SWT Skript SoSe 2018

Guido

von Spokey 4buczko

Der Werkzeug & Material-Ansatz (WAM)

Anwendungsorientierung von Software

- Anwendungsorientierung ist auch unserer Sicht der Schlüssel zu einer qualitativ hochwertigen Softwareentwicklung, denn Mitarbeiter eines Unternehmens sollen ihre Aufgaben mithilfe der Software einfach und angemessen erledigen können.
    - Frage: Wie muss Anwendungssoftware gestaltet werden, um Menschen bei ihren Aufgaben am Arbeitsplatz optmial zu unterstützen?

Anwendungsorientierung und der WAM-Ansatz

- Der WAM-Ansatz ist vor allem auf Software ausgericht, die von Anwendern bei ihrer täglichen Arbeit eingesetzt wird
- Software soll den Anwendern helfen ihre verschiedenen Aufgaben zu erlediegen und sie flexibel bei den komplexen Geschäftsprozessen unterstützen, in die sie eingebunden sind.
- Aufgaben und Prozesse verändern sich häufig und gute Software muss entsprechend anpassungsfähig sein

Entwurf nach WAM: Strukturähnlichkeit

- Software soll strukturähnlich zu Modellen sein
- Fachliche Begriffe und Gegenstände sowie die relevanten Geschäftsprozesse bestimmen den Entwurf der Software.
- Das anwendungsfachliche Begriffsgebäude soll sich in den Elementen der Anwendungssoftware und in der Struktur der Softwarearchitektur widerspiegeln.
- Dadurch finden Anwender die Gegenstände ihrer Arbeit und die Begriffe ihrer Fachsprache im Anwendungssystem wieder. Sie können ihre Arbeit anhand ihrer Erfahrung organisieren.
- Die Entwickler werden beim Entwurf und der Weiterentwicklung der Software unterstützt. Sie werden angeleitet, Architekturen fachlich zu strukturieren und die Softwareelemente bei fachlichen und softwaretechnischen Änderungen mit Anwendungskonzepten in Beziehung zu setzen.

Gestaltung von Anwendungssoftware

- Layout der Oberfläche (Präsentation)
- Möglichkeiten der Interaktion (Handhabung, Look & Feel)
- vor allem: Benutzungsmodell.

→ das Benutzungsmodell das in einer Software drinsteckt, muss vom Entwickler reingeschrieben werden.
→ Man stellt sich den dümmsten menschen den man kennt als seinen Benutzer vor → Idiotensicheres Modell

Der Begriff Leitbild

→ Wie gucken wir auf einen Anwendungsbereich rauf wenn wir Software für jemanden entwickeln?
→ dessen Aufgabe ist es, den Beteiligten zu helfen das Produkt und den Prozess zu verstehen

Leitbild, allgemein

- eine benennbare, grundsätzliche Sichtweise, anhand derer wir einen Ausschnitt von Realität wahrnehmen, verstehen und gestalten.
- repräsentiert immer auch eine Wertvorstellung.

Leitbild in der Softwareentwicklung

- gibt im Entwicklungsprozess und für den Einsatz einen gemeinsamen Orientierungsrahmen für die beteiligten Gruppen.
- unterstützt den Entwurf, die Verwendung und die Bewertung von Software und basiert auf Wertvorstellungen und Zielsetzungen.
- kann konstruktiv oder analytisch verwendet werden.

Leitbild	Gestaltungsziel	Rolle der Anwender	Rolle der Entwickler
Objektwelten	Objektorientierung unmittelbar auf den Entwurf übertragen	Impulsgeber, der die Objekte aktiviert	Schöpfer von Miniwelten
Direkte Manipulation von Arbeitsgegenständen	bekannte Arbeitsgegenstände selbstverständlich manipulieren	Akteur, Bearbeiter	Konstrukteur von Artefakten
Fabrik	menschliche Arbeit automatisieren und kontrollieren	Maschinenbediener, Störfaktor	Maschinenbauer, Maschineneinrichter
Arbeitsplatz für eigenverantwortliche Expertentätigkeit	qualifizierte Arbeit durch geeigneten Arbeitsplatz unterstützen	eigenverantwortlicher Experte, der Fachsprache spricht	Werkzeugbauer, Arbeitspatzgestalter

Benutzungsmodell

- ein fachlich orientiertes Modell darüber, wie Anwendungssoftware bei der Erledigung der anstehenden Aufgaben im jeweiligen Einsatzkontext benutzt werden kann.
- umfasst eine Vorstellung von der Handhabung und Präsentation der Software aber auch von den fachlichen Gegenständen, Konzepten und Abläufen, die von der Software unterstützt werden.
- wird auf der Grundlage eines Leitbilds mit Entwurfsmetaphern realisiert.

Eine Entwurfsmetapher

- eine bildhafte, gegenständliche Vorstellung, die ein Leitbild fachlich und konstruktiv “ausgestaltet", d.h. konkretisiert.

- strukturiert die Wahrnehmung und trägt zur Begriffsbildung bei. Sie leitet die Vorstellung und Kommunikation über das, was fachlich analysiert, modelliert und technisch realisiert werden soll.
- dient der Gestaltung von Softwaresystemen, indem sie Handhabung und Funktionalität für die Beteiligten verständlicher macht.
- hat im WAM-Ansatz immer auch eine technisch konstruktive Interpretation in Form von Konstruktionsanleitungen und Entwurfsmustern.

