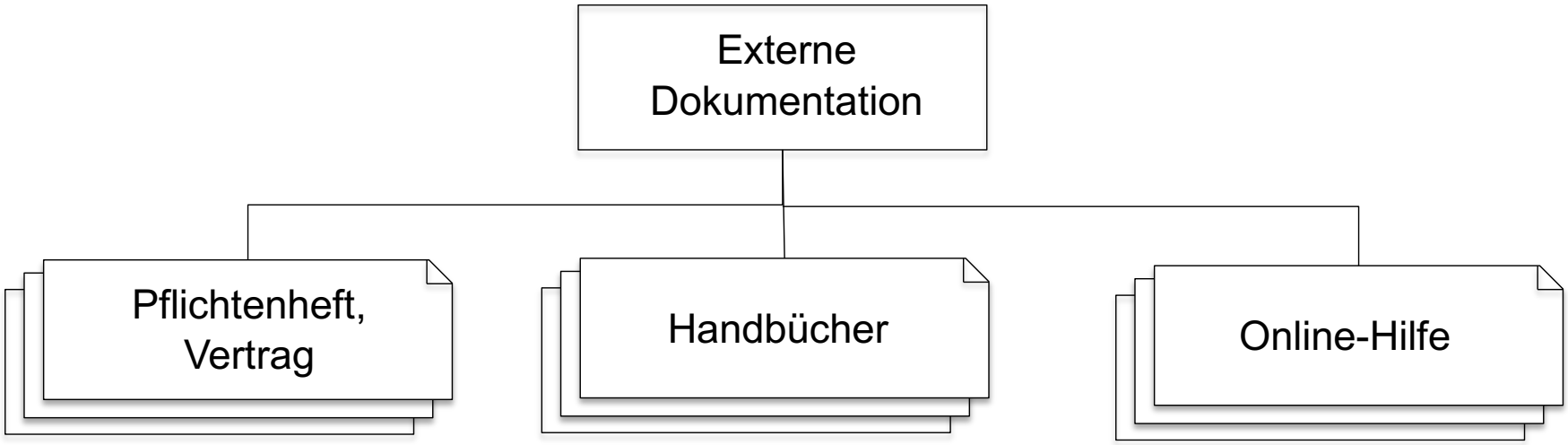
7. Erstellung einer angemessenen Dokumentation

- Wir erinnern uns: Software ist mehr als nur das ausführbare Programm
- Im Rahmen der Softwareentwicklung werden neben dem Quellcode noch zahlreiche weitere Artefakte erstellt
- Die Güte der erstellten Dokumente haben Einfluss auf unterschiedliche Qualitätsmerkmale
- Problem:
  - Manager unterschätzen der Wert einer angemessenen Dokumentation
  - Softwareentwickler (insbesondere Programmierer) fühlen sich durch die Dokumentation in ihrer „eigentlichen“ Arbeit gestört
- Lösung:
  - Festlegung auf eine sinnvolle Dokumentation
  - Unterstützung durch Richtlinien und Vorgaben
  - Sensibilisierung (Schulung) der Mitarbeiter (inkl. Management)

auch XP-Projekte kommen nicht ohne ein Handbuch und einer Online-Hilfe aus

- Dichotomie der Software-Dokumentation (nach [Hof13]):





- Externe Dokumentation
  - Mit der Erstellung kann/soll frühzeitig begonnen werden (Aufdeckung von widersprüchlichen und unvollständigen Anforderungen)
  - Hohe Anforderung an formale Kriterien (Rechtschreibung, Grammatik, Formatierung)
  - Hohe Anforderung an die Verständlichkeit

- Spezifikationsdokumentation

*Die **Spezifikation** dokumentiert die wesentlichen Anforderungen an eine Software und ihre Schnittstellen, und zwar präzise, vollständig und überprüfbar*

nach [LL10]

Darstellungsformen
- Informelle Spezifikation
- Semi-formale Spezifikation
- Formale Spezifikation

