Software Engineering: Wartung und Qualitätssicherung

17. Oktober 2019

1 Software-Entwicklung, Wartung und (Re)Engineering

Erstellte Software muss oft geändert werden, entweder aufgrund von geänderten Anforderungen oder neuen Anforderungen, welche eingebaut werden müssen.

1.1 Einleitung

1.1.1 Geschichte

- Softwarekrise 1968
- Nato **Working Conference on Software Engineering**
- Zuordnungen
  - Praktische Informatik
  - Theoretische Informatik
  - Projektplanung
  - Organisation
  - Psychologie
  - ...

1.1.2 "Software-Technik" Definition

Software-Engineering(Software-Technik) ist nach **Entwicklung**, **Pflege** und **Einsatz**.
Eingesetzt werden:

- Wissenschaftliche Methoden
- Wirtschaftliche Prinzipien
- Geplante Vorgehensmodellen
- Werkzeuge
- Quantifizierbare Ziele

1.1.3 50 Jahre nach Beginn der SSoftware-Krise"

- 19% aller Projekte sind gescheitert, früher 25%
- 52% aller Projekte sind dabei zu scheitern, früher 50%
- 29% aller betrachteten IT-Projekte sind erfolgreich, früher 25%

Hauptgründe fürs Scheitern der Projekte:
Unklare Anforderungen und Abhängigkeiten sowie Problemen beim Änderungsmanagement.

1.2 Software-Qualität

Ziel der Software-Technik ist die effiziente Entwicklung messbar qualitativ hochwertiger Software.

1.2.1 Qualitätsdefinition

Qualität ist der Grad, in dem ein System, eine Komponente oder ein Prozess die Kundenerwartungen und Kundenbedürfnisse erfüllt.

1.2.2 Softwarequalität

Softwarequalität ist die Gesamtheit der Funktionalitäten und Merkmale eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

1.2.3 Qualitätsmerkmale

- Funktionalität

1.2.4 Nichtfunktionale Merkmale

- Zuverlässigkeit (Reliability)
- Benutzbarkeit (Usability)
- Effizienz (Efficiency)
- Änderbarkeit (Maintainability)
- Übertragbarkeit (Portability)

1.2.5 Prinzipien der Qualitätssicherung

- **Qualitätszielbestimmung**: Auftraggeber und Auftragnehmer legen vor Beginn der Software-Entwicklung gemeinsames Qualitätsziel für Software-System mit nachprüfbaren Kriterienkatalog fest (als Bestandteil des abgeschlossenen Vertrags zur Software-Entwicklung)
- **Quantitative Qualitätssicherung**: Einsatz automatisch ermittelbaren Metriken zur Qualitätsbestimmung (objektivbare, ingenieursmäßige Vorgehensweise)
- **Konstruktive Qualitätssicherung**: Verwendung geeigneter Methoden, Sprachen und Werkzeuge (Vermeidung von Qualitätsproblemen)
- **Integrierte, frühzeitige, analytische Qualitätssicherung**: Systematische Prüfung aller erzeugter Dokumente (Aufdeckung von Qualitätsproblemen)
- **Unabhängige Qualitätssicherung**: Entwicklungsprodukte werden durch eigenständige Qualitätssicherungsabteilung überprüft und abgenommen (verhindert u.a. Verzicht auf Testen zugunsten Einhaltung des Entwicklungsplans)

1.2.6 Konstruktives Qualitätssicherung zur Fehlervermeidung

:

- Technische Maßnahmen
  - Sprachen (UML, Java)
  - Werkzeuge (UML-CASE-TOOL)
- Organisatorische Maßnahmen
  - Richtlinien (Gliederungsschema für Pflichtenheft, Programmierrichtlinien)
  - Standards (für verwendete Sprachen, Dokumentformate, Management)
  - Checklisten

1.2.7 Analytisches Qualitätsmanagement für Fehleridentifikation

- **Analysierende Verfahren**: Der "Prüfling"(Programm, Modell, Dokumentation) wird von Menschen oder Werkzeugen auf Vorhandensein/Abwesenheit von Eigenschaften untersucht
  - **Review**: Prüfung durch Menschen
  - **Statische Analyse**: Werkzeuggestützte Ermittlung von Änomalien"
  - **Formale Verifikation**: Werkzeuggestützter Beweis von Eigenschaften
- **Testende Vefahren**: Der "Prüfling"wird mit konkreten oder abstrakten Eingabewerten auf einem Rechner ausgeführt