Forderung: *Wähle Leitbild und Entwurfsmetaphern so, dass sie bruchlos zueinander passen und Analyse, Entwurf und Verwendung durchgängig unterstützen.*

Grundlage der WAM-Leitbilder: die unterstützende Sichtweise

Merkmale der unterstützenden Sichtweise:

- Experten erledigen häufig wechselnde Aufgaben von hoher Komplexität.
- Die Arbeit erfordert einen hohen Grad an Qualifikation, Kenntnissen und Erfahrung, der nicht formalisiert werden kann.
- Die Erledigung von Aufgaben orientiert sich zielgerichtet an vorhandenen Arbeitsmitteln und -gegenständen und nicht an vorgegebenen Routinen.
- Die situative Auswahl von Arbeitsschritten wird durch die vorhandenen Mittel unterstützt und führt zu jeweils adäquaten Ergebnissen.
- Pläne werden zur Orientierung und nicht als ausführbare Handlungsvorschrift betrachtet.
- Die Kontrolle über die Handlung verbleibt beim Menschen.

→ *Unterstützung und Automatisierung von Handlungen sind wesentliche Entwurfskriterien für Arbeitsplatzsysteme. Welche Handlungen verbleiben beim Benutzer, welche werden auf die Software übertragen?*

Leitbilder und Entwurfsmethapern

WAM-Leitbild: der Arbeitsplatz für eigenverantwortliche Tätigkeit, Platz mit ganz vielen Elementen drin (Kalender, Papierkorb, Dateien etc)

1. Merkmale:
   - Expertentätigkeit: Anwender sind Fachleute in ihrem Gebiet und wissen, wie die jeweiligen Aufgaben sinnvoll zu erledigen sind.
   - kooperative Tätigkeit: Aufgaben werden meist arbeitsteilig erledigt. Dabei sind die Kooperationsformen und -mittel allgemein bekannt.
   - komplexe, oft wechselnde Tätigkeit: Arbeitssituationen und Arbeitsinhalte wechseln oft und wenig vorhersehbar und erfordern situatives Handeln.
2. Kontexte:
   - Kundenorientierung: Das Leitbild passt zur Kundenorientierung als Unternehmensstrategie.
   - Domäne: Der Finanz- und Dienstleistungsbereich ist ein passendes Anwendungsgebiet des Leitbilds

Entwurfsmetaphern: Material, Werkzeug, Automat, Arbeitsumgebung

WAM: Menschen bearbeiten Material mit Werkzeugen (Metaphern)

Arbeiten bedeutet oft, dass Menschen einen Arbeitsgegenstand mit geeigneten Werkzeugen bearbeiten. Dies gilt nicht nur im Handwerk.

Arbeitsumgebung/Arbeitsplatz: Die Arbeitsumgebung ist der Ort, wo Werkzeuge, Materialien und anderen Gegenständen, die bei der Erledigung von Aufgaben griffbereit sein müssen, fachlich motiviert angeordnet sind. Dabei findet die eigentliche Arbeit am Arbeitsplatz statt, während zur Umgebung noch die Orte gehören, die unmittelbar zugänglich sind. Der (individuelle) Arbeitsplatz ist gegen den Zugriff von außen geschützt. Wenn nur die Arbeit eines einzelnen Benutzers unterstützt werden soll, fallen Arbeitsplatz und -umgebung meist zusammen.

Materialien sind also Gegenstände, die im Rahmen einer Aufgabe Teil des Arbeitsergebnisses werden. Sie werden durch Werkzeuge und Automaten bearbeitet und verkörpert fachliche Konzepte. Materialien müssen für die Bearbeitung geeignet sein.Viele Eigenschaften vorhandener Arbeitsgegenstände lassen sich sinnvoll auf Softwarematerialien übertragen.

→ *Was Werkzeug und was Material ist, erkennen wir erst in einer Arbeitssituation; dort allerdings eindeutig.*

Werkzeuge sind Gegenstände, mit denen Menschen im Rahmen einer Aufgabe Materialien verändern oder sondieren. Sie eignen sich meist für verschiedene Zwecke und unterschiedliche Materialien. Sie müssen geeignet gehandhabt werden. Werkzeuge vergegenständlichen wiederkehrende Arbeitshandlungen. Viele konzeptionelle Eigenschaften von (Hand-) Werkzeugen lassen sich auf Softwarewerkzeuge übertragen. **Eine direkte Abbildung von (manuellen) Werkzeugen in Software ist selten sinnvoll.**

→ *Materialien sondieren: „Kann ich mich gefahrlos auf diese Material (Tisch) setzen? „Ja“*

Automaten sind im Rahmen einer zur erledigenden Aufgabe ein Arbeitsmittel, um Material zu bearbeiten. Sie erledigen lästige Routinetätigkeiten als eine definierte Folge von Arbeitsschritten mit festem Ergebnis ohne weitere äußere Eingriffe. Sie laufen, wenn sie einmal vom Benutzer oder von der Arbeitsumgebung gestartet sind, unauffällig im Hintergrund. Und können auf ihren Zustand überprüft und im vorgegebenen Rahmen eingestellt werden. Automaten, die so funktionieren, nennen wir auch kleine Automaten. Sie unterscheiden sich damit von großen Automaten, die in Fabriken und Steuerungsanlagen “den Takt angeben.”

→ *Beispiele: Fahrkartenautomat, Waschmaschine, Ampelknopf*

Kleine Automaten können vom Anwender nach eigenem Ermessen innerhalb einer Arbeitsumgebung aus Werkzeugen, Materialien und Behältern eingesetzt werden und lassen sie sich beliebig wie Werkzeuge in größeren Arbeitszusammenhängen einsetzen. Sie decken den Standardfall ab und werden mit einigen wenigen Parametern eingestellt und laufen dann ohne weitere äußere Eingriffe ab.

