Softwaretechnik

Lernzusammenfassung

Lukas Bach
1. Oktober 2016

zum Modul Softwaretechnik I
am Karlsruher Institut für Technologie

1. Oktober 2016

Inhaltsverzeichnis

1 Einführung

Software sind Computer Programme, Prozeduren, Regeln sowie zur Operation von Computer Systemen gehörende Dokumentation und Daten.

System ist ein Ausschnitt aus der realen oder gedanklichen Welt, und beinhaltet reale Gegenstände (z.B. Menschen, Maschinen), Konzepte (z.B. Betriebssystem) oder Mischungen von beiden (z.B. Gesundheitssystem aus Ärzten, Krankenhäusern, Konventionen, Pharmaindustrie usw.).

Systemsoftware ist eine für spezielle Hardware entwickelte Software, z.B. Betriebssystem, Compiler... Nutzt die funktionalen Fähigkeiten der Hardware und erweitert diese.

Anwendungssoftware (application software) löst Probleme des Anwenders und setzt auf Systemsoftware.

Computersystem (Datenverarbeitungssystem, DV) verwendet besonders Anwendungssoftware und Systemsoftware, teils auch eigene Hardware.

Softwareentwicklung beschreibt die ausschließliche Entwicklung von Software.

Systementwicklung beschreibt die Entwicklung eines aus Software und Hardware bestehenden Systems.

Softwaretechnik ist die technologische und organisatorische Disziplin zur systematischen Entwicklung und Pflege von Softwaresystemen, die spezifizierte funktionale und nichtfunktionale Attribute erfüllen.

Softwareforschung ist die Bereitstellung von Bewertungen von *Methoden*, *Verfahren* und *Werkzeugen* für die Softwaretechnik.

1.1 Versionsverwaltung

Softwarekonfigurationsverwaltung ist die Disziplin zur Verfolgung und Steuerung der Evolution von Software.

Softwarekonfiguration ist eine eindeutige Menge von Softwareelementen mit jeweiligen Versionsangaben, die zu einem spezifisichen Zeitpunkt im Produktlebenszyklus aufeinander abgestimmt sind und gemeinsam eine vorgesehene Aufgabe erfüllen sollen.

Softwareelement (SE) ist ein *identifizierbarer Bestandteil* eines Produkts, z.B. eine Datei oder eine Konfiguration. SEs haben eindeutige Bezeichner, Änderungen am SE fordern neue Bezeichner. Man unterscheidet zwischen Quellelementen und Abgeleiteten (generierten) Elementen.

Einbuchen, Ausbuchen (Check-In, Check-Out) ermöglicht den Zugriff auf Softwareelemente. Lesezugriff der ausgebuchten Elemente wird für andere Teammitglieder nie dadurch gesperrt, höchstens der Schreibzugriff.

Striktes Ein-/Ausbuchen sperrt den Schreibzugriff für andere

Mitglieder.

Optimistisches Ein-/Ausbuchen ist ohne Änderungsreservierung, sodass gleichzeitig andere Mitglieder an dem Element arbeiten können. Fordert danach eventuell Verschmelzen der Versionen.

Deltaspeichern erspart Speicherplatz, indem nur eine vollständige Version und dann nur noch Änderungen gespeichert werden. Findet Anwendung bei SVC, nicht aber bei Git.

Vorwärtsdeltas: Speichere Grundversion und alle Änderungen daran.

Rückwärtsdeltas: Speichere aktuelle Version und Änderungen gegenüber früheren Versionen.

Subversion (SVN) ist ein *internetfähiges*, *zentralisiertes* VCS mit *atomarem Einbuchen* und *Optimistischem Ausbuchen*. Es versioniert das gesamte Depot, einschließlich Verschieben, Umbenennen und Kopieren. Es hat keine Semantik für Variantenbildung und erfordert Konventionsmäßig festgelegte Unterverzeichnisse “trunk “, “branches “und “tags “.

Git ist ein *internetfähiges*, *dezentralisiertes* VCS mit *atomarem Einbuchen*, *Optimistischem Ausbuchen*, *integrierter Variantenbildung* und *kryptografischer Sicherung*. Depotkommunikation erfolgt mit den Befehlen “push “, “pull “, “fetch “. Zu sichernde Dateien durchqueren die Bereiche “working directory “, “staging area “, “local repo “, (“remote repo “) und können die Zustände “untracked “, “unmodified “, “modified “, “staged “haben.

1.2 Wasserfallmodell

Wasserfallmodell In SWT als allgemeines Lehrmodell verwendete Softwareentwicklungsprozedur:

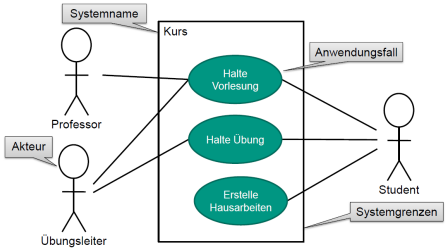
1. Planung
   - Anwendungsfall ausgedrückt durch Anwendungsdomänenobjekte
   - Dokumente: Lastenheft, Projektplan, Kalkulation, Durchführbarkeitsstudie
   - Ziel: Beschreibe das zu entwickelnde System in Worten des Kunden und prüfe Durchführbarkeit
2. Definition
   - Anwendungsfall strukturiert durch Teilsysteme
   - Dokumente: Pflichtenheft, GUI-Beschreibung, Benutzerhandbuch
3. Entwurf
   - Anwendungsfall realisiert durch Lösungsdomänenobjekte
   - Arichtektur, UML-Modelle, Modulführer
4. Implementierung

- Anwendungsfall implementiert durch Quellcode
- Komponenten, Dokumentation, Testdaten, Code

5. Testen
   - Anwendungsfall getestet durch Testfälle
   - fertiges System, Testprotokoll
6. Abnahme, Einsatz und Wartung

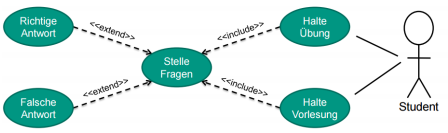
Szenario ist die Beschreibung eines Ereignisses oder einer Folge von Ereignissen oder Aktionen, definiert aus Sicht des Benutzers, können auch in der Testphase und in der Auslieferung verwendet werden.

Anwendungsfalldiagramm stellt das Verhalten des Systems dar. Besteht aus Aktueren (Rolle des Benutzers oder anderen Systemen) und allen Anwendungsfällen (Funktionen des Systems), welche die Systemfunktionalität darstellen.



«include» Beziehung beschreibt für ein Verhalten die interne Notwendigkeit für ein weiteres.

«extends» Beziehung beschreibt für ein Verhalten, dass es eine Konkretisierung eines anderen ist.



Anforderungen Verschiedene Arten:

Funktionale Anforderung: Interaktion zwischen System und Systemumgebung, vom System unterstützte Benutzeraufgaben. "Benutzer muss Einträge erstellen können. "

Nichtfunktionale Anforderung: Eigenschaften des Systems/-Domäne. "Systemabsturz darf nicht zu Datenverlust führen. ". Arten: Benutzbarkeit, Zuverlässigkeit, Geschwindigkeit, Wartbarkeit.

Einschränkungen sind durch den Kunden oder die Umgebung gegeben. Arten: Implementierung, Schnittstellen, Einsatzumgebung, Lieferumfang, Rechtliches.

Validierung von Anforderungen Wird nach Planungsphase oder Definitionsphase durchgeführt, prüft: Korrektheit, Vollständigkeit, Konsistenz, Eindeutigkeit, Realisierbarkeit, Verfolgbarkeit (Systemfunktionen sind Anforderungen zuordnungsbar).

Lastenheft entsteht in der Planungsphase. Gliederung:

1. Zielbestimmung
2. Produkteinsatz
3. Funktionale Anforderungen

4. Produktdaten
5. Nichtfunktionale Anforderungen
6. Systemmodelle
   a) Szenarien
   b) Anwendungsfälle
7. Glossar

Durchführbarkeitsuntersuchung prüft die *fachliche Durchführbarkeit*, *alternative Lösungsvorschläge*, *personelle Durchführbarkeit*, *Risiken*, *ökonomische Durchführbarkeit*, *rechtliche Gesichtspunkte*.

Pflichtenheft entsteht in der Definitionsphase. Es erweitert das Lastenheft, indem es das System genau genug modelliert, sodass Entwickler ohne Nachfragen implementieren können. Gliederung:

1. Zielbestimmung
2. Produkteinsatz
3. Produktumgebung
4. Funktionale Anforderungen
5. Produktdaten
6. Nichtfunktionale Anforderungen
7. Globale Testfälle
8. Systemmodelle
   a) Szenarien
   b) Anwendungsfälle
   c) Objektmodell
   d) Dynamische Modelle
   e) Benutzerschnittstelle - Bildschirmskizzen, Navigationspfade
9. Glossar

Modellarten Die im Pflichtenheft verwendbaren Modelle sind:

Funktionales Modell Szenarien und Anwendungsfalldiagramme (Lastenheft).

Objektmodell Klassen- und Objektdiagramme.

Dynamisches Modell Sequenz-, Zustands-, Aktivitätsdiagramme.

2 UML

2.1 Klassendiagramm, Objektdiagramm

Objekt ist ein Exemplar einer Klasse, erkennbares, eindeutig unterscheidbares, bestimmbares Element aus einer Grundmenge.

Exemplar ist ein konkretes Element einer Klasse.

Klasse ist ein abstraktes Modell für eine Reihe von ähnlichen Objekten.

Grundmenge: alles Substanzielles und Konzeptuelles.