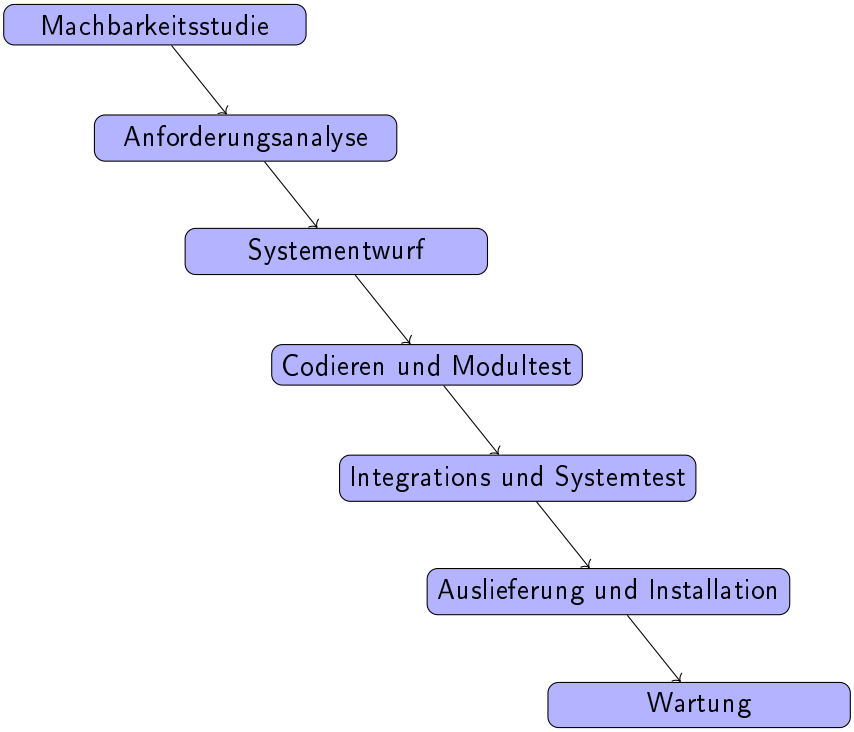
  - **Dynamischer Test**: "normaleÄusführung mit ganz konkreten Eingaben
  - **Symbolischer Test**: Ausführung mit symbolischen Eingaben

1.3 Iterative Softwareentwicklung

Voraussetzung für den sinnvollen Einsatz von Notationen und Werkzeugen zur Software-Entwicklung ist ein:

- **Vorgehensmodell**, das den Gesamtprozess der Software-Erstellung und pflege in einzelne Schritte aufteilt
- Zusätzlich müssen Verantwortlichkeiten der beteiligten Personen in Form von **Rollen** im Software-Entwicklungsprozess klar geregelt sein.

1.3.1 Übersicht der Phasen des Wasserfallmodells



1.3.2 Machbarkeitsstudie (feasability study)

Die Machbarkeitsstudie schätzt Kosten und Ertrag der geplanten Software-Entwicklung ab. Dazu grobe Analyse des Problems mit Lösungsvorschlägen.

- **Aufgaben**

  - Problem informell und abstrahiert beschreiben
  - Verschiedene Lösungsansätze erarbeiten
  - Kostenschätzung durchführen
  - Angebotserstellung

- **Ergebnisse**
  - Lastenheft
  - Projektkalkulation
  - Projektplan
  - Angebot an Auftraggeber

1.3.3 Anforderungsanalyse (requirements engineering)

In der Anforderungsanalyse wird exakt festgelegt, was die Software leisten soll, aber nicht wie diese Leistungsmerkmale erreicht werden.

- **Aufgaben**
  - genaue Festlegung der Systemeigenschaften wie Funktionalität, Leistung, Benutzungsschnittstelle, Portierbarkeit, ... im Pflichtenheft
  - Bestimmen von Testfällen
  - Festlegung erforderlicher Dokumentationsdokumente

- **Ergebnisse**
  - Pflichtenheft = Anforderungsanalysedokument
  - Akzeptanztestplan
  - Benutzungshandbuch (1-te Version)

1.3.4 Systementwurf (system design/programming-in-the-large)

Im Systementwurf wird exakt festgelegt, wie die Funktionen der Software zu realisieren sind. Es wird der Bauplan der Software, die Software-Architektur, entwickelt.

- **Aufgaben**
  - Programmieren-im-Großen = Entwicklung eines Bauplans
  - Grobentwurf, der System in Teilsysteme/Module zerlegt
  - Auswahl bereits existierender Software-Bibliotheken, Rahmenwerke, ...
  - Feinentwurf, der Modulschnittstellen und Algorithmen vorgibt

- **Ergebnisse**
  - Entwurfsdokument mit Software-Bauplan
  - detaillierte(re) Testpläne

1.3.5 Codieren und Modultest (programming-in-the-small)

Die eigentliche Implementierungs- und Testphase, in der einzelne Module (in einer bestimmten Reihenfolge) realisiert und validiert werden.

- **Aufgaben**
  - Programmieren-im-Kleinen = Implementierung einzelner Module
  - Einhaltung von Programmierrichtlinien
  - Code-Inspektionen kritischer Modulteile (Walkthroughs)
  - Test der erstellten Module
- **Ergebnisse**
  - Menge realisierter Module
  - Implementierungsberichte (Abweichungen vom Entwurf, Zeitplan, ...)
  - Technische Dokumentation einzelner Module
  - Testprotokolle

1.3.6 Integrations- und Systemtest

Die einzelnen Module werden schrittweise zum Gesamtsystem zusammengebaut. Diese Phase kann mit der vorigen Phase verschmolzen werden, falls der Test isolierter Module nicht praktikabel ist.

- **Aufgaben**
  - Systemintegration = Zusammenbau der Module
  - Gesamtsystemtest in Entwicklungsorganisation durch Kunden (alpha-Test)
  - Fertigstellung der Dokumentation
- **Ergebnisse**
  - Fertiges System
  - Benutzerhandbuch
  - Technische Dokumentation
  - Testprotokolle

1.3.7 Auslieferung und Installation

Die Auslieferung (Installation) und Inbetriebnahme der Software beim Kunden findet häufig in zwei Phasen statt.

- **Aufgaben**
  - Auslieferung an ausgewählte (Pilot-)Benutzer (Beta-Test)
  - Auslieferung an alle Benutzer
  - Schulung der Benutzer
- **Ergebnisse**
  - Fertiges System
  - Akzeptanztestdokument

1.3.8 Wartung (Maintenance)

Nach der ersten Auslieferung der Software an die Kunden beginnt das Elend der Software-Wartung, das ca. 60% der gesamten Software-Kosten ausmacht.