→ *Entwurfsfrage: Ein Automat steuert keine Werkzeuge, aber wollen wir (Einstell-) Werkzeuge für kleine Automaten verwenden?*

Handhabung über mentale Indirektion

- Gegenstand im Benutzungsmodell wahrnehmen
- Veränderung antizipieren
- Optionen bewusst auswählen
- Operation aktivieren
- Veränderung am Gegenstand wahrnehmen

Werkzeug & Material: Grundsätzliches für den Entwurf

Zentrale Fragen:

- In welchem Verhältnis stehen die Gegenstände und Konzepte des bisherigen Anwendungsgebiets zu den Komponenten (Werkzeuge & Materialien) des Anwendungssystems?
- Was ist übertragbar, was nicht?

Ein naheliegender Entwurfsansatz ist, dass

- der Anwender bei der aktiven Erledigung von Aufgaben seine Materialien immer nur mit Hilfe eines Werkzeugs betrachten und verändern kann.
- es nicht möglich ist, Material direkt in die Hand nehmen oder diesen Eindruck zu vermitteln, sondern wir müssen für jede Handhabung ein geeignetes Werkzeug haben.

Merkmale von Software-Werkzeugen

- haben einen Namen, ggf. ein graphisches Symbol,
- zeigen eine Materialsicht, sind aber auch selbst im Blick,
- bieten mögliche Handhabungen, d.h. Operationen an, um das Material unterschiedlich zu bearbeiten,
- zeigen jederzeit ihre Einstellung, die die Materialsicht bestimmt,
- sollen keine Arbeitsabläufe implementieren.

Forderungen & Richtlinien: Materialen

- Die Erledigung von Aufgaben sollte mit Hilfe des Anwendungssystem so unterstützt werden, dass eine Ähnlichkeit zur bisherigen Arbeit erkennbar ist.
- Diese Ähnlichkeit schlägt sich zunächst im fachlichen Kern der Materialien nieder.
- Umgangsformen für Materialien ergeben sich aus der Art und Weise, wie die Gegenstände bei der Erledigung der verschiedenen Arbeitsaufgaben verändert und sondiert werden. Damit ist der fachliche Kern der Materialien definiert.
- Wichtige Materialien im Anwendungsbereich bilden damit den Ausgangpunkt für die Modellierung von (Software-) Materialien.

Eine besondere Art von Materialien: Behälter

- Zu einem Behälter gehört ein Inhaltsverzeichnis ebenso wie Konsistenzprüfungen und Operationen, die sich auf die Sammlung als Ganzes beziehen.
- Behälter verkörpern meist Ordnungen.
  - Nur wenige konventionelle Behälter sind als Wühlkisten gedacht, auch wenn sie es im Gebrauch gelegentlich werden.
  - Wir konstruieren Ordnungen explizit bei den fachlichen Behältern. Ein Behälter kennt das Ordnungsprinzip, das er verkörpert, und sorgt für seine Wahrung.
- Materialien können mit ihren Behältern an verschiedene Orte transportiert werden. Damit bieten Behälter den Ansatz zur Kooperation und Koordination.
- *Beispiel für einen spezialisierten fachlichen Behälter: Mappe mit Dokumenten*

EXKURS Siehe Folien 01, Seite 27

Kooperation und Koordination im WAM-Ansatz

WAM-Classic

Im Mittelpunkt; der individuelle Arbeitsplatz für qualifizierte, eigenverantwortliche Tätigkeit

Merkmale:

- Aufgabenorientierung
- Anwender- und Benutzerintegration
- Vergegenständlichung der relevanten fachlichen Konzepte als Systemkomponenten

Unterstützung von Zusammenarbeit

Jetzt im Mittelpunkt: vernetzte Arbeitsplätze zur eigenverantwortlichen Erledigung von arbeitsteiligen Aufgaben.

Merkmale:

- Verknüpfung von fachlicher Funktionalität und Kooperation in den Aufgaben
- Unterscheidung von expliziter und impliziter Kooperation und deren Unterstützung
- Vergegenständlichung von Kooperationsmitteln und -medien und der Kooperation selbst

→ Wir fragen nicht, wie Menschen kooperieren, sondern konzentrieren uns auf die Aufgaben und die Merkmale der Arbeitssituationen. Wir analysieren die Art der Arbeit und wer was mit wem und vor allem weshalb macht.

Kooperative Arbeit

Bei kooperativer Arbeit arbeiten verschiedene Personen geplant und koordiniert zusammen, um ein gemeinsames Ergebnis zu erreichen.

- Die Beteiligten wollen ein gemeinsames Produkt herstellen oder eine gemeinsame Dienstleistung erbringen
- Sie müssen sich begrenzte Ressourcen teilen. Im einfachsten Fall müssen sie den Arbeitsgegenstand austauschen
- Sie müssen ihre Arbeitshandlungen soweit notwendig aufeinander abstimmen, damit zeitliche oder sachlogische Reihenfolgen eingehalten werden
- Sie müssen sich in irgendeiner Weise darüber, verständigen was von wem wie getan wird

Koordination

Koordination ist der Prozeß oder der Mechanismus zur Abstimmung von Arbeitsteilung bei kooperativer Arbeit. Koordination kann auf wechselseitigen Konventionen oder ausdrücklichen Regeln beruhen.