Attribut ist ein für alle Exemplare einer Klasse def. Eigenschaft. Notation: `Attributname: Typ [=Wert];`.

Domäne sind die Werte, die ein Attribut annehmen kann.

Gleichheit n-ter Stufe: Zwei Objekte

sind auf n-ter Stufe gleich, wenn:

0-ter Stufe: Identität

1-ter Stufe: Identität (0-te Stufe gleich) oder alle Attribute identisch (0-te Stufe gleich).

2-ter Stufe: 1-te Stufe gleich oder alle Attribute 1-te Stufe gleich.

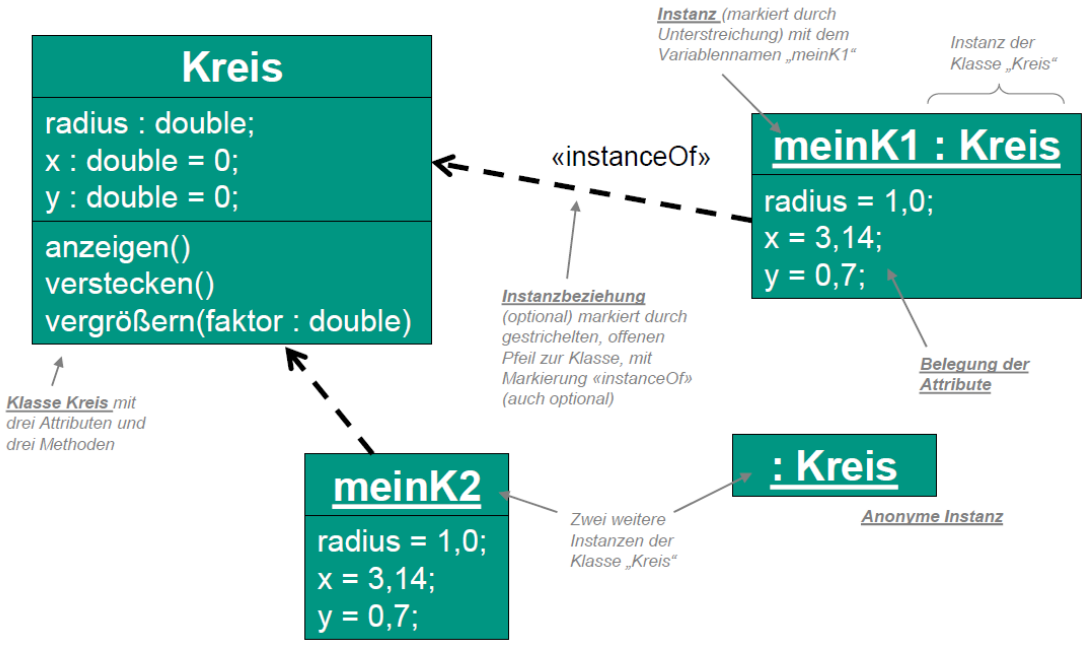
n-ter Stufe: $(n-1)$-te Stufe gleich oder alle Attribute $(n-1)$-te Stufe gleich.

Zustand eines Objekts: Solange sich ein Objekt in einem Zustand befindet, reagiert es bei jedem Aufruf gleich auf seine Umwelt.

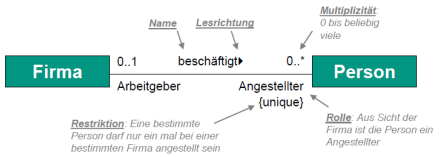
Kapselungsprinzip : Der Zustand ist nach außen sichtbar, wird aber im Inneren des Objektes verwaltet und kontrolliert geändert.

Methodensignatur `Methodenname(Parameterliste) : Rückgabetyp;`, Parameterliste: `Attributname : Typ, ...`

Objekt-/Instanzdiagramm stellt Verhältnisse zwischen Klassen und Instanzen derer dar. (Siehe Bild) Die $<< instanceof >>$ Beziehung wird auch oft als Verknüpfung durch einfachen Strich gezeichnet.



Assoziation stellt ein Verhältnis zwischen zwei Klassen dar. Rollen sind dabei die Bezeichner, die die beiden Klassen in der Beziehung beschreiben, Multiplizitäten stellen dar in welcher Häufigkeit sie zueinander stehen (eindeutig 1:1, mehrdeutig 1:n, m:n).



Bei UML Diagrammen sollte man Assoziationen eher als Attribute nehmen, falls man Transaktionali-

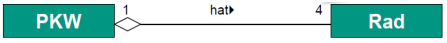
tät, mehrere Gegenüber oder Rücknavigation vom Gegenüber braucht.

Transaktionalität Eine Assoziationsänderung ist *Unteilbar* (Atomicity), hinterlassen *konsistente* Zustände (Consitency), laufen bei Parallelität *ohne Beeinflussung anderer Fäden* ab (Isolation), und sind nach Abschluss für alle Programmteile *dauerhaft* zu sehen (Durability). $\Rightarrow ACID$.

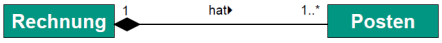
{**ordered**} Sei $p \subseteq K_1 \times K_2$ die auf K_2 geordnete Assoziation, dann $\exists$ lineare Ordnung $\sigma \subseteq K_2^2$, sodass: $\forall x \in K_1 \forall y, z \in K_2 : xpy \wedge xpz \Rightarrow y\sigma z \vee z\sigma y$.

{**unique**} Sei $p \subseteq K_1 \times K_2$ die auf K_2 einzigartig zugewiesene Assoziation, dann $\forall x \in K_1 \forall y, z \in K_2 : xpy \wedge xpz \Rightarrow y \neq z$.

Aggregation: Assoziation mit Teil-Ganzes-Beziehung, wird mit leerer Raute dargestellt.

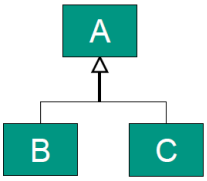


Komposition: Strengere Aggregation, Teil darf nicht ohne Ganzes existieren, wird mit voller Raute dargestellt.



Qualifizierte Assoziation: Assoziation, bei der die referenzierten Objekte durch Qualifizierer ansprechbar sind.

Vererbung: Eine Klasse A erbt dann von einer Klasse B, wenn [Funktionalität und Zustand von A] $\subseteq$ [Funktionalität und Zustand von B].



Klassenattribute, -methoden: Klasse selbst wird als Objekt aufgefasst und hat eigene Methoden und Attribute. In UML werden solche Elemente unterstrichen dargestellt.

Konstruktoren: Erstellmethode der Klasse, wird mit Präfix << *create* >> dargestellt.

Liskovsches Substitutionsprinzip: In einem Programm, in dem U eine Unterklasse von K ist, kann jedes Exemplar der Klasse K durch ein Exemplar von U ersetzt werden, wobei das Programm weiterhin korrekt funktioniert.

Signaturvererbung: Eine in der Oberklasse definierte und evtl. implementierte Methode vererbt seine Signatur.

Implementierungsvererbung: Eine in der Oberklasse definierte und evtl. implementierte Methode vererbt seine Signatur und Implementierung.

Überschreiben: Eine geerbte Methode wird unter gleicher Signatur neu implementiert.

Überladen: Eine bereits existente Methode wird unter anderer Signatur neu definiert.

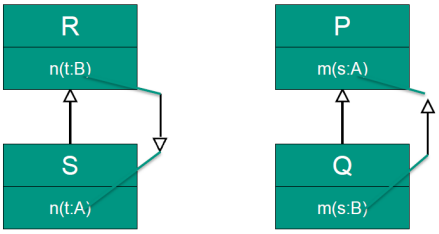
Parameter-Varianz: Parameter (oder auch Rückgabetyp) ändert sich bei Methodenüberschreibung.

Varianz Parametertypen ändern sich.

Kovarianz Parametertypen werden spezieller.

Kontravarianz Parametertypen werden allgemeiner.

Invarianz Parametertypen ändern sich nicht.



Um dem Substitutionsprinzip zu genügen muss Varianz invariant oder:

Eingabeparameter Kontravarianz (Unterklasse hat *schwächere Vorbedinungen*)

Ausgabeparameter Kovarianz (Unterklasse hat *stärkere Nachbedinung*), gilt auch für Ausnahmen

Polymorphie Vielgestaltigkeit.

statisch Überladung von Methoden, es kann mehrere Methoden mit demselben Namen geben, sofern Signatur varriert. Es wird die Methode mit der korrekten Signatur verwendet.

dynamisch Bei Vererbung, es wird diejenige Methode mit gegebener Signatur verwendet, die in der Vererbungshierarchie am speziellsten ist.

Sichtbarkeit Attribute und Methoden können ihren Zugriff schützen:

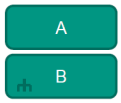
-privat (alle!) Exemplare derselben Klasse.

#geschützt Exemplare derselben Klasse, aller abgeleiteten Klassen, Klassen anderer Pakete.

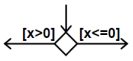
+public Jedes Exemplar.