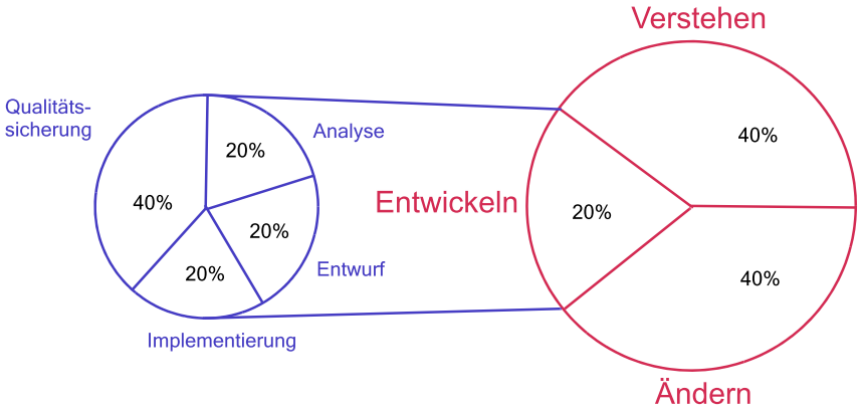
- **Aufgaben**
  - ca. 20% Fehler beheben (corrective maintenance)
  - ca. 20% Anpassungen durchführen (adaptive maintenance)
  - ca. 50% Verbesserungen vornehmen (perfective maintenance)
- **Ergebnisse**
  - Software-Problemberichte (bug reports)
  - Software-Änderungsvorschläge
  - Neue Software-Versionen

1.3.9 Probleme mit dem Wasserfallmodell

- zu Projektbeginn sind nur ungenaue Kosten- und Ressourcenschätzungen möglich
- ein Pflichtenheft kann nie den Umgang mit dem fertigen System ersetzen, das erste sehr spät entsteht (Risikomaximierung)
- es gibt Fälle, in denen zu Projektbeginn kein vollständiges Pflichtenheft erstellt werden kann (weil Anforderungen nicht klar)
- Anforderungen werden früh eingefroren, notwendiger Wandel (aufgrund organisatorischer, politischer, technischer, ... Änderungen) nicht eingeplant

- strikte Phaseneinteilung ist unrealistisch (Rückgriffe sind notwendig)
- Wartung mit ca. 60% des Gesamtaufwandes ist eine Phase

Abbildung 1: Andere Darstellung der Aufwandsverteilung



1.3.10 Typische Probleme in der Wartungsphase

- Einsatz wenig erfahrenen Personals (nicht Entwicklungspersonal)
- Fehlerbehebung führt neue Fehler ein
- Stetige Verschlechterung der Programmstruktur
- Zusammenhang zwischen Programm und Dokumentation geht verloren
- Zur Entwicklung eingesetzte Werkzeuge (CASE-Tools, Compiler, ...) sterben aus
- Benötigte Hardware steht nicht mehr zur Verfügung
- Resourcenkonflikte zwischen Fehlerbehebung und Anpassung/Erweiterung
- Völlig neue Ansprüche an Funktionalität und Benutzeroberfläche

1.4 Forward-, Reverse- und Reengineering

1.4.1 Software Evolution

- Wünsche
  - Wartung ändert Software kontrolliert ohne Design zu zerstören
  - Konsistenz aller Dokumente bleibt erhalten

- Wirklichkeit
  - Ursprüngliche Systemstruktur wird ignoriert
  - Dokumentation wird unvollständig oder unbrauchbar
  - Mitarbeiter verlassen Projekt

1.4.2 Forward Engineering

Beim Forward Engineering ist das fertige Softwaresystem das Ergebnis des Entwicklungsprozesses. Ausgehend von Anforderungsanalyse (Machbarkeitsstudie) wird ein neues Softwaresystem entwickelt.

1.4.3 Reverse Engineering

Beim Reverse Engineering ist das vorhandene Software-System der Ausgangspunkt der Analyse. Ausgehend von existierender Implementierung wird meist „nur“ das Design rekonstruiert und dokumentiert. Es wird (noch) nicht das betrachtete System modifiziert.

1.4.4 Reengineering

Reengineering befaßt sich mit der Sanierung eines vorhandenen Software-Systems bzw. seiner Neuimplementierung. Dabei werden die Ergebnisse des Reverse Engineerings als Ausgangspunkt genommen