- *Implizite Kooperation*: Der konkurrierende Zugriff mehrerer Benutzer auf gemeinsame Materialien aus verschiedenen Arbeitsplätzen wird deutlich.
- *Explizite Kooperation:* Gemeinsam benutzbare Posträume oder Registraturen werden genutzt, Vorgange werden unterstützt, Formen komplexer Kooperation sind relevant.
- *Explizite Koordination:* Zusätzlich zur Kooperation wird im Kooperationsmodell deutlich, dass und wie sich mehrere benutzer über ihre Art der Arbeitsteilung verständigen.

Kooperationsmodell

- Ein Kooperationsmodell ist ein Modell, das entweder explizit oder implizit die
- Zusammenarbeit (Kooperation) beschreibt und regelt.
- Wir betrachten hier Kooperationsmodelle, die Bestandteil eines Arbeitsplatzsystems sind.

Kooperationsmittel

- Ein Kooperationsmittel ist ein fachlich motivierter Gegenstand, der die Kooperation unterstützt. Er vergegenständlicht die Kooperation oder die dabei notwendige Koordination.
- Beispiele: sind Vorgangsmappen und Laufzettel. Eine Vorgangsmappe ermöglicht die Weitergabe von Unterlagen in einem arbeitsteiligen Prozeß. Laufzettel vergegenständlichen die Reihenfolge von kooperativen Arbeitsschritten und die jeweils Verantwortlichen.

Kooperationsmedium

- Ein Kooperationsmedium ist ein fachlich motivierter Gegenstand, der zur Realisierung von Kooperation in Anwendungssystemen dient.
- Gemeinsam ist allen Kooperationsmedien, daß sie den Austausch von Materialien oder von Information ermöglichen und selbst vergegenständlicht sind. Beispiele für Kooperationsmedien sind ein elektronisches Postversandsystem, Gruppenpostfächer oder ein elektronisches Notizbrett

Kooperationsmodelle: Stufen der Integration

- Kein explizites Kooperationsmodell: der Regelfall.
- Implizite Kooperation: der konkurrierende Zugriff aus verschiedenen
- Arbeitsplätzen wird deutlich.
- Explizite Kooperation: Verwendung von gemeinsam benutzbaren Posträumen oder Archiven, Vorgangsunterstützung und Formen komplexerKooperation.
- Jedes Kooperationsmodell wird durch geeignete, gegenständliche Kooperationsmittel ermöglicht

Implizite Kooperation

Bei der Impliziten Kooperation wird der konkurrierende Zugriff mehrerer Benutzer auf gemeinsame Ressourcen im Benutzungsmodell ermöglicht und verdeutlicht. Kooperation oder Koordination selbst sind aber nicht vergegenständlicht.
Merkmale:

- Arbeitsteiliger Zugriff auf gemeinsames Arbeitsmaterial.
- Mehrere Arbeitsplätze innerhalb einer gemeinsamen Arbeitsumgebung.
- Die anderen Arbeitsplätze sind nicht sichtbar.
- Die Beteiligten koordinieren sich durch Konventionen.

→ *zu dieser Kooperationsform passt das Konzept des Archivs. Es ist von mehr als einem Arbeitsplatz zugänglich, damit haben wir ein fachliches Modell (Materialaustausch) für ein technisches Konzept (gemeinsamer Datenzugriff). Durch Vermerke in der Bestandsliste sind Konkurrenzsituation und Entleiher deutlich und eine einfache Koordination (außerhalb des Systems) ist ermöglicht.*

Problem bei der impliziten Kooperation: Transparenz

Transparenz bei der Kooperation bedeutet, dass der Benutzer die Konkurrenzsituation mit Hilfe des Anwendungssystems erkennen kann:

- Das System zeigt, dass mehr als ein Benutzer am selben Material arbeitet.
- Die Einheit von Raum und Zeit bei der Materialbearbeitung bleibt gewahrt.
- Die Koordination erfolgt durch Konvention.

Die drei Einheiten des Dramas

Die drei Aristotelischen Einheiten sind Prinzipien zur Konstruktion von Dramen, die nach dem griechischen Philosophen Aristoteles benannt sind, weil sie sich auf Aussagen in seiner Poetik stützen.

Gemäß der Forderung nach Einhaltung der drei Einheiten sollten **Zeit, Raum** und **Handlung** eines Dramas einheitlich bleiben. Das bedeutet, dass Zeitsprünge, Ortsveränderungen und Nebenhandlungen ausgeschlossen sind.

Einheit von Raum und Zeit nach WAM bei der Materialbearbeitung

- Ein Material kann zu einem Zeitpunkt nur an genau einem Ort sein und dort bearbeitet werden.
- Hat ein Benutzer ein Material auf seinem persönlichen Arbeitsplatz, so kann es nicht von anderen bearbeitet werden. In der Regel ist es dann für andere auch nicht sichtbar.

Umgang mit Materialien: Mechanismen für konkurrierenden Zugriff

Exklusiver Zugriff auf das Material:

- es gibt keine Kopien. Das entnommene Material ist für andere Benutzer gesperrt.
- Grund für die Modellierung: Ein Material soll nur einmal vorhanden sein.
- Konsequenz: Die Benutzer müssen darüber informiert werden, wo sich das Material momentan befindet, da es sonst zu Unterbrechungen und Störungen im Arbeitsprozess kommen kann

Exklusiver Zugriff auf ein Original, **Kopien sind möglich.**

- Nur ein Benutzer kann das Original des Material aus dem Archiv nehmen. Weitere Benutzer erkennen, dass das Original ausgeliehen ist, können sich aber Arbeitskopien ziehen.
- Grund für die Modellierung: Auch wenn das Original eines Materials nicht mehr verfügbar ist, sollen Arbeitskopien zur Verfügung stehen.
- Konsequenz: Die Koordination von Änderungen liegt bei den Benutzern. Änderungen am Original ziehen keine automatischen Änderungen an den Kopien nach sich. Benutzer von Kopien können über die Rückgabe des Originals informiert werden.