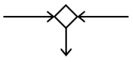
2.2 Aktivitätsdiagramm

- Aktionen
  - Elementare Aktion
  - Verschachtelte Aktion
- Knoten
  - Startknoten
    - Startpunkt eines Ablaufs
  - Endknoten
    - Beendet alle Aktionen und Kontrollflüsse
  - Ablaufende
    - Beendet einen einzelnen Objekt- und Kontrollfluss

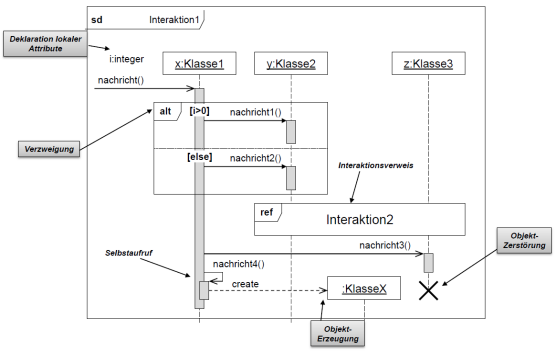


- Entscheidung
  - bedingte Verzweigung
- Zusammenführung
  - „oder“-Verknüpfung
- Teilung
  - Aufteilung eines Kontrollflusses
- Synchronisation
  - „und“-Verknüpfung





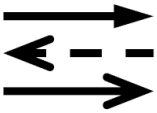
2.3 Sequenzdiagramm



Operator	Bed./Parameter	Bedeutung
alt	[bed.1], [bed.1], … [else]	Nur eine der Alternativen wird ausgeführt.
break	[bedingung]	Ist die Bedingung wahr, dann wird nur der Block ausgeführt und anschließend endet das Szenario.
opt	[bedingung]	Optionale Sequenz. Die Teilsequenz wird nur ausgeführt, wenn die Bedingung wahr ist.
par		Enthaltene Teilsequenzen werden parallel ausgeführt.

Nachrichtentypen

- Synchrone Nachrichten
- Antworten (optional)
- Asynchrone Nachrichten



2.4 Zustandsdiagramm

Klar.

2.5 Verwendung von UML

Zur Verwendung von Aktivitätsdiagramm, Sequenzdiagramm, Zustandsdiagramm.

Dynamisches Modell Verwendung dieser Diagramme:

Aktivitätsdiagramm: Beschreibe parallele und sequentielle Abläufe.

Sequenzdiagramm: Identifiziere Methoden zwischen Objekten, zeige Aufrufabläufe zwischen Objekten.

Zustandsdiagramm: Zeige Zustandsübergänge innerhalb eines einzelnen Objekts.

Finden von Klassen .

- Klasse lässt sich als konkretes Objekt identifizieren.
- Syntaktische Analyse.
- Linguistische Analyse nach thematischen Rollen.
- Inhaltliches Durchforsten nach Attributen für potenzielle Klassen und Aktuere.

Anzeichen, dass es sich nicht um Klasse handelt:

- Weder Attribute noch Operationen lassen sich identifizieren.
- Klasse enthält dieselben Attribute, Operationen, Restriktionen und Assoziationen wie eine andere Klasse.
- Klasse modelliert Implementierungsdetails.
- Klasse enthält nur wenige Attribute $\Rightarrow$ vlt. anderer Klasse

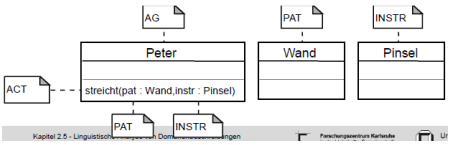
zuordnen?

Modellierung von Subsystemen: Subsysteme fassen einzelne Klassen mit gemeinsamen Bezug zu einem Paket zusammen. *Innerhalb* eines Subsystems soll *starke Kohäsion* herrschen, *zwischen Subsystemen* dagegen *schwache Kopplung*.

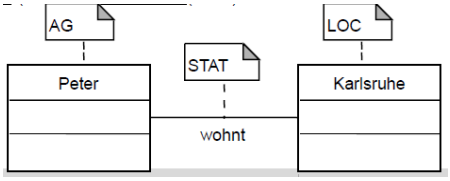
2.6 Linguistische Analyse

Rolle	Bezeichnung	UML-Umsetzung
agens	Der Handelnde	Klasse
patiens	Der Behandelte	Klasse, Methodenoparameter bei actus
actus	Die Handlung	Methode bei agens
instrumentum	Hilfsmittel	Klasse, Methodenparameter beim actus
status	Zustandsverben und Nominalisierung	Beziehung zw. Klassen
locus	Ort-/Positionsangabe	Klasse
omnium	Das Ganze	Klasse
pars	Ein Teil	Klasse
omnium + pars		Komposition
donor	Der Gebende	
recipiens	Der Empfangende	
habitum	Das Übergebene	
donor + recipiens + habitum		3 Klassen mit mehrstelliger Assoziation und 3 Methoden zum Geben, Nehmen und Übergeben werden.
creator(+)	Erzeuger	Klasse
creator(-)	Zerstörer	Klasse
opus	Erzeugte/zerstörte Werk	Klasse
creator + opus (+actus)		Methode zum erzeugen/zerstören des Werks beim creator

Beispiel AG, ... Peter (AG) streicht (ACT) die Wand (PAT) mit einem Pinsel (INSTR).

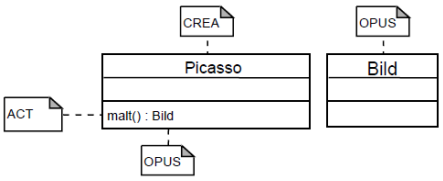


Beispiel AG, ... Peter (AG) wohnt (STAT) in Karlsruhe (LOC).

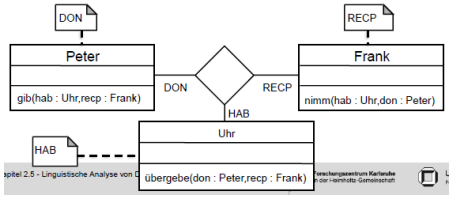


Beispiel OMN, PARS Peter puzzelt ein Puzzle (OMN) aus 1000 Teilen

Beispiel CREA, OPUS Picasso (CREA+) malt (ACT) ein Bild (OPUS).



Beispiel DON, RECP, HAB Peter (DON) schenkt Frank (RECP) eine Uhr (HAB).



3 Entwurf

3.1 Modularer und objektorientierter Entwurf

Softwarearchitektur gliedert ein Softwaresystem in Komponenten (Module oder Klassen) und Subsysteme (Pakete, Bibliotheken). Durch sie werden Benutzt-Relationen zwischen Komponenten aufgestellt. Auch als *Grobentwurf* bezeichnet.

Entwurfsmethoden Man unterscheidet zwischen:

- Modularer Entwurf (im folgenden verwendet)
- Objekt-orientierter Entwurf

Systemarchitektur beim modularen Entwurf Baut sich auf aus:

Modulführer: (Grobentwurf) Gliederung der Kompenente und Subsysteme, Beschreibung der Funktion jeden Moduls, verwendet Entwurfsmuster

Modulschnittstellen: Beschreibung zu den Schnittstellen aller Module, informell oder formal.

Benutztrelationen: Gliederung in Kompenenten des Subsystems, zeige, welche Module sich gegenseitig benutzen. Sollte ein azyklischer gerichteter Graph sein, um Inkrementalität zu ermöglichen.

Feinentwurf: (Optional) Beschreibe modul-interne Datenstrukturen und Algorithmen mit Pseudocode.

Diese Entwurfselemente werden gegliedert in:

Externer Entwurf Grobentwurf und Modulschnittstellen

Interner Entwurf Benutztrelationen und Feinentwurf

Modul ist eine Menge von Programmelementen, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden.

Geheimnisprinzip, Kapselungsprinzip:

Jedes Modul verbirgt eine wichtige Entwurfsentscheidung hinter einer wohldefinierten Schnittstelle, die sich bei einer Änderung der Entscheidung nicht mit ändert.

Modulschnittstellenbeschreibung .

- Liste der öffentlichen Programmelemente.
- Ein-/Ausgabeformate (falls E/A Modul).
- Parameter und Rückgabewerte der Unterprogramme/Operationen.
- Beschreibung des Effekts der Unterprogramme.
- (evtl.) Zeitverhalten, Genauigkeit, Speicherplatzbedarf etc.
- Fehlerbeschreibung und Fehlerbehandlung.
- Ausnahmen, die ausgelöst und nicht behandelt werden.

Benutztrelation: Programmkompenente A benutzt Programmkompenente B genau dann, wenn A für den korrekten Ablauf die Verfügbarkeit einer korrekten Implementierung von B erfordert.

Benutzthierarchie ist eine *zyklenfreie* Benutztrelation.
Zur Konstruktion einer Benutzthierarchie: A soll B benutzen, wenn:

- A durch die Benutzung von B einfacher wird,
- B durch die Benutzung durch A nicht viel komplexer wird,
- es min. eine nützliche Untermenge gibt, die B, aber nicht A enthält,
- und es keine Untermenge gibt, die A, aber nicht B enthält.

Benutztrelationen sollen immer zyklenfrei sein, um Inkrementalität und Bilden von Untermengen zu ermöglichen.

Geheimnisprinzip und Benutztrelationsdefinitionen gelten auch bei Objektorientierung. Analoga zum Modul sind hier Klasse und Paket, zu Modulschnittstellen die Schnittstellen der Klassen, abstrakte Klassen und reine Schnittstellen. Anstelle des Modulführers wird der Paket- und Klassenführer als UML-Klassen- und Paketdiagramm dargestellt und mit Text dokumentiert.

3.2 Architekturstyle

3.2.1 Definitionen

Abstrakte Maschine oder virt. Maschine ist eine Menge von Softwarebefehlen und -objekten, die auf einer darunterliegenden (abstrakten oder realen) Maschine aufbauen und diese ganz oder teilweise verdecken können. (z.B. Programmiersprache, OS, Java VM, Anwendungskern)
Benutztrelationen zwischen mehreren abstrakten Maschinen ist hierarchisch.