Abbildung 2: Round Trip Engineering

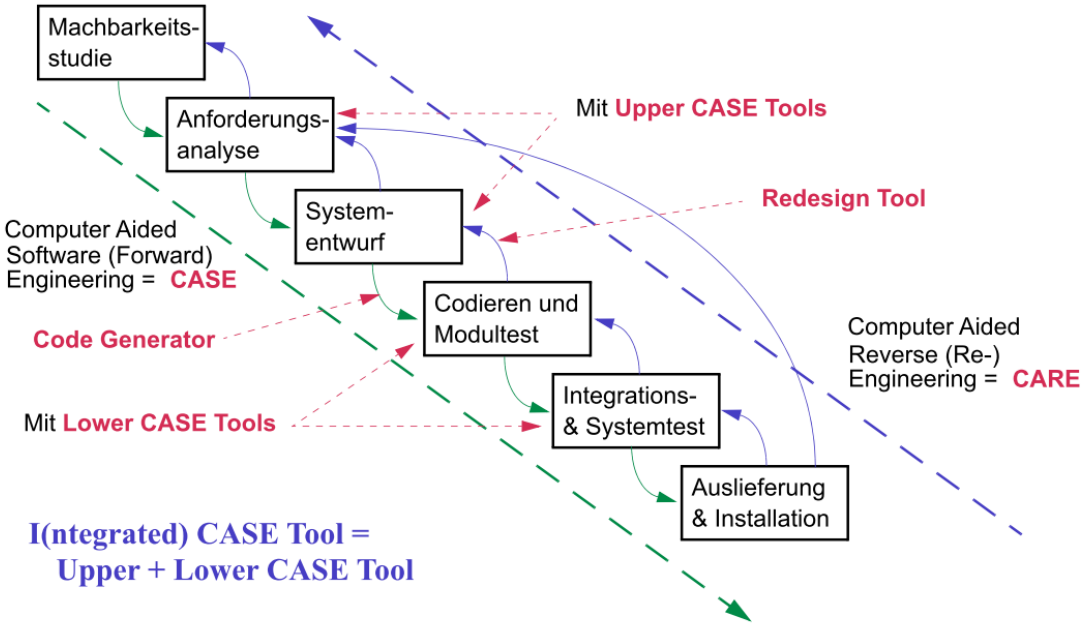
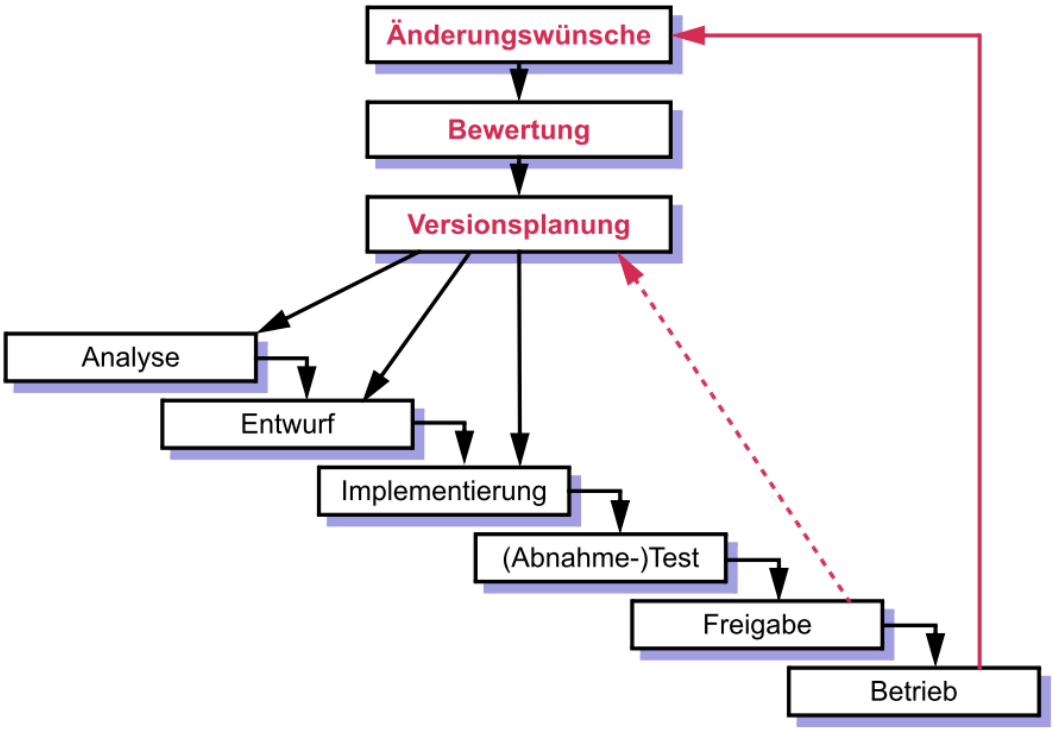


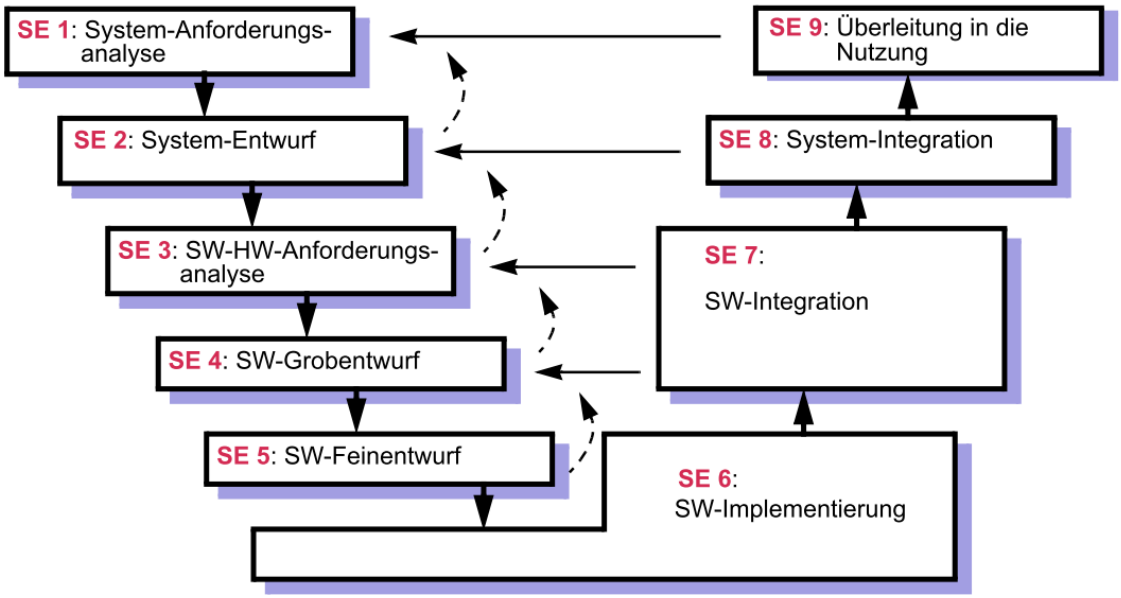
Abbildung 3: Einfaches Software-Lebenszyklus-Prozessmodell für die Wartung