Zugriff immer nur auf Kopien möglich.

- Jeder Benutzer erhält nur eine Arbeitskopie mit internem Zeitstempel. Das Original kann durch eine Arbeitskopie ersetzt werden. Dabei erhält es einen aktuellen

Zeitstempel. Wenn ein Original durch Arbeitskopie mit einem älteren Zeitstempel ersetzt werden soll, wird der Benutzer über den Konflikt und seine Ursache informiert.

- Grund für die Modellierung: Bei meist lesendem Zugriff ist es uninteressant, dass viele Benutzer konkurrierend zugreifen. Oft kann das Original nicht dauerhaft für einen Benutzer exklusiv verfügbar gemacht werden (z.B. ein Konto).
- Konsequenz: Der Benutzer wird über Bearbeitungskonflikte informiert und kann sich anhand der mitgelieferten Informationen mit anderen Benutzern über die Regelung des Konflikts verständigen.

Original und Kopie

- Materialien werden häufig als Originale und Kopien zur Verfügung gestellt. Kopien, besser Arbeitskopien, sind Materialien , die wie Fotokopien weiter bearbeitet werden können. Zwischen Kopie und Original besteht keine technische Verbindung, so dass Änderungen an dem einen, nicht auf das andere rückwirken.
- Von Arbeitskopien sind technische Kopien zu unterscheiden. Diese können bei der Realisierung des Systems aus softwaretechnischen Gründen auch von fachlichen Originalen erzeugt werden, etwa um Zugriffszeiten zu minimieren.

Registratur

- Eine gemeinsame Registratur ist ein Kooperationsmittel, das von mehr als einem Arbeitsplatz aus zugänglich ist.
- Der Zweck einer Registratur ist die Bereitstellung und konsistente Verwaltung von Materialien.
- Eine Registratur führt eine Bestandsliste, aus der die verwalteten Materialien ersichtlich sind. Jeder Zugriff auf die Registratur wird in der Bestandsliste durch Vermerke über die entnommenen Materialien dokumentiert.

Explizite Kooperation

Bei der expliziten Kooperation wird im Benutzungsmodell deutlich, dass mehrere Benutzer kooperativ in einer gemeinsamen Arbeitsumgebung arbeiten. Geeignete Kooperationsmittel und -medien stehen für die Weitergabe von Materialien und die Koordination bereit.

Explizite Kooperation durch Materialaustausch

Merkmale:

- Zur Erledigung einer kooperativen Aufgabe werden arbeitsteilig gemeinsame Arbeitsmaterialien explizit ausgetauscht und weitergegeben.
- Die verschiedenen Arbeitsplätze sind durch Kooperationsmedien verbunden.
- Am einzelnen Arbeitsplatz ist sichtbar, welche anderen Arbeitsplätze vorhanden sind.
- Die Beteiligten kennen die Art und Weise ihrer Zusammenarbeit und benötigen keine besonderen Mechanismen innerhalb des Anwendungssystems, um ihre Arbeit zu koordinieren.

→ zu dieser Kooperationsform passt das Konzept von Postfächern zum Materialaustausch. Von jedem Arbeitsplatz ist der gemeinsame Postraum mit seinen Fächern sichtbar. Jeder Benutzer kann Material in jedes beliebige Postfach hineinlegen oder aus einem Fach herausnehmen. Es gibt Konventionen, wer was in welche Fächer legt oder aus ihnen herausholen darf.

Anwendungsbeispiel für kooperative Tätigkeiten aus dem Projekt-Kontext:

Kreditvergabe

Beteiligte Rollen: Kundenberater, Sekretariat, Schalterangestellte

Nr.	Tätigkeit	Frühestens	Spätestens	Muß/Kann
1	Kundenzuordnung	–	vor 2	M
2	Gesprächsvorbereitung	nach 1	vor 3	M
3	Beratungsgespräch	nach 2	vor 4	M
4	Berechnung des Angebots	während 3	während 5	M
5	Gesprächsnachbereitung	nach 4	vor 6	K
6	Protokollerstellung	nach 4	vor 12	K
7	Mündliche Zusage	während 4	vor 13	K
8	Schriftliche Genehmigung	nach 7	vor 12	M
9	Vertragserstellung	während 4	nach 8	M
10	Vertragsunterschrift	während 4	vor 12	M
11	Konten-/Aktenanlage	nach 4	nach 10	M
12	Kreditkontrolle	nach 10	vor 13	K
13	Auszahlung	während 4	nach 12	M

Prozessmuster zur Vergegenständlichung der situierten Koordination

Ein Prozessmuster

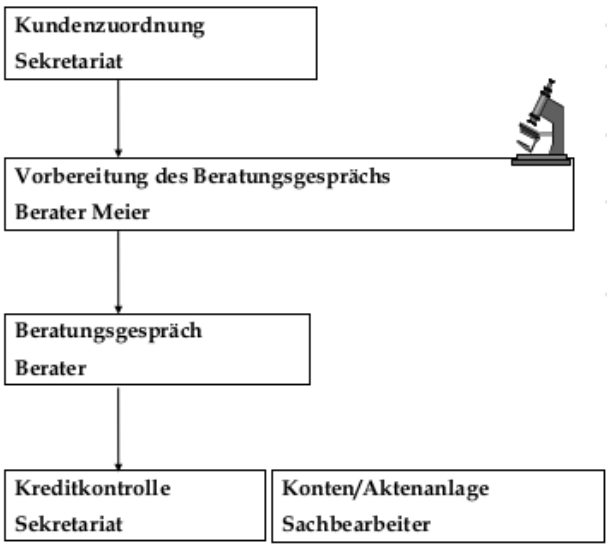
- ist ein gemeinsames Material zur Vergegenständlichung eines kooperativen Arbeitsprozesses,

- legt Verantwortlichkeiten von Personen oder Rollenträgern und Tätigkeiten in einem kooperativen Arbeitsprozess fest,
- besteht aus der Angabe der Abhängigkeiten von und zwischen Tätigkeiten, die bei der kooperativen Arbeit zu erledigen sind und dazu notwendigen Dokumenten.