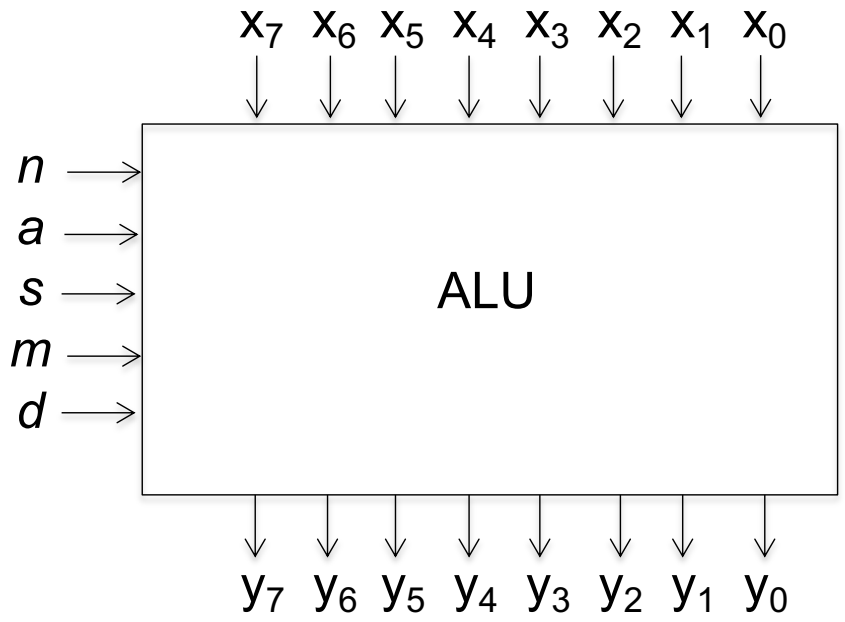
Eigenschaften
- zutreffend
- vollständig
- konsistent
- neutral
- nachvollziehbar
- überprüfbar (quantifizierbar)

Basis für:
- Entwurf und Implementierung
- Benutzerhandbuch
- Testvorbereitung
- Abnahme

Spezifikation beeinflusst die Qualität der nachfolgenden Aktivitäten

- Beispiel: Spezifikation einer ALU (serielle arithmetisch-logische Einheit)

nach [Hof13]



Ist die informelle Spezifikation konsistent und vollständig?

Informelle Spezifikation: Die ALU berechnet aus dem seriellen Eingabestrom (x_7,...,x_0) den Ausgabestrom (y_7,...,y_0). Die von der ALU ausgeführte Operation wird durch die Steuerleitungen *n*, *a*, *s*, *m*, *d* bestimmt. Für *n*=1 negiert die ALU den Eingabewert. Für *a*=1 berechnet sie die Summe, für *s*=1 die Differenz, für *m*=1 das Produkt und für *d*=1 den Quotienten der letzten beiden Eingabewerte

- Beispiel: Spezifikation einer ALU

Semi-formale Spezifikation

Eingabe:

`x[t]` : Eingabe zum Zeitpunkt *t* in 8-Bit-Zweierkomplementdarstellung

`n, a, s, m, d` : Steuerleitungen

Ausgabe:

`y[t]` : Ausgabe zum Zeitpunkt *t* in 8-Bit-Zweierkomplementdarstellung

Verhalten:

$$y[0] = \begin{cases} -x[0] & \text{für } n=1 \text{ und } a=s=m=d=0 \\ 0 & \text{sonst} \end{cases}$$

$$y[n+1] = \begin{cases} -x[n+1] & \text{für } n=1 \text{ und } a=s=m=d=0 \\ x[n] + x[n+1] & \text{für } a = 1 \text{ und } n=s=m=d=0 \\ x[n] - x[n+1] & \text{für } s = 1 \text{ und } n=a=m=d=0 \\ x[n] * x[n+1] & \text{für } m=1 \text{ und } n=a=s=d=0 \\ x[n] / x[n+1] & \text{für } d=1 \text{ und } n=a=s=m=0 \\ 0 & \text{sonst} \end{cases}$$