1.4.5 Das V-Modell

- **Systemanforderungsanalyse**: Gesamtsystem einschließlich aller Nicht-DV-Komponenten wird beschrieben (fachliche Anforderungen und Risikoanalyse)
- **Systementwurf**: System wird in technische Komponenten (Subsysteme) zerlegt, also die Grobarchitektur des Systems definiert
- **Softwareanforderungsanalyse**: Technischen Anforderungen an die bereits identifizierten Komponenten werden definiert
- **Softwaregrobentwurf**: Softwarearchitektur wird bis auf Modulebene festgelegt
- **Softwarefeinentwurf**: Details einzelner Module werden festgelegt
- **Softwareimplementierung**: Wie beim Wasserfallmodell (inklusive Modultest)
- **Software-/Systemintegration:**: Schrittweise Integration und Test der verschiedenen Systemanteile
- **Überleitung in die Nutzung:**: Entspricht Auslieferung bei Wasserfallmodell

Abbildung 4: Das V-Modell



2 Konfigurationsmanagement

Beim Konfigurationsmanagement handelt es sich um die Entwicklung und Anwendung von Standards und Verfahren zur Verwaltung eines sich weiterentwickelnden Systemprodukts.

2.1 Einleitung

2.1.1 Fragestellungen

- Das System lief gestern noch; was hat sich seitdem geändert?
- Wer hat diese (fehlerhafte?) Änderung wann und warum durchgeführt?
- Wer ist von meinen Änderungen an dieser Datei betroffen?
- Auf welche Version des Systems bezieht sich die Fehlermeldung?
- Wie erzeuge ich Version x.y aus dem Jahre 1999 wieder?
- Welche Fehlermeldungen sind in dieser Version bereits bearbeitet?
- Welche Erweiterungswünsche liegen für das nächste Release vor?
- Die Platte ist hinüber; was für einen Status haben die Backups?

2.1.2 Definitionen

Definition von Software-KM nach IEEE-Standard 828-1988 SCM (Software Configuration Management) constitutes **good engineering practice** for all software projects, whether phased development, rapid prototyping, or ongoing maintenance. It enhances the reliability and quality of software by:

- Providing structure for **identifying and controlling** documentation, code, interfaces, and databases to support all life cycle phases
- Supporting a chosen **development/maintenance methodology** that fits the requirements, standards, policies, organization, and management philosophy
- Producing **management and product information** concerning the status of baselines, change control, tests, releases, audits etc.

Diese Definition ist jedoch nicht konkret und unabhängig vom Begriff "Software"

Definition nach DIN EN ISO 10007 KM (Konfigurationsmanagement) ist eine Managementdisziplin, die über die gesamte Entwicklungszeit eines Erzeugnisses angewandt wird, um Transparenz und Überwachung seiner funktionellen und physischen Merkmale sicherzustellen.
Der KM-Prozess umfasst die folgenden integrierten Tätigkeiten:

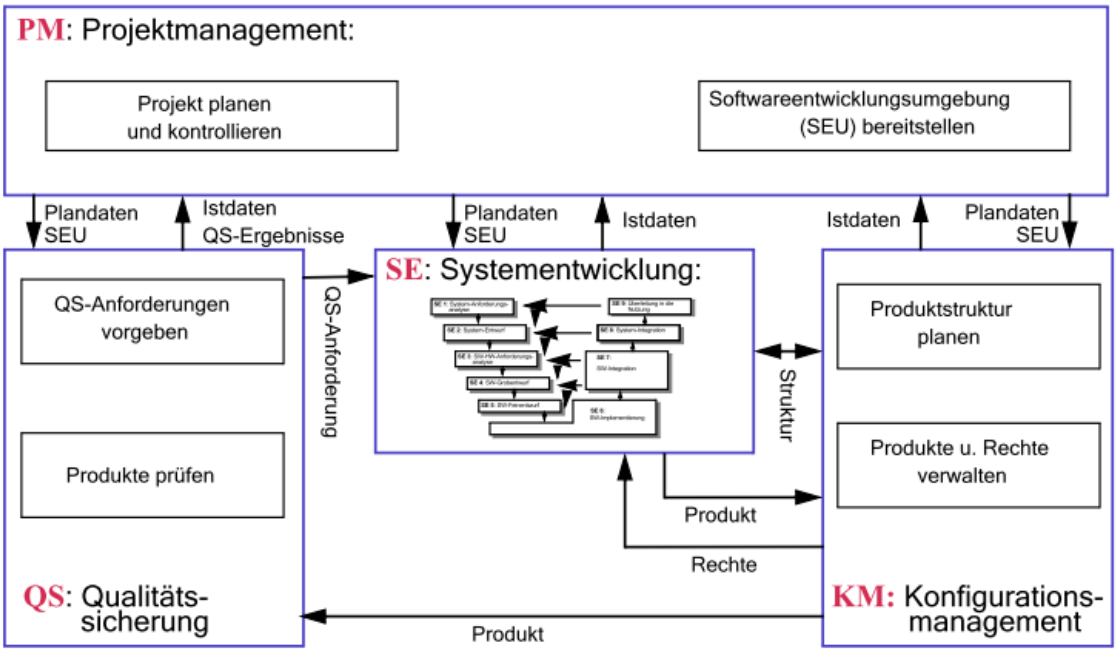
- **Konfigurationsidentifizierung**: Definition und Dokumentation der Bestandteile eines Erzeugnisses, Einrichten von Bezugskonfigurationen, ...
- **Konfigurationsüberwachung**: Dokumentation und Begründung von Änderungen, Genehmigung oder Ablehnung von Änderungen, Planung von Freigaben, ...
- **Konfigurationsbuchführung**: Rückverfolgung aller Änderungen bis zur letzten Bezugskonfiguration, ...
- **Konfigurationsauditierung**: Qualitätssicherungsmaßnahmen für Freigabe einer Konfiguration eines Erzeugnisses
- **KM-Planung**: Festlegung der Grundsätze und Verfahren zum KM in Form eines KM-Plans