Eine Situierte Koordination ist die wechselseitige Abstimmung über die Reihenfolge und Zuständigkeit von Tätigkeiten bei kooperativer Arbeit.

→ *Der kooperative Arbeitsprozess wird von den Beteiligten vergegenständlicht, um ihn so nachvollziehbar, steuerbar und selbst wieder zum Gegenstand von Arbeit zu machen!*

Prozessmuster zur Beschreibung fachlicher Abhängigkeiten: Vergabe eines Kleinkredits



- Prozessmuster sind benannt.
- Prozessmuster legen Tätigkeiten fest.
- Tätigkeiten können sequentiell voneinander abhängen.
- Für die Erledigung von Tätigkeiten wird eine Zuständigkeit festgelegt.
- Zuständig ist Person, Rollenträger.

Explizite Koordination

Manchmal reicht es nicht aus die Kooperation durch den Austausch von Material zu realisieren, sondern wir müssen auch die Koordination explizit im System modellieren.
Merkmale:

- In der Regel kennen sich die kooperierenden Personen persönlich und haben ein bestimmtes "Muster" der Zusammenarbeit entwickelt.
- Trotzdem läßt sich kein allgemeingültiger Arbeitsablauf für die Vorgangsbearbeitung definieren. Dafür kann es verschiedene Gründe geben:

    - Viele Tätigkeiten können an einem Arbeitsplatz parallel durch eine Person in einem Arbeitszusammenhang ausgeführt werden.
    - Nur für einen Teil der Tätigkeiten sind Ergebnisse aus vorangehenden Tätigkeiten zwingend erforderlich.
    - Je nach Situation können einzelne Tätigkeiten auch entfallen oder spezielle andere hinzukommen.
- Materialien werden als Teil des angestrebten Arbeitsergebnisses oder zu Informationszwecken zwischen den Beteiligten ausgetauscht, etwa in Vorgangsmappen oder Ordnern. Erst dadurch kann eine Gesamtaufgabe räumlich und zeitlich getrennt erledigt werden.
- Die Anzahl der an der Kooperation beteiligten Personen ist begrenzt. Sie haben einen gemeinsamen Erfahrungshintergrund.
- Die Menge der zu erledigenden Tätigkeiten liegt in einer überschaubaren Größenordnung. Inhalt und Zweck der anfallenden Tätigkeiten sind allen Beteiligten bekannt.
- Die Kontrolle über den weiteren Verlauf der Zusammenarbeit liegt jeweils bei der Person, die den gemeinsamen Vorgang gerade bearbeitet.

Untersützung der expliziten Koordination durch Laufzettel

Ein Laufzettel:

- ist ein Arbeitsgegenstand, der von einem Anwender für einen speziellen Vorgang erstellt wird (wer ist wofür zuständig, welche Reihenfolge, welche Dokumente).
- kann für Routinevorgänge aus einer Sammlung vorformulierter Laufzettel entnommen werden.
- lässt sich in der konkreten Situation verändern oder anpassen.
- ist eine Anweisung für den Transport der damit verbundenen Vorgangsmappe.
- koordiniert die Zusammenarbeit. Er informiert über den Stand der Vorgangsbearbeitung.
- kann mit seiner Vorgangsmappe verfolgt werden. Ein Anwender kann beim Postversandsystem anfragen, an welchem Arbeitsplatz eine Vorgangsmappe derzeit in Arbeit ist und wo sie bisher war.

Kooperationstypen

- Unsere Analyse der unterschiedlichen Kooperationssituationen zeigt Ähnlichkeiten, die wir zu verschiedenen Kooperationstypen zusammengefaßt haben.
- Ein Kooperationstyp abstrahiert von der konkreten Art der Kooperation.
- Merkmale:
  - Charakteristika der Arbeitsteilung,
  - Art der Aufgaben,
  - Art der Koordination,
  - verwendete Kooperationsmittel und -medien.
- Beispiele:
  - Implizite Kooperation über eine gemeinsame Registratur.
  - Explizite Kooperation über Postfächer.
  - Explizite Kooperation und Koordination über Laufzettel.

Arbeitsplatz- und Kooperationstypen

- Arbeitsplatztypen werden durch unterschiedliche Ausstattung mit Arbeitsmitteln und -gegenständen realisiert.
- Kooperationstypen werden durch durch unterschiedliche Ausstattung mit Kooperationsmitteln und -medien realisiert.
- Arbeitsplatz- und Kooperationstypen sind weitgehend orthogonal, d.h. wir finden alle denkbaren Kombinationen in realen Arbeitssituationen.
- Daraus folgt: vielfältige Arbeitssituationen in ähnlichen Anwendungsbereichen lassen sich durch Rekombination von Arbeitsplatz- und Kooperationstypen realisieren.