Programmfamilie oder Softwareproduktlinie ist eine Menge von Programmen, die erhebliche Anteile von Anforde-

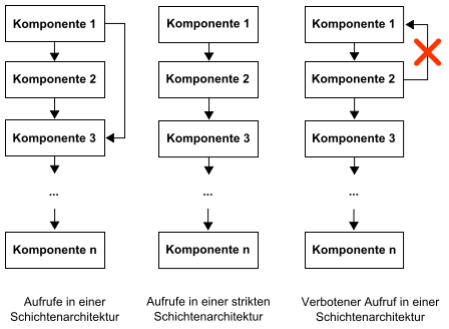
rungen, Entwurfsbestandteilen oder Softwarekompenenten gemeinsam haben.
Das ermöglicht einfache Adaption bestehender Software ausgehend von anderen Mitgliedern der bestehenden Familie.
Familienmitglieder unterscheiden sich extern in Hardwarekonfigurationen, Ein-/Ausgabeformat, Funktionsumfang, intern in Datenstrukturen und Algorithmen.

Allgemeines Programm: Programm ist in vielen Situationen ohne Änderung nutzbar (meist hohe Laufzeit- und Speicherkosten).

Flexibles Programm: Programm ist leicht für viele Situationen abänderbar (meist hohe Entwurfskosten).

3.2.2 Architekturauflistung

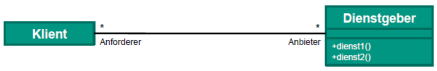
Schichtenarchitektur gliedert die Software in mehrere übereinander liegende Schichten. Jede Schicht hat eine wohldefinierte Schnittstelle und stellt diese darüber liegenden Schichten zur Verfügung. Schichten nutzt nur Dienste von niedrigeren Schichten (falls intransparent: nur die nächst untere Schicht), niemals die von höheren Schichten.



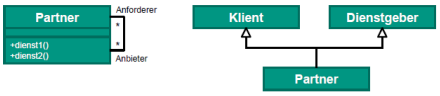
3-Schichten-Architektur ist eine Schichtenarchitektur aus Benutzerschnittstelle, Anwendungskern, Datenbanksystem als Schichten.

3-stufige Architektur ist einer 3 Schichten Architektur, bei der alle Schichten auf seperaten Rechnern laufen.

Klient/Dienstgeber (Client/Server) besteht aus mehreren Dienstgebern, die Dienste an Klienten (Subsysteme) anbieten.



Partnernetze (Peer-to-peer Networks) ist eine Verallgemeinerung von Klient/Dienstgeber, alle Subnetze sind gleichberechtigt. Jedes Subsystem ist gleichzeitig Klient und Dienstgeber.



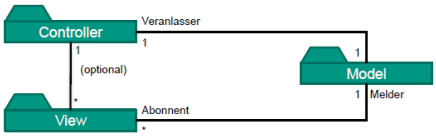
Eigenschaften: +Autonomie der Partner, -Zuverlässigkeit, -Verfügbarkeit

Modell-Präsentations-Steuerung (MVC) besteht aus:

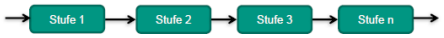
Modell, Model speichert die anwendungsspezifischen Daten.

Präsentation, View stellt die Objekte in der Anwendung dar.

Steuerung, Controller steuert die Benutzerinteraktion, aktualisiert das Modell, stößt das Melden von Änderungen am Modell an die Präsentation an.



Fließband (Pipeline) ist eine Reihe von eigenständiger Prozesse/Fäden, durch die Daten fließen und weiter verarbeitet werden.



Rahmenarchitektur (Framework) ist ein fast fertiges Programm, das durch Einfüllen vorgesehener Lücken, Erweiterungen, oder Überschreiben abstrakter Methoden individualisiert werden kann. Oft verwendet, wenn die Grundversion der Anwendung schon funktionsfähig sein soll.

3.3 Entwurfsmuster

Im folgenden werden einzelne Entwurfsmuster nicht genauer erläutert, stattdessen nur den Kategorien zugeordnet und diese erklärt.

Entkopplungsmuster teilen das System in mehrere Einheiten, die dadurch unabhängig verändert, ausgetauscht und wiederverwendet werden können.

- Adapter
- Beobachter
- Brücke
- Iterator
- Stellvertreter
- Vermittler

Variantenmuster ziehen Gemeinsamkeiten von verwandten Einheiten heraus und beschreiben diese an einer einzigen Stelle. Dies ermöglicht einheitliche Schnittstellen.

- Abstrakte Fabrik
- Besucher
- Fabrikmethode
- Kompositum
- Schablonenmethode
- Strategie
- Dekorierer

Zustandshandhabungsmuster bearbeiten den Zustand von Objekten, unabhängig von deren Zweck.

- Einzelstück
- Fliegengewicht
- Memento
- Prototyp
- Zustand

Steuerungsmuster steuern den Kontrollfluss und bewirken, dass zur rechten Zeit die richtigen Methoden aufgerufen werden.

- Befehl
- Master/Worker

Bequemlichkeitsmuster sparen Schreib- und Denkarbeit.

- Bequemlichkeits-Klasse
- Bequemlichkeits-Methode
- Fassade
- Null-Objekt

4 Implementierung

Assoziationen implementieren: Assoziationen werden bei der Implementierung wie folgt auf Instanzvariablen der Klasse abgebildet:

Multiplizität 0..1 Instanzvariable.

Multiplizität 1 Instanzvariable, deren Referenz niemals null sein darf.

Multiplizität * Instanzvariable, die eine List von Referenzen speichert.

{**ordered**} Geordnete Listenimplementation, z.B. `ArrayList` oder `Vector`.

{**unique**} Einzigartige Listenimplementation, z.B. `HashSet` oder `TreeSet`.

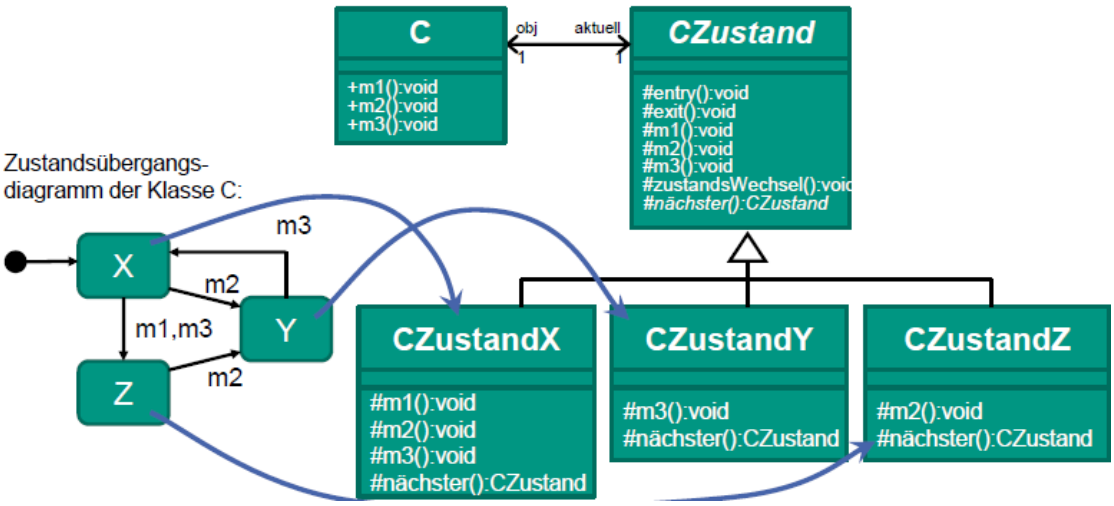
Qualifizierte Assoziation Abbildende Referenzliste, z.B. `HashMap`.

Implizite Speicherung: Objektzustand kann aus Attributwerten hergeleitet bzw. berechnet werden, wird aber nicht direkt abgespeichert. Zustandsübergangsfunktion ist implizit.
⇒ speichereffizienter, komplexer, nicht immer möglich.

Explizite Speicherung: Objektzustand wird in dedizierten Instanzvariablen gespeichert und kann direkt abgerufen werden. Zustandsübergangsfunktion ist explizit.
⇒ laufzeiteffizienter, umfangreicher, offensichtlicher, immer möglich.

Eingebette Zustandsspeicherung. Jede Methode (=Zustandsübergangsfunktion) implementiert große Fallunterscheidung, um zu entscheiden, in welchen Zustand sie sich befindet. Kompakter und schneller.

Ausgelagerte Zustandsspeicherung: Jeder Zustand ist ein eigenes Objekt, Code für Zustandsübergangsfunktionen (Äste der Fallunterscheidung) sind auf gleichnamige Methoden in den Zustandsklassen verteilt (Strategie-muster). Flexibler, übersichtlicher. (Siehe Bild)



4.1 Eingebettete Zustandsspeicherung implementieren

```
class Automat {
  private Zustand zustand;
  public void a() {
    if (zustand == Zustand.X) throw new IllegalStateException();
    if (zustand == Zustand.Y) // mach was
    if ...
  }
  public void b() {..} public void c() {..}
}
```

4.2 Ausgelagerte Zustandsspeicherung implementieren

```
class Automat {
  protected CZustand aktuell = new CZustand(this);
  public void a() {
    aktuell.a();
    aktuell.zustandswechsel();
  }
  public void b() {..} public void c() {..}
}

abstract CZustand {
  private Automat obj; // Bei Konstruktion speichern
  protected void entry() {/*leer*/}
  protected void exit() {/*leer*/}

  protected void a() {
    throw new IllegalStateException();
    // Standardverhalten: Fehler
  }
  protected void b() {..} protected c() {..}

  protected void zustandswechsel() {
    CZustand = naechster(); // Einschubmethode
    if (n != this) {
      this.exit();
      n.entry();
      obj.aktuell = n;
    }
  }

  protected abstract CZustand naechster() {}
}

class CZustandX extends CZustand {
  private CZustand next;
  protected void a() {
    // ...
    next = new CZustandB(obj);
  }

  protected void b() {..} protected void c() {..}

  protected CZustand naechster() { return next; }
}
```

4.3 Parallelisierung

Moore's Law: "The complexity for minimum component cost has increased at a rate of roughly a factor of two per year. "(1965)
"The new slope might approximate a doubling every two years, rather than every year, by the end of the decade. "(1975)
Neue Version: Verdopplung der Anzahl Prozessoren pro Chip mit jeder Chip-Generation bei etwa gleicher Taktfrequenz.

parallel: Nebeneinander verlaufend, in gleichem Abstand. gleichzeitig ablaufend, simultan.