- Implementierungsdokumentation
  - Beschreibt, wie die Spezifikation umgesetzt wird
  - Man unterscheidet Code-Dokumentation und externe Dokumente
  - Externe Dokumente liefern in der Regel eine Schnittstellenbeschreibung (API) und geben einen Bezug zur Gesamtarchitektur
  - Der Wert einer guten Implementierungsdokumentation (gerade für die Wartung) wird häufig unterschätzt
  - Dritte können sich schneller einarbeiten, und die Wahrscheinlichkeit für fehlerhafte Änderungen wird minimiert
  - Auf Grund der hohen Mitarbeiterfluktuation im IT-Bereich ist die Implementierungsdokumentation von großer Wichtigkeit

- *Beispiel*: Sie sind neu in der Abteilung. Es tritt eine Fehlberechnung im produktiven Betrieb auf. Sie müssen schnellstmöglich den Fehler in der folgenden Funktion finden. Der Autor der Funktion hat die Firma verlassen

```
int ack(int n, int m)
{
    while(n != 0) {
        if (m == 0) {
            m = 1;
        } else {
            m = ack(m, n-1);
        }
        n--;
    }
    return m+1;
}
```

Kommentare würden die Fehlerbehebung deutlich beschleunigen

```
/*
            Funktion:  int ack (int n, int m)
               Autor:  Tom Hacker
                Date:  15/08/2013
   Revision History
         12/05/2010:  While-Iterator ergänzt
         20/12/2012:  Funktionsname geändert
*/
int ack(int n, int m)
{
    /* Solange die erste Variable nicht null ist ... */
    while(n != 0) {
        /* Teste zweite Variable auf null */
        if (m == 0) {
            m = 1;
        } else {
            /* Hier erfolgt der rekursive Aufruf */
            m = ack(m, n-1);
        }
        n--;
    }
    /* Liefert Ergebniswert zurück */
    return m+1;
}
```

Können Sie den Fehler nun schneller finden?

Streichen Sie alle Kommentare, die nicht hilfreich sind!

```
/* Autor       : Tom Hacker              Erstellungsdatum: 15/08/2013
   Beschreibung: Berechnet die Ackermann-Funktion
                 Die Ackermann Funktion ist berechenbar, aber
                 nicht primitiv berechenbar. Die Funktion ist
                 wie folgt definiert:
                 (1) ack(0,m) = m+1                    [m >= 0]
                 (2) ack(n,0) = ack(n-1,1)             [n > 0]
                 (3) ack(n,m) = ack(n-1, ack(n, m-1)) [m,n > 0]
*/
int ack(int n, int m)
{
    while(n != 0) {
        if (m == 0) {
         /* Fall(2) */
            m = 1;
        } else {
         /* Fall(3) */
            m = ack(m, n-1);
        }
        n--;
    }
    /* Fall(1) */
    return m+1;
}
```

Optimal!

Hier wird dokumentiert, was gemacht wird, und nicht, wie es gemacht wird

- Dokumentationsextraktion
  - Implementierungsdokumentation und externe Dokumentation laufen mit der Zeit auseinander
  - Informationen sollten daher nur an einer Stelle dokumentiert werden (in der Regel im Quellcode)
  - Es gibt Werkzeuge, die den Quelltext nach gekennzeichneten Kommentaren durchsuchen und daraus automatisch eine externe Dokumentation generieren (z.B. im HTML, oder PDF-Format)