Kombination von Arbeitsplatz und Kooperationstypen

	Experten-arbeitsplatz	Funktions-arbeitsplatz	Sach-bearbeitungs-platz
Kooperation über Registratur	***	*	**
Kooperation über Postfächer	***	***	***
Kooperation mit Laufzetteln	***	**	***

* möglich
** gut geeignet
*** sehr gut geeignet

Vorgehensmodelle der Software-Entwicklung - Phasenmodelle

Fahrplan

Was ist das zugrunde liegende Problem?

→ Lösungsvorschlag: Wasserfallmodelle

Was ist das neue Problem?

→ Lösungsvorschlag: Partizipative Software-Entwicklung aka Prototyping

Wir diskutieren Ausprägungen:

- Das Spiralmodell
- Rational Unified Process (RUP)
- Agile Entwicklungsmethoden

Gegenstand der Betrachtung & Methodischer Ansatz

Gegenstand unserer Betrachtung ist die Entwicklung interaktiver Anwendungssysteme.

Ein explizites Modell von Zeit spielt in den betrachteten Systemen keine Rolle (Im Gegensatz zur Betrachtung von Soft- und Hard-Realtime-Systemen).

Was kann betrachtet werden?

- Technische Qualität: Innere Produktmerkmale
- Prozeßqualität (wie kommen wir dahin, dass unsere Produkte so aussehen wie wir wollen): Vorgehensmodelle
- Gebrauchsqualität: Äußere Produktmerkmale

Die Softwarekrise

Software war fehlerhaft, nicht wartbar, nicht termingerecht fertig, nicht den Anforderungen entsprechend.

Es gab keine lehrbaren Grundlagen für: Programmiermethodik, Gliederung der Softwareentwicklung, Arbeitsteilung, Projektorganisation, technische Unterstützung . . .

Heute: es gibt lehrbare Grundlagen, aber . . . ist die Softwarekrise überwunden?

→ *Problem gab es seit 50 Jahren. Es handelt sich um ein fundamentales Problem. Weil sonst hätten wir schon eine Lösung gefunden. Es gibt keine Lösung für das Problem Softwareentwicklung.*

Management-Leitfragen für Vorgehensmodelle

Wie kann Softwareentwicklung zeitlich gegliedert werden?
Welche Teilaufgaben / Arbeitspakete ergeben sich?
Wie lässt sich der Projektfortschritt messen?
Wie ist Arbeitsteilung sinnvoll?
Was sind zuverlässige Zwischenergebnisse?
Wie lässt sich der richtige Weg einhalten?
Wie wird frühzeitig Qualität gesichert?

Vorgehensmodelle

dienen zur Benennung und Ordnung von Aktivitäten bei der Softwareentwicklung.

Phasenmodelle

- Traditionell besteht ein Vorgehensmodell aus Phasen, in denen im Wesentlichen eine Projektaktivität bis zu einem vordefinierten Ergebnis verfolgt wird.
- Die Anordnung der Phasen ist üblicherweise sequentiell, d.h. zeitlich folgt eine Phase auf die andere. (es ist keine Parallelität vorgesehen)
- Man muss sich fragen, wie die Phasen miteinander zusammen hängen

Vorgehensmodelle klassisch: Das Wasserfallmodell

- Analyse und Definition
- Entwurf
- Implementation
- Test
- Einsatz und Wartung

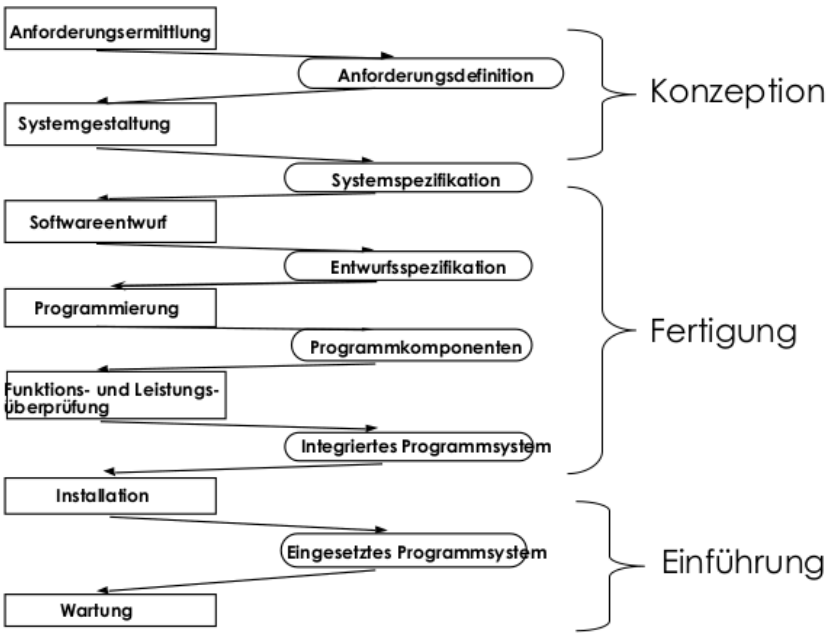
Definition Wasserfall- oder Dokumentenmodell

Die Software-Entwicklung wird als Folge von Aktivitäten betrachtet, die durch Teilergebnisse (Dokumente) gekoppelt sind. Diese Aktivitäten können auch gleichzeitig oder iterativ ausgeführt werden. Davon abgesehen ist die Reihenfolge fest definiert, nämlich (sinngemäß)