Gemeinsamer Speicher: (shared memory) verschiedene Prozessoren ha-

ben einen gemeinsam genutzten Speicherbereich.

Verteilter Speicher: Jeder Prozessor hat einen dedizierten Speicher. Zur Kommunikation schicken Prozessoren sich Nachrichten.

Prozess wird vom Betriebssystem erzeugt und enthält Informationen über Programmressourcen und Ausführungszustand.

Kontrollfaden (Thread) ist eine Instruktionsfolge, die ausgeführt wird und innerhalb von einem Prozess existiert. Ein Faden hat einen eigenen Befehlszeiger, Keller und Register, teilt sich mit anderen Fäden aber Adressraum, Code/Daten Segmente und Ressourcen.

4.3.1 Implementierung

Interface Runnable Soll Instruktionsfolgen enthalten, die von Threads ausgeführt werden.

```
public interface Runnable {
  public abstract void run();
}
```

Klasse Thread Steuert die parallelisierung von Runnables.

```
publ. cl. Thread impl. Runnable {
  public Thread(String name);
  public Thread(Runnable target);
  public void start();
  public void run();
}
```

Fadenerzeugung `(new Thread(new Runnable() { @Override public void run() { ... } }).start();`

Kritischer Abschnitt ist ein Codesegment, in dem mehrere Faden gleichzeitig arbeiten (können).

Wettlaufsituation ist die Verfälschung von Daten durch einen kritischen Abschnitt. Kann verhindert werden, indem der kritische Abschnitt nur von einem Faden auf einmal betreten wird.

Monitor: Kann mit `monitor.enter()` betreten und mit `monitor.exit()` verlassen werden. Innerhalb des Monitors kann nur ein Faden arbeiten, andere müssen warten.

Synchronisierte Abschnitte: Mit `synchronized` gekennzeichnete Abschnitte können nur von einem Faden auf einmal betreten werden. Synchronisierte Abschnitte blockieren "für ein Objekt ", daher auch andere synchronisierte Abschnitte blockieren, falls sie auf demselben Objekt definiert sind. Wenn ein anderer synchronierter Abschnitt auf einem anderen Objekt definiert ist, können sie parallel laufen. Es sollte nach dem Objekt synchronisiert werden, das die kritischen Daten hält.

Objektmethoden zu Synchronisierung: Java bietet folgende Methoden für `javas.lang.Objekt`:

wait() Versetzt Faden in Wartezustand. Der Prozessor gibt den Faden ab.

wait(long timeout, [int nanos)] Versetzt den Faden in einen zeitlich begrenzten Wartezustand. In Klausuraufgaben fast immer falsch, dies zu tun.

notifyAll() Weckt alle wartenden Aktivitäten auf. Da danach immernoch mehr als eine Aktivität den kritischen Abschnitt

betreten wollen könnte, sollte nach dem Aufwecken erneut geprüft werden und unter Umständen der Faden zurück in den Wartezustand geschickt werden (Prüfe ob frei in `while` Schleife, falls nicht, `wait()`).

notify() Wecke irgendeine Aktivität auf. In Klausuraufgaben fast immer falsch, dies zu tun.

Achtung: `wait()` kann eine `InterruptedException` werfen, wenn die Aktivität unterbrochen wurde.

Verklemmung: Zwei Threads reservieren für sich zwei verschiedene Objekte gleichzeitig, und brauchen dann das jeweils andere Objekt synchronisiert. Beide Threads warten, bis der andere sein Objekt freigibt. Das Programm kommt dann nicht mehr weiter. "Deadlock ". Kann manchmal verhindert werden, indem Objekte "mit Reihenfolge "reserviert werden.

java.util.concurrent bietet weitere Mengentypen wie `ConcurrentHashMap` oder Ähnliche sowie atomare Operationen zur Verfügung, die im parallelen Fall sicher zu nutzen sind.

`java.util.concurrent` enthält auch `Semaphore`, die die beiden Methoden `acquire` (blockiert, bis Ticket verfügbar ist, und erniedrigt dann die Anzahl der Genehmigungen um 1/Parameterwert) und `release` (erhöht die Anzahl der Tickets um 1/Parameterwert).

volatile bewirkt bei Deklaration von Variablen, dass die Variable immer aktualisiert wird. Falls dies nicht der Fall ist, könnten mehrere Threads die Variable an verschiedenen Stellen cachen und sie könnte verfälscht werden.

Paralelle Matrix-Matrix Multiplikation:
Gewöhnlicher ijk-Algorithmus (Zeile i von A mal Spalte j von B ergibt Zelle i,j in C)...

$$c_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

...ist nicht Cache-freundlich, da er B spaltenweise durchläuft.

Stattdessen: ikj-Algorithmus (Zwischenspeichere Zelle aus A und laufe dann Zeile in B durch):

```
for (int i = 0; i < n; i++) {
  for (int k = 0; k < n; k++) {
    r = a[i][k];
    for (j = 0; j < n; j++) {
      c[i][j] += r * b[k][j];
    }
  }
}
```

Bewertung paralleler Algorithmen:
Dazu folgende Kriterien:

Beschleunigung gibt an, wieviel schneller der Algorithmus mit p Prozessoren verglichen zu sequentieller Ausführung ist.
$S(p) = \frac{T(1)}{T(p)}$

Effizienz gibt den Anteil der Ausführungszeit an, die jeder Prozessor mit nützlicher Arbeit verbringt.
$E(p) = \frac{T(1)}{p \cdot T(p)} = \frac{S(p)}{p}$

Idealerweise ist $S(p) = p$ und $E(p) = 1$.

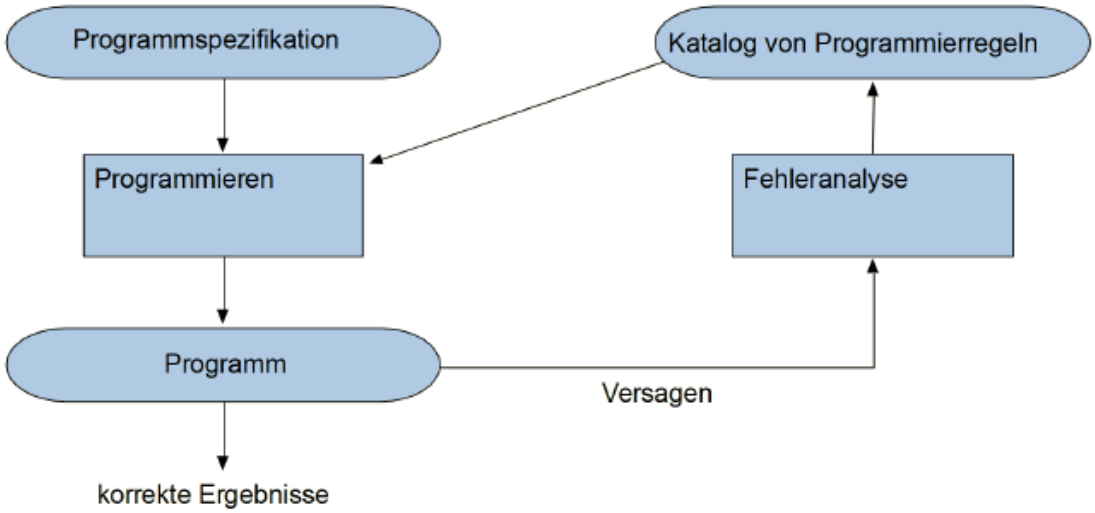
Weiter existieren:

- Zeit für sequentiellen Teil σ.

- Zeit für sequentielle Ausführung des parallelisierbaren Teils π.
- Gesamtlaufzeit $T(p) = \sigma + \frac{\pi}{p}$.

Amdahlsche Gesetz: $S(p) \leq \frac{1}{f}$ mit $f = \frac{\sigma}{\sigma+\pi}$ als sequentieller nicht parallelisierbarer Anteil.

4.4 Selbstkontrolliertes Programmieren



Fehlerlogbuch sollte vom Programmierer geführt werden und Fehler eingetragen werden ,deren Suche lange gedauert hat, deren verursachten Kosten hoch waren oder der lange unentdeckt geblieben ist. Relevante Daten:

- Laufende Fehlernummer
- Datum (wann entstanden, wann entdeckt)
- Aus welcher Phase entstanden
- Fehlerkurzbeschreibung (Titel)
- Ursache (Verhaltensmechanismus)

Zeitlogbuch sollte von Teammitarbeitern geführt werden, um Zeit für einzelne Tätigkeite zu erfassen. Das hilft Schätzungen in Zukunft genauer geben zu können.

5 Testen

5.1 Definitionen

"Testing shows the presence of bugs, not their abstance. "

Verfahren Man unterscheidet:

Testende Verfahren: Fehler erkennen.