Werkzeugorientierte Sicht auf KM-Aktivitäten

1. **KM-Planung**: Beschreibung der Standards, Verfahren und Werkzeuge, die für KM benutzt werden; wer darf/muss wann was machen
2. **Versionsmanagement**: Verwaltung der Entwicklungsgeschichte eines Produkts; also wer hat wann, wo, was und warum geändert
3. **Variantenmanagement**: Verwaltung parallel existierender Ausprägungen eines Produkts für verschiedene Anforderungen, Länder, Plattformen

4. **Releasemanagement**: Verwaltung und Planung von Auslieferungsständen; wann wird eine neue Produktversion mit welchen Features auf den Markt geworfen
5. **Buildmanagement**: Erzeugung des auszulieferenden Produkts; wann muss welche Datei mit welchem Werkzeug generiert, übersetzt, ... werden
6. **Änderungsmanagement**: Verwaltung von Änderungsanforderungen; also Bearbeitung von Fehlermeldungen und Änderungswünschen (Feature Requests) sowie Zuordnung zu Auslieferungsständen

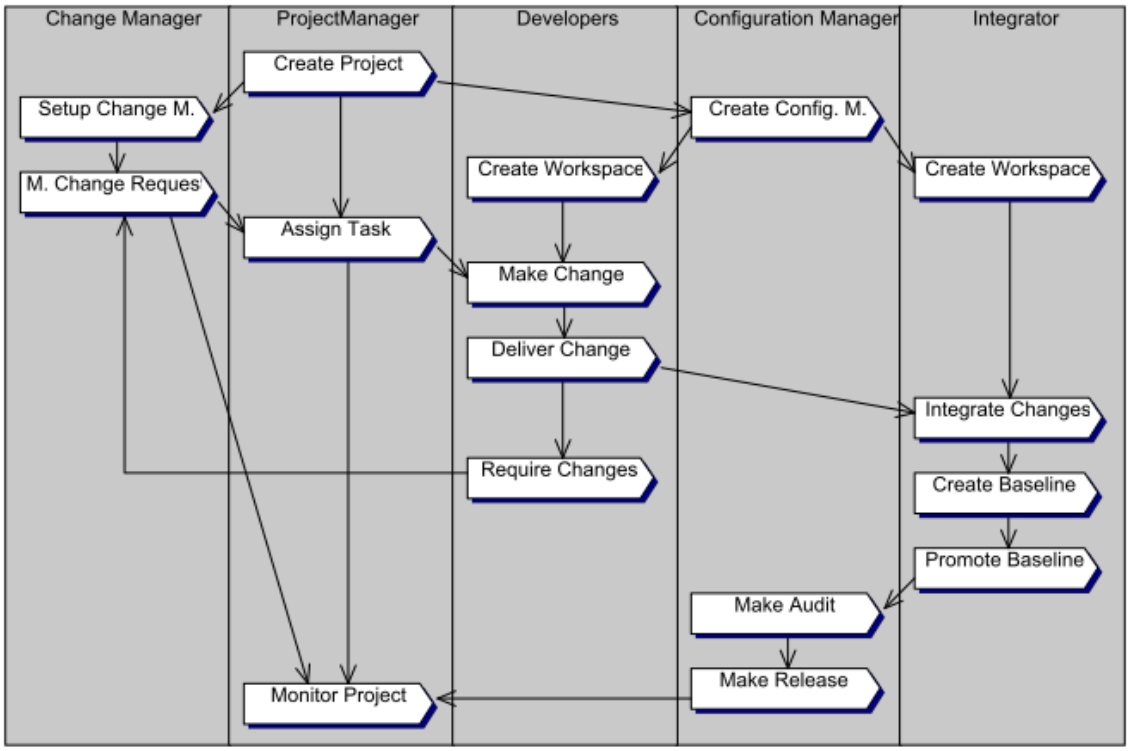
Abbildung 5: Integration des Konfigurationsmanagements im V-Modell