JavaDoc

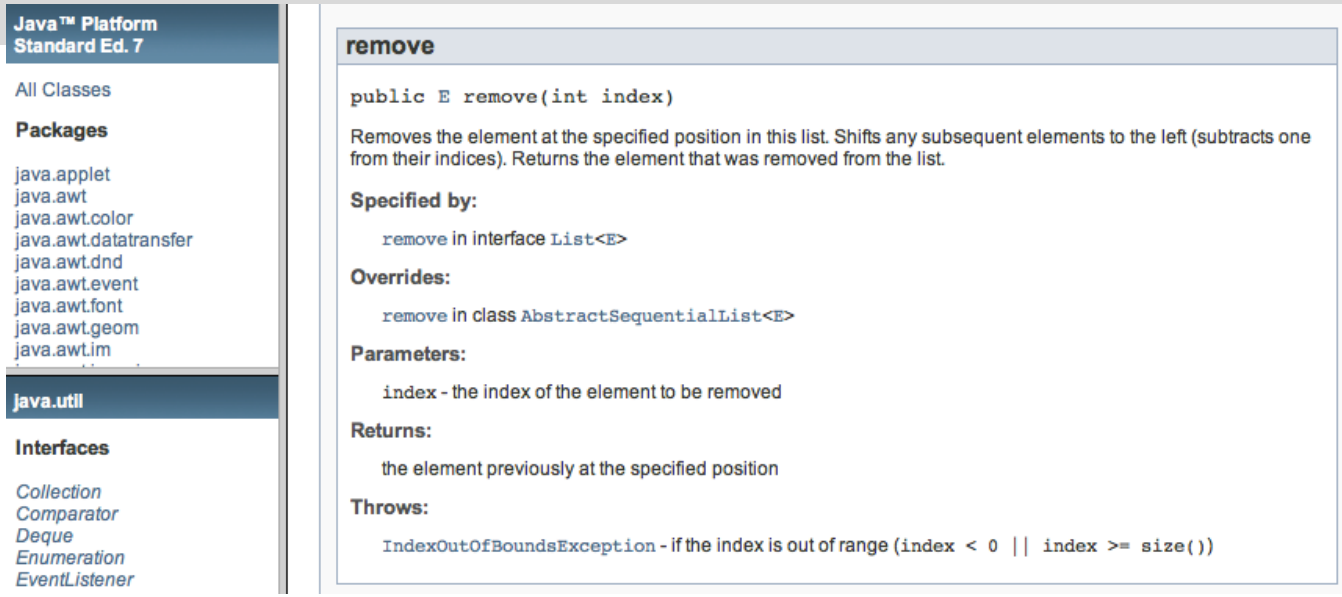
JavaDoc-Kommentare:
`/**  */`

Klassifikation der Kommentare:
Über Tags (mit @ als Prefix)

Tag	Bedeutung
@author	Name des Autors
@deprecated	Aktualitätsstatus
@exception	Ausgelöste Ausnahme
@param	Name und Funktion eines Parameters
@version	Version
@return	Rückgabewert
@since	Erstellungsdatum

```
/**
 * Removes the element at the specified position in this list.  Shifts any
 * subsequent elements to the left (subtracts one from their indices).
 * Returns the element that was removed from the list.
 *
 * @param index the index of the element to be removed
 * @return the element previously at the specified position
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public E remove(int index) {
   checkElementIndex(index);
   return unlink(node(index));
}
```

JavaDoc



8. Entwurf einer angemessenen Architektur
   - Komplexe Systeme sind schwer zu beherrschen und provozieren Fehler
   - Die Struktur eines Systems sollte übersichtlich und klar sein
   - Beim Entwurf sind insbesondere die folgenden Prinzipien zu befolgen
     - Modularisierung
     - Hierarchisierung
     - Starker Zusammenhalt und schwache Kopplung
     - Trennung von Zuständigkeiten
     - Information Hiding
   - Die Architektur hat auch einen direkten Einfluss auf die Qualitätsmerkmale Portabilität und Wartbarkeit

9. Automatisierung

- Immer wieder durchzuführende (mechanische) Routinetätigkeiten sind potentielle Fehlerquellen
  - Die Mitarbeiter werden nachlässig (Flüchtigkeitsfehler)
  - Die Zeit für die Routinetätigkeit kann nicht mehr in andere Maßnahmen zur Qualitätssicherung investiert werden
  - Neue Mitarbeiter müssen sich erst einarbeiten und machen zu Beginn viele Fehler
  - Wiederholbarkeit nicht gegeben
- Daher sind die folgenden Aktivitäten zu automatisieren
  1. Build-Prozess (inkl. Integration)
  2. Programmtest (Regressionstests)
  3. Versionierung
  4. Datensicherung der erstellten Artefakte
  5. Installation der Umgebungen (Entwicklung, Integration, Produktion)