Verifizierende Verfahren: Korrektheit

beweisen.

Analysierende Verfahren: Eigenschaften einer Systemkompoenente bestimmen.

Fehlerarten: Man unterscheidet:

Versagen (Ausfall) ist die Abweichung des Verhaltens von Software von der Spezifikation (also ein Ereignis). Verursacht durch Defekt.

Defekt (bug) ist ein Mangel im Softwareprodukt, der zu Versagen führen kann. Verursacht von Irrtum.

Irrtum (Herstellungsfehler) ist eine menschliche Aktion, die einen Defekt verursacht (also ein Vorgang).

Testhelfern: Man unterscheidet:

Stummel (stub) ist ein grob implementierter Softwareteil und dient als Platzhalter unfertiger Funktionalität.

Attrappe (dummy) simuliert Implementierung zu Testzwecken.

Nachahmung (mock objekt) ist eine Attrappe mit zusätzlicher Funktionalität, die für das Testen behilflich ist.

Fehlerklassen: Man unterscheidet:

Anforderungsfehler: Defekt im Pflichtenheft.

- Inkorrekte Angaben der Benutzterwünsche.
- Unvollständige Angaben über Anforderungen.
- Inkonsistenz von Anforderungen.
- Undurchführbarkeit.

Entwurfsfehler: Defekt in der Spezifikation.

- Unnvollständige/fehlerhafte Umsetzung der Anforderungen.
- Inkonsistenz in/zwischen Anforderungen, Spezifikation, Entwurf.

Implementierungsfehler: Defekt im Programm

- Fehlerhafte Umsetzung der Spezifikation.

Modul-/Softwaretestverfahren: Man unterscheidet:

Softwaretest führt Softwarekompenenten oder Konfiguration von Softwareelementen unter bekannten Bedinungen aus und prüft ihr Verhalten.

Testling (Prüfling, Testobjekt, component under test, CUT) ist die zu prüfende Softwarekompenente/Softwarekonfiguration.

Testtreiber (Testrahmen) versorgt Testlinge mit Testfällen und stößt deren Ausführung an.

Testphasen: Der Entwickler macht folgende Tests:

Kompenententest überprüft Funktionen einzelner Module, definiert durch Objektentwurfsdokument.

Integrationstest überprüft *schrittweise* das Zusammenwirken bereits einzeln getesteter Systemkomponenten, definiert durch das Systementwurfsdokument.

Systemtest ist vom Auftraggeber festgelegt und überprüft abschließend in realer Umgebung ohne Kunden. Unterteilt in funktionaler und nichtfunktionaler Systemtest. Definiert durch Anforderungsanalysedokument.

Der Kunde ist bei folgendem Test anwesend:

Abnahmetest ist der abschließende Test in realer Umgebung mit Beobachtung des Kunden. Mitwirkung und/oder Federführung des Kunden. Definiert durch Kundenerwartung.

Testverfahren: In den Testphasen wird verwendet:

Dynamische Verfahren zum *Testen*:

Strukturtests (white box testing)

- Kontrollflussorientierte Tests
- Datenflussorientierte Tests

Funktionale Tests (black box testing) Teste die erwartete Funktionalität.

Leistungstests (black box testing) *Lasttest* (teste zu unterstützende Grenzen) oder *Stresstest* (übertrete Grenzen und schaue was passiert).

Statische Verfahren zum *Prüfen*:

Manulle Prüfmethoden Inspektion, Review, Durchsichten.

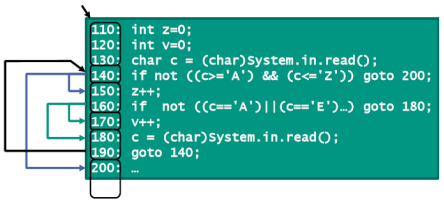
Prüfprograme zur statischen Analyse von Programmen.

Regressionstest ist die Wiederholung eines bereits vollständig durchgeführten *Systemtests*, z.B. aufgrund von Pflege, Änderungen oder Korrekturen.

5.2 Strukturtest (Kontrollflussgraph)

Strukturerhaltende Transformation von einer Quellsprache zur Zwischensprache:

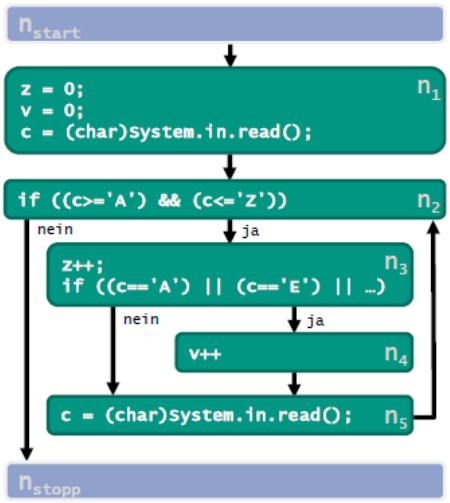
- Ausschließlich Befehle, die die *Ausführungsreihenfolge* beeinflussen, zur Zwischensprache transformiert werden.
- Alle anderen Befehle und deren Ausführungsreihenfolge bleibt gleich.
- Kein Ausrollen von Schleifen, Optimierungen, sonstiges.



Vorgehensweise:

1. Transformiere in Zwischensprache.
2. Fasse alle Folgen, die in einem Sprung enden, zu je einem Grundblock zusammen.
3. Sprungeintritt nur bei Blockanfang, falls nicht jeweilige Blöcke

aufteilen.



Zweig = Kante $e \in E$ im Kontrollflussgraphen.

Vollständiger Pfad ist ein Pfad im Kontrollflussgraph der bei n_{start} anfängt und bei n_{stopp} aufhört.

Teststrategien definieren Möglichkeiten, mit Kontrollflussgraphen Fehler zu finden.

Anweisungsüberdeckung $C_{Anweisung}$ (statement coverage) verlangt die Ausführung aller Grundblöcke des Programms.

$$C_{Anweisung} = \frac{\#\text{durchlaufener Anw.}}{\#\text{aller Anw.}}$$

Nicht ausreichendes Testkriterium, nicht ausgeführte Teile können gefunden werden, fehlende Teile aber nicht.

Zweigüberdeckung C_{Zweig} (branch coverage) verlangt Traversierung aller Zweige.

$$C_{Zweig} = \frac{\#\text{traversierter Zweige}}{\#\text{aller Zweige}}$$

Unausgeführte Zweige können gefunden werden, aber keine Pfade, noch komplexe Bedinungen, noch Schleifen gründlich. Fehlende Zweige werden nicht entdeckt.

Pfadüberdeckung verlangt die Ausführung aller möglichen Pfade. Nicht praktikabel wegen Schleifen.

Subsumierung erfolgt bei den genannten Verfahren in nach oben.

Boundary-Interior-Test .

- Konstruiere Testfälle, die Schleifenrumpf einmal durchqueren, dabei alle Verzweigungen durchqueren.
- Konstruiere Testfälle, die Schleifenrumpf min. zweimal durchqueren, beim zweiten mal alle Zweige durchqueren.

Subsumieren: Ein Testverfahren subsumiert ein anderes, wenn es alles überprüft, was auch das subsumierte Verfahren prüfen kann.

5.3 Funktionale Tests

Äquivalenzklassenbildung Suche Klassen von Eingabewerten. Partitioniere entlang Definitionsgrenzen. Teste dann nur mit jeweils einem Mitglied pro ÄK. Verwende möglichst Elemente auf und um den Rand der ÄK (um off-by-one Fehler zu verhindern).

Zufallstest Verwende zufällige Testfälle. Sorgt dafür, dass intuitiv der Im-

plementierung naheliegende Testfälle benutzt werden. Sinnvoll als Ergänzung.

Testen von Zustandsautomaten Durchlaufe jeden Übergang mindestens einmal. (Garantiert aber noch keine Fehlerfreiheit)

α-ω-Zyklus bezeichnet den kompletten Lebenszyklus eines Objekts, von Speicherallokation bis Speicherfreigabe.

5.4 Statische/manuelle Verfahren

Manuelle Prüfung prüft die Semantik, sehr teuer und aufwendig. Psychologischer Druck bei Arbeitsbegutachtung.

Überprüfung (review) Formalisierter Prozess zur Überprüfung schriftlicher Dokumente durch externen Gutachter.

Durchsicht (walkthrough) Der Entwickler führt Kollegen durch Teil seines Codes/Entwurfs. Kollegen machen Bemerkungen und Anmerkungen zu Stil, Defekten, Entwicklungsstandards, Probleme.

Software-Inspektion 2-8 Inspektoren untersuchen unabhängig das Dokument. Gefundene Defekte werden aufgeschrieben und gemeinsam durchgesprochen. Verwendung von Prüflisten und Formularen. Kann auch auf anderen Dokumenten als Software durchgeführt werden, sowie auch frühzeitig. Allerdings aufwendig, teuer, "statisch ". Funktionsweise:

1. Vorbereitung.
2. Individuelle Fehlersuche. Benutze vereinbarte Lesetechnik, dokumentiere Problempunkte. 1 Seite pro Stunde pro Inspektor.
3. Gruppensitzung. 2h. Bespreche Problempunkte, kläre Fragen, weitere Problempunkte und Verbesserungsvorschläge sammeln.
4. Nachbereitung. Editor erhält Liste von Problempunkten, identifiziert und klassifiziert tatsächliche Defekte. Problempunkte werden bearbeitet und überprüft. Es verbleiben etwa so viele Defekte wie gefunden wurden. $\frac{1}{6}$ der Korrekturen sind fehlerhaft oder verursachen neue Defekte. Dokument wird freigegeben wenn geschätzte Fehlernummer gering genug.