Workspaces für das Konfigurationsmanagement

- alle Dokumente (Objekte, Komponenten) zu einem bestimmten Projekt werden in einem gemeinsamen Repository (**public workspace**) aufgehoben
- im Repository werden nicht nur aktuelle Versionen, sondern auch alle **früheren Versionen** aller Dokumenten gehalten
- beteiligte Entwickler bearbeiten ihre eigenen Versionen dieser Dokumente in ihrem privaten Arbeitsbereich (private workspace, **developer workspace**)
- es gibt genau einen Integrationsarbeitsbereich (**integrations workspace**) für die Systemintegration

Abbildung 6: Grafische Übersicht über Aufgaben- und Rollenverteilung



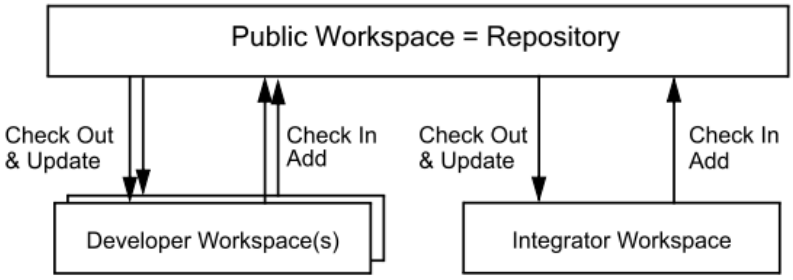
Aktivitäten bei der Arbeit mit Workspaces

- Personen holen sich Versionen neuer Dokumente, die von anderen Personen erstellt wurden(**checkout**), ih ihren privaten Arbeitsbereich
- Personen passen ihre Privatversionen ggf. von Zeit zu Zeit an neue Versionen im öffentlichen Repository an (**update**).
- Sie fügen (hoffentlich) nur konsistente Dokumente als neue Versionen in das allgemeine Repository ein (**checkin** = **commit**).
- Ab und an werden neue Dokumente dem Repository hinzugefügt (**add**).
- Jede Person kann alte/neue Versionen frei wählen.

Probleme

- Wie wird Konsistenz von Gruppen abhängiger Dokumente sichergestellt?
- Was passiert bei gleichzeitigen Änderungswünschen für ein Dokument?

Abbildung 7: Workspaces für das Konfigurationsmanagement



- Wie realisiert man die Repository-Operationen effizient?
- Wie unterstützt man "Offline"-Arbeit (ohne Zugriff auf Repository)?

Weitere Begriffe des Konfigurationsmanagements

- **Dokument** = Gegenstand, der der Konfigurationsverwaltung unterworfen wird (eine einzelne Datei oder ein ganzer Dateibaum oder ...)
- **(Versions-)Objekt** = Zustand einer Dokument zu einem bestimmten Zeitpunkt in einer bestimmten Ausprägung
- **Varianten** = parallel zueinander (gleichzeitig) existierende Ausprägungen eines Dokuments, die unterschiedliche Anforderungen erfüllen
- **Revisionen** = zeitlich aufeinander folgende Zustände eines Dokuments
- **Konfiguration** = komplexes Versionsobjekt, eine bestimmte Ausprägung eines Programmsystems (oft hierarchisch strukturierte Menge von Dokumenten)
- **Baseline** = eine Konfiguration, die zu einem Meilenstein (Ende einer Entwicklungsphase) gehört und evaluiert (getestet) wird
- **Release** = eine stabile Baseline, die ausgeliefert wird (intern an Entwickler oder extern an bestimmte Kunden oder ...)

2.2 Versionsmanagement

Bekannteste "open source"-Produkte (in zeitlicher Reihenfolge) sind:

- Source Code Control System **SCCS** von AT&T (Bell Labs):
  - effiziente Speicherung von Textdateiversionen als "Patches"
- Revision Control System **RCS** von Berkley/Purdue University

  - schnellerer Zugriff auf Textdateiversionen

- Concurrent Version (Control) System **CVS** (zunächst Skripte für RCS)
  - Verwaltung von Dateibäumen
  - parallele Bearbeitung von Textdateiversionen
- Subversion **SVN** - CVS-Nachfolger von CollabNet initiiert (http://www.collab.net)
  - Versionierung von Dateibäumen
- **Git**, Mercurial, ... als verteilte Versionsmanagementsysteme
  - jeder Entwickler hat eigene/lokale Versionsverwaltung

2.2.1 Source Code Control System SCCS von AT&T (Bell Labs)

Je Dokument (Quelltextdatei) gibt es eine eigene **History-Datei**, die alle Revisionen als eine Liste jeweils geänderter (Text-)Blöcke speichert:

- jeder Block ist ein **Delta**, das Änderungen zwischen Vorgängerrevision und aktueller Revision beschreibt
- jedes Delta hat **SCCS-Identifikationsnummer** der zugehörigen Revision: <ReleaseNo>.<LevelNo>.<BranchN

Revisionsbäume von SCSS

- Release 1.1 Neuentwicklung
- Release 1.1 Wartung
- Release 1.2 Weiterentwicklung
- Release 2 Weiterentwicklung

Erläuterungen zu "diff" und "patch"

- "**diff**"-Werkzeug bestimmt Unterschiede zwischen (Text-)Dateien = **Deltas**
- ein Delta(diff) zwischen zwei Textdateien besteht aus einer Folge von "Hunks", die jeweils Änderungen eines Zeilenbereichs beschreiben:
  - Änderungen von Zeilen: werden mit "**!**" markiert
  - Hinzufügen von Zeilen: werden mit "+" markiert
  - Löschen von Zeilen: werden mit "**-**" markiert
- reale Deltas enthalten unveränderte **Kontextzeilen** zur besseren Identifikation von Änderungsstellen

- ein **Vorwärtsdelta** zwischen zwei Dateien d1 und d2 kann als "**patch**" zur Erzeugung von Datei d2 auf Datei d1 angewendet werden
- inverses **Rückwärtsdelta** zwischen zwei Dateien d1 und d2 kann als "patch" zur Wiederherstellung von Datei d1 auf Datei d2 angewendet werden
- SCCS-Deltas sind in einer Datei gespeichert, deshalb weder Vorwärts- noch Rückwärts- sondern **Inline-Deltas**

Genauere Instruktionen zur Erzeugung von Deltas Jedes "diff"-Werkzeug hat seine eigenen Heuristiken, wie es möglichst kleine und/oder lesbare Deltas/Patches erzeugt, die die Unterschiede zweier Dateien darstellen. Ein möglicher (und in den Übungen verwendeter) Satz von Regeln zur Erzeugung von Deltas sieht wie folgt aus:

1. Die Anzahl der geänderten, gelöschten und neu erzeugten Zeilen aller Hunks eines **Deltas** zweier Dateien wird möglichst klein gehalten.
2. Jeder Hunk beginnt mit genau einer unveränderten **Kontextzeile** und enthält sonst nur geänderte, gelöschte oder neu eingefügte Zeilen (Ausnahme: Dateianfang).
3. Aufeinander folgende Hunks sind also durch jeweils **mindestens eine unveränderte Zeile** getrennt.
4. Optional: Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man die **Änderungsmarkierung "!"**

Durch diese Regeln nicht gelöstes Problem Wie erkenne ich, ob eine Änderung in Zeile i durch Einfügen einer neuen Zeile oder durch Ändern einer alten Zeile zustande gekommen ist?

Create- und Apply-Patch in Eclipse Die "Create Patch"- und "Apply Patch"-Funktionen in Eclipse benutzen genau das gerade eingeführte "Unified Diff"-Format. Dabei werden bei der Erzeugung von Hunks wohl folgende Heuristiken/Regeln verwendet:

- ein Hunk scheint in der Regel mit drei unveränderten Kontextzeilen zu beginnen (inklusive Leerzeilen).
- zwei Blöcke geänderter Zeilen müssen durch mindestens sieben unveränderte Zeilen getrennt sein, damit dafür getrennte Hunks erzeugt werden

Bei der Anwendung von Patches werden folgende Heuristiken/Regeln verwendet:

- werden der Kontext oder die zu löschenden Zeilen eines Patches so nicht gefunden, dann endet die Patch-Anwendung mit einer Fehlermeldung
- befindet sich die zu patchende Stelle eines Textes nicht mehr an der angegebenen Stelle (Zeile), so wird trotzdem der Patch angewendet

- gibt es mehrere (identische) Stellen in einem Text, auf die ein Patch angewendet werden kann, so wird die Stelle verändert, die am nächsten zur alten Position ist

Eigenschaften von SCSS

- für beliebinge (Text-)Dateien verwendbar (und nur für solche
- Schreibsperren auf "ausgecheckten" Revisionen
- Revisionsbäume mit manuellem Konsistenthalten von Entwicklungszweigen
- Rekonstruktionszeit von Revisionen steigt linear mit der Anzahl der Revisionen (Durchlauf durch Blockliste)
- Revisionsidentifikation nur durch Nummer und Datum

Offene Probleme

- Kein Konfigurationsbegriff und kein Variantenbegriff
- Keine Unterstützung zur Verwaltung von Konsistenzbeziehungen zwischen verschiedenen Objekten

Probleme mit Schreibsperren SCCS realisiert ein sogenanntes "pessimistisches" Sperrkonzept. Gleichzeitige Bearbeitung einer Datei durch mehrere Personen wird verhindert:

- ein Checkout zum Schreiben (**single write access**)
- mehrere Checkouts zum Lesen (**multiple read access**)

In der Praxis kommt es aber öfter vor, dass mehrere Entwickler dieselbe Datei zeitgleich verändern müssen (oder Person mit Schreibrecht "commit" vergisst...)

Unbefriedigende Lösungen

- Entwickler mit Schreibrecht macht "commit" unfertiger Datei, Entwickler mit dringendstem Änderungswunsch macht "checkout" mit Schreibrecht
  - inkonsistente Zustände in Repository, nur einer darf "Arbeiten"
- weitere Entwickler mir Schreibwunsch "stehlen" Datei, machen also "checkout" mit Leserecht und modifizieren Datei trotzdem
  - Problem: Verschmelzen der verschiedenen Änderungen

2.2.2 Revision Control System RCS von Berkley/Purdue University

Je Dokument (immer Textdatei) gibt es eine eigene History-Datei, die eine neueste Revision vollständig und andere Revisionen als **Deltas** speichert:

- optionale **Schreibsperren** (verhindern ggf. paralleles Ändern)
- **Revisionsbäume** mit besserem Zugriff auf Revisionen:
  - schneller Zugriff auf neueste Revision auf Hauptzweig
  - langsamer Zugriff auf ältere Revisionen auf Hauptzweig (mit **Rückwärtsdeltas**)
  - langsamer Zugriff auf Revisionen auf Nebenzweigen (mit **Vorwärtsdeltas**).
- Versionsidentifikation auch durch frei wählbare Bezeichner

Offene Probleme Kein Konfigurationsbegriff und kein Variantenbegriff

Abbildung 8: Deltaspeicherung von Revisionen als gerichtete Graphen