Rollen:

Inspektionsleiter leitet alle Phasen der Inspektion.

Moderator leitet Gruppensitzung, ist meistens Inspektionsleiter.

Inspektoren prüfen das Dokument.

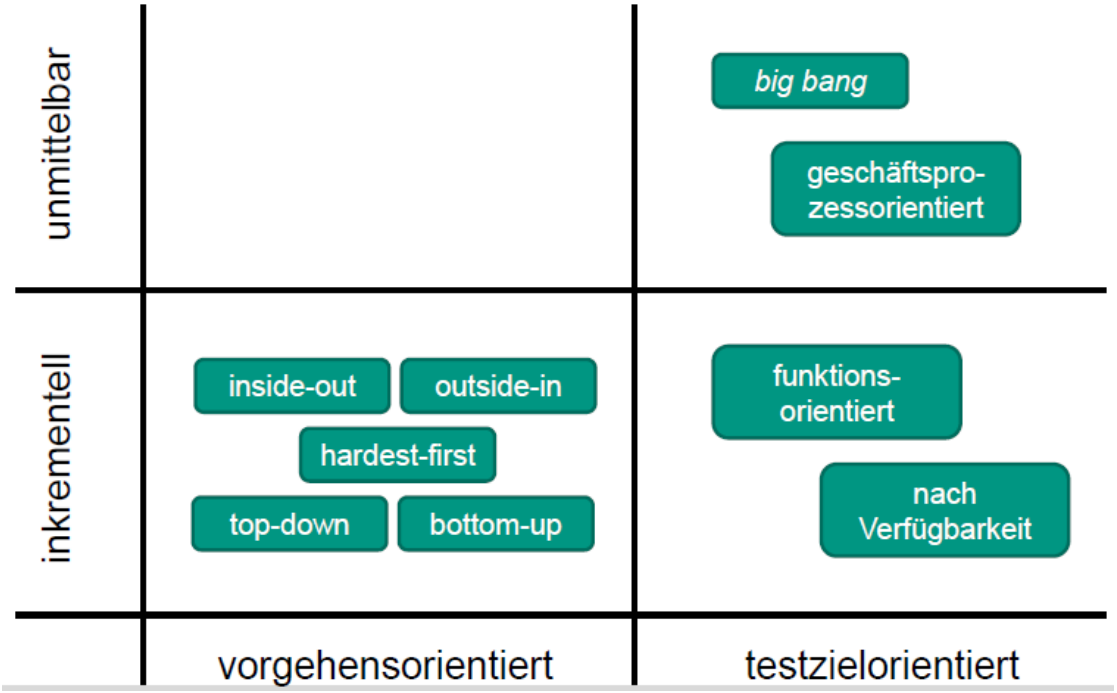
Schriftführer protokolliert Defekte in Gruppensitzung.

Editor klassifiziert und behebt Defekte, ist meistens Autor.

Autor hat das Dokument verfasst.

5.5 Integrationstests

Nachdem einzelne Kompenenten mit den zuvor angeführten Verfahren getestet wurden, folgen die Integrationstests, um die Software zusammenzuführen und das Zusammenwirken der Komponenten zu testen.



Integrationsstrategien Man unterscheidet:

Big bang: Alle Komponenten werden gleichzeitig integriert. Ist kaum systematisierbar, macht Konstruktion von Testfällen und Defektsuche schwierig. "Nichts geht bis alles geht ".

geschäftsprozessorientiert: Module werden entsprechender Anwendungsfälle integriert.

funktionsorientiert: Schrittweise Integration, sodass *funktionale Tests* nacheinander funktionieren.

nach Verfügbarkeit: Komponentenintegration direkt nach Implementierung. Reihenfolge durch Produktionsreihenfolge festgelegt.

top-down: Integration von höchster logischer Ebene (z.B. GUI).

bottom-up: Integration von niedrigster logischen Ebene. Erfordert Testtreiber, dafür leichtes Herstellen von Testbedinungen und leichte Interpretation von Testergebnissen.

inside-out: Beginne auf mittlerer Ebene mit Integration nach außen. Nicht sehr sinnvoll.

outside-in: Schrittweise Integration von höchster und niedrigster Ebene nach innen.

hardest-first: Integriere zuerst die

kritischen und kompliziertesten Kompenenten, dann immer einfachere. Kritische Kompenenten werden dabei jedes mal mitgeprüft und sind am Ende am besten geprüft worden.

6 Abnahme und danach

Abnahmephase Übergabe des Gesamtprodukts einschließlich der Dokumentation an Auftraggeber, Abnahmetest (dabei Stress- und Belastungstests). Ergebnis: Abnahmeprotokoll.

Abnahmetest Prüfe Erfüllung der Qualitätsmerkmale, falls Auftraggeber Wartung und Pflege selbst übernimmt, übergebe Entwurfs- und Implementierungsdokumentation, Testeinrichtung und führe ihn in Architektur ein.

Einführungsphase Installation und Einrichtung des Produkts, Schulung der Benutzer/Betriebspersonals, Inbetriebnahme. Falls älteres System vorhanden ist dieses Umzustellen (insbesondere Datenbestände übertragen), mit einer der folgenden drei Methoden:

- Direkte Umstellung
- Parallellauf
- Versuchslauf

Wartungs- und Pflegephase Software *altert*, wenn nicht ständig Defekte behoben werden und Anpassungen an Umwelt und an neue Anforderungen gemacht werden. Falls veraltet, kann sie nicht mehr für ursprünglichen Zweck verwendet werden. (Achtung: Sie altert, unterliegt aber keinem Verschleiß!)
"Software veraltet in dem Maße, wie sie in Wirklichkeit nicht Schritt hält."
Verschiedene Wartungstätigkeiten:

- Korrektive Tätigkeiten (Wartung)
  - Stabilisierung/ Korrektur.
  - Optimierung.

  Wartung ist Ereignisgesteuert und damit schwer planbar: Beinhaltet Lokalisierung und Beheben von Defekten während Betrieb.
- progressive Tätigkeiten (Pflege).
  - Anpassung/ Änderung.
  - Erweiterung.

  Pflege ist planbar: Beinhaltet geplante und definierte Erweiterung des Systems während Betrieb.

Software-Sanierung Alte Software ist weitgehend Teil der installierten Software. Altsoftware wird vorher oft umgewandelt in andere Sprache oder neuerer Entwurf, um Sanierung zu vereinfachen.

7 Kosten- und Aufwandsschätzung

Einflussfaktoren sind bei Kostenschätzung:

- Quantität (Umfang + Komplexität)
- Qualität (Qualitätsmerkmale)
- Entwicklungsdauer (Kommunikationsaufwand, Produktivität)
- Kosten

Diese beeinflussen sich im *Teufelsquadrat* gegenseitig.

Schätzungsmethoden werden zur Kostenschätzung verwendet:

Analogiemethode Schätze Projektaufwand mithilfe protokollierten Aufwand von bestehendem Projekt. Relativ einfach und intuitiv, aber nicht allgemein.

Relationsmethode Vergleiche Produkt direkt mit ähnlichen Entwicklungen und passe Aufwand mit Erfahrungswerten an. Dafür stehen Faktorenlisten und Richtlinien zur Verfügung.

Multiplikatormethode Zerlege Produkt in Teile, bis jedem Teil ein feststehender Aufwand zugeordnet werden kann (in LOC).

Phasenaufteilung Messe, wieviel Aufwand einzelne Phasen einer Entwicklung war, schließe *eine* Phase der neuen Entwicklung ab und vergleiche diese mit der ersten Phase der alten Entwicklung, dann schätze andere Phasen. (im Allgemeinen aber unbrauchbar)

CoCoMo II Verwende geschätzte Größe und 22 Einflussfaktoren, dann benutze CoCoMo II Modell.

Delphi-Schätzmehode Runde von Schätzern geben anonym Schätzung und Begründung ab, Moderator fasst zusammen.

Planungspoker Wie Delphi, aber mit Kartenstapel. Schätzungen werden gleichzeitig aufgedeckt, höchste und niedrigste Schätzer begründen.

8 Prozessmodelle

Programmieren durch Probieren (trial and error) lässt ein vorläufiges Programm entstehen, danach überlegt man sich Anforderung, Entwurf, Testen und Wartung und verbessert das Programm entsprechend. Fast reines Coding, aber schlecht strukturierter Code.

Wasserfallmodell (Phasenmodell).

1. Planung

   ⇓ Lastenheft, Projektplan, Kalkulation
2. Definition

   ⇓ Pflichtenheft, GUI-Beschreibung, evtl. Benutzerhandbuch
3. Entwurf

   ⇓ Entwurfsdokumente, Modulführer
4. Implementierung

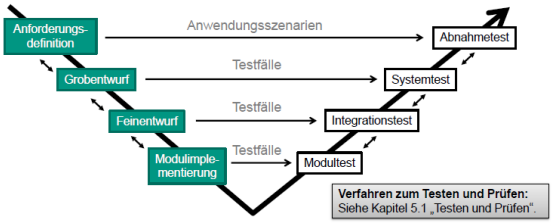
   ⇓ Komponenten, Dokumentation, Testeinrichtung

5. Testen

   ⇓ “fertiges “System

6. Einsatz und Wartung

V-Modell 97 .

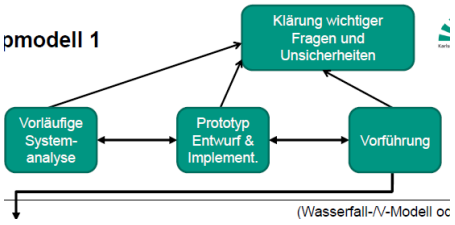


V-Modell XT Entwicklungsstandard für öffentliche IT-Systeme in Deutschland. Aktivitäten, Produkte und Verantwortliche werden festgelegt, Phasengrenzen oder Reihenfolge nicht. Projekt wird aus Perspektiven (definiert als 30 Personenrollen) betrachtet.

Im alten V-Modell 97 gab es 4 Submodelle, die mit XT in Vorgehensbausteine gegliedert werden. Submodelle:

- Projektmanagement
- Qualitätssicherung
- Konfigurationsmanagement
- Systemerstellung

Prototypmodell Prototyp (eingeschränktes funktionsfähiges System) stärkt Arbeitsmoral und Anbieter-Kunden-Vertrauen. Prototyp *muss* weggeworfen werden.



Danach geht man zu alternativen Prozessmodell über.

Iteratives Modell Funktionalität wird Schritt für Schritt erstellt und dem Produkt hinzugefügt. Wie Prototypmodell, aber mehr wird wiederverwendet.

Evolutionär: Plane und analysiere nur den Teil, der als nächstes dazugefügt wird.

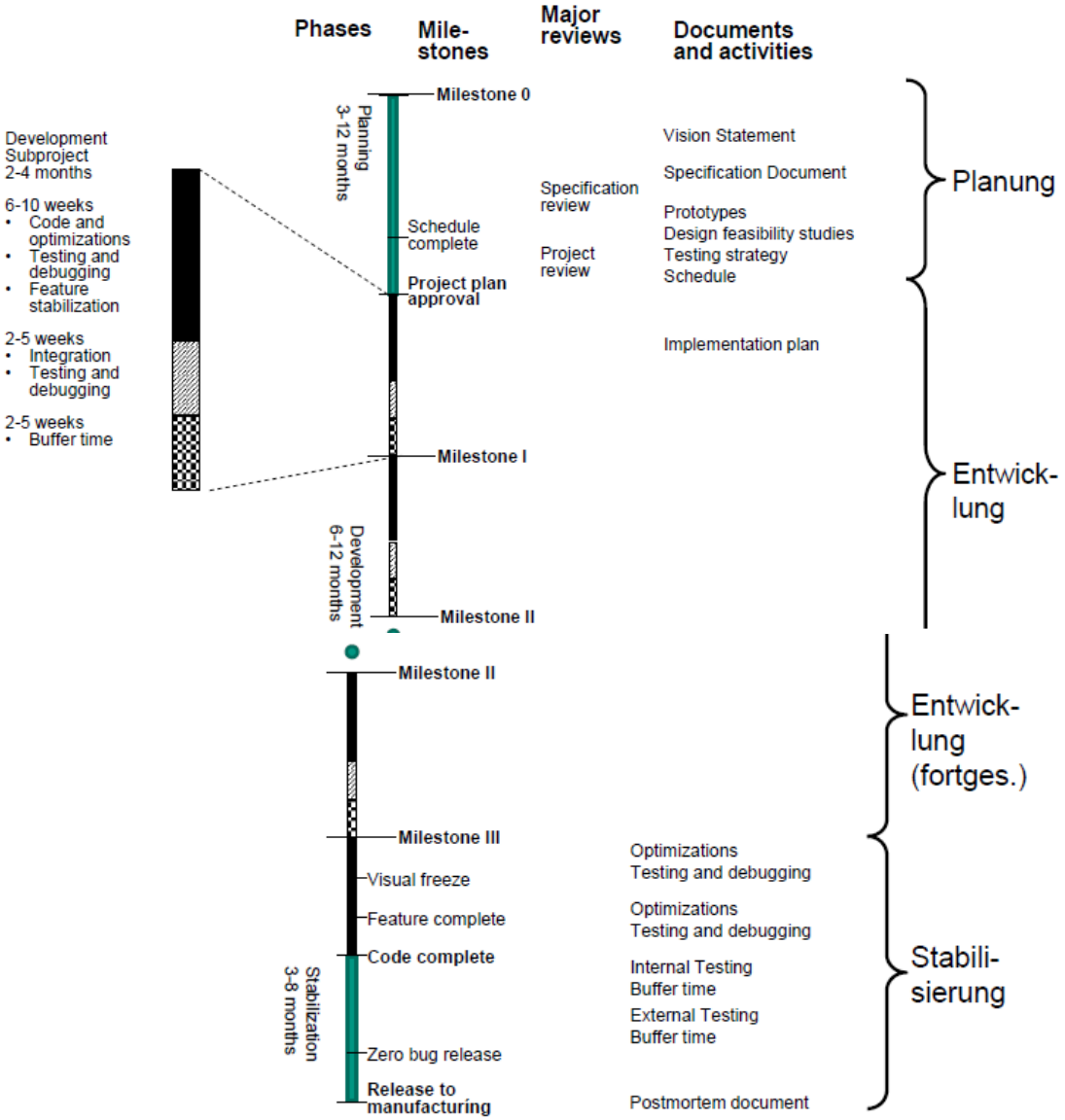
Inkrementell: Plane und analysiere alles, dann iteriere über Entwurf, Implementierung und Testen.

Synchronisiere und Stabilisiere (Microsoft Modell) Organisiere Programmierer in kleine Teams, die eigenständig arbeiten, regelmäßige Synchronisation (nächtlich, auch Neuübersetzen aller Quellen) und Stabilisierung (3 Mon. Meilensteine). Drei Phasen:

- Planungsphase
- Entwicklungsphase in 3 Subprojekten
- Stabilisierungsphase

Erstes Subprojekt/Meilenstein implementiert wichtigsten Funktionen, letztes Subprojekt die unwichtigsten Funktionen. Automatische Regressionstests, Sanktionen für Fehler (da sie andere Entwickler behindern).

Alle 18 Monate sind 50% des Codes überarbeitet worden.


Phases
Mile-stones
Major reviews
Documents and activities
Development Subproject 2-4 months
6-10 weeks
• Code and optimizations
• Testing and debugging
• Feature stabilization
2-5 weeks
• Integration
• Testing and debugging
2-5 weeks
• Buffer time
Planning 3-12 months
Milestone 0
Schedule complete
Project plan approval
Milestone I
Development 6-12 months
Milestone II
Milestone II
Milestone III
Visual freeze
Feature complete
Code complete
Stabilization 3-8 months
Zero bug release
Release to manufacturing
Specification review
Project review
Vision Statement
Specification Document
Prototypes
Design feasibility studies
Testing strategy
Schedule
Implementation plan
Optimizations
Testing and debugging
Optimizations
Testing and debugging
Internal Testing
Buffer time
External Testing
Buffer time
Postmortem document
Planung
Entwick-lung
Entwick-lung (fortges.)
Stabili-sierung

8.1 Agile Prozesse

Agiler Prozess hat minimale Vorausplanung, inkrementelle Planung und vermeidet unterstützende Dokumente. Bezieht Kunde stark in Entwicklung ein. Z.B. Scrum, Extreme Programming.

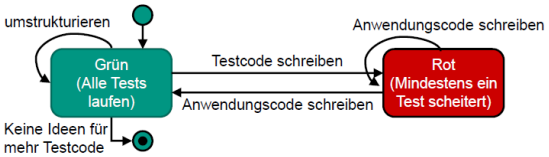
Paarprogrammierung führt zu höherer Qualität und ersetzt Inspektionen und Reviews. Viel teurer.

Extreme Programmierung Geeignet für vage und schnell ändernde Anforderungen, für kleine Entwicklerteams.

"Softwareentwicklung mit leichtem Gepäck."

Testen: Programmierer schreiben automatische Komponententests, die bei jeder Codeintegration zu 100% laufen müssen, Kunde spezifiziert Akzeptanztests, die spätestens zur Auslieferung laufen müssen.

Testgetriebene Entwicklung Zustandsdiagramm:



Aktionen für Testergebnisumschwung:

grün → **rot** Schreibe fehlschlagenden Test, minimal viel Code (Stummel, Attrape..)

rot → **grün** Schreibe minimal viel Code, damit alle Tests erfolgreich laufen.

grün → **grün** Refaktorisierung.

Auslieferungsplanung ersetzt konventionelle Anforderungsanalyse. Projektplan wird vom Kunden und von Entwicklern gemeinsam erstellt.

Iterationsplanung findet mehrmals zwischen Planung und Auslierferung statt. Teammitglieder übernehmen geplante Aufgaben und schätzen deren Aufwand.

Kunde vor Ort Echter Kunde, der das Produkt später verwendet, ist vor Ort und ständig Verfügbar.

Scrum Agiles Projektmanagement. Rollen:

Auftraggeber (product owner) legt Anforderungen und Auslieferungstermin fest, stellt Budget.

Scrum Master Stellt Einhaltungen der Scrum-Werte und -Techniken sicher. Unterstützt Zusammenarbeit und schützt vor Störungen.

Entwicklungsteam Selbstorganisiertes Team.

Artefakte:

- Anforderungsliste (product backlog)
- Aufgabenliste (sprint backlog)
- Hindernisliste (impediment backlog)

Treffensarten:

Sprintplanung Das Team wählt die Anforderungen, die in dem Sprint erledigt werden können und erstellt mit Scrum-Master die Aufgabenliste.

Daily Scrum Tägliches Meeting, alle sagen was sie am Vortag erle-

digt haben und heute erledigen wollen.

Review-Treffen am Ende des Sprints, das Team stellt die Sprintergebnisse vor als Demonstration.

Retrospektive Der zurückgelegte Sprint wird analysiert.