1. Analysieren,
2. Spezifizieren,
3. Entwerfen,
4. Codieren,

5. Testen,
6. Installieren und
7. Warten.

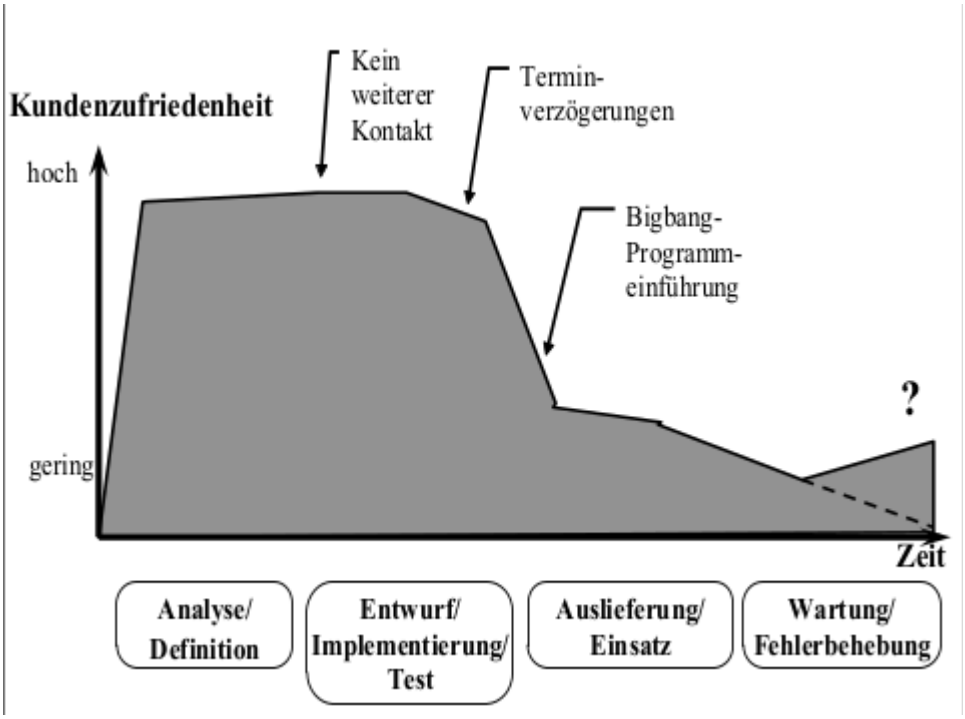
Strenges Wasserfall- oder Einbahnstraßenmodell



Diskussion des Wasserfallmodells

- Das Wasserfallmodell war historisch das erste sequentielle Vorgehensmodell für die Softwareentwicklung.
- Die zahlreichen Variationen fordern:
  - benannte und standardisierte Entwicklungsschritte,
  - die nacheinander durchlaufen werden sollen.
- Rückgriffe auf vorangegangene Phasen sind zwar erlaubt und in der Praxis die Regel.
- Rückgriffe werden als Fehler und Unzulänglichkeiten betrachtet, die bei optimalem Projektverlauf nicht nötig wären und damit minimiert werden sollen.
- Zugrundeliegende Annahme: Das durch die Software zu lösende Problem läßt sich vollständig vor der Implementation spezifizieren. Software-Entwicklung ist Produktion.

Das Kundenorientierte Grundproblem von Wasserfallmodellen



Grenzen linearer Phasenmodelle

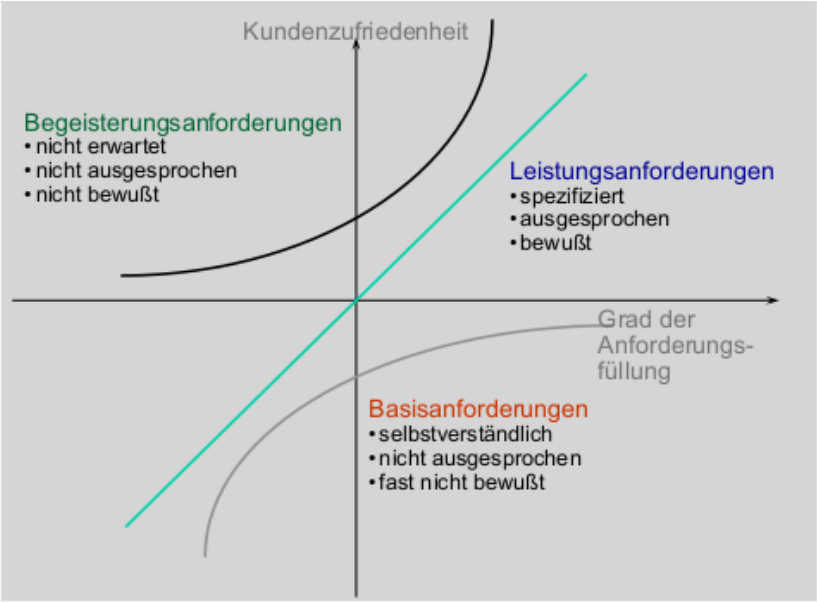
Probleme in der Praxis

- Die lineare Vorgehensweise „Top Down“ kann i.d.R. nicht eingehalten werden.
- Anforderungen können nur teilweise vorab ermittelt werden und ändern sich.
- Dokumente sind keine zuverlässigen Zwischenergebnisse.
- Lernprozess der Beteiligten bei der Entwicklung wird nicht berücksichtigt.
- Trennung von Entwicklung und Wartung bei Software unscharf.

Projektplanung mit Phasenmodellen

- Phasen sind vorgegeben
- Meilensteine = Ergebnisdokumente der Phasen
- Projektfortschritt anhand der Meilensteine gemessen
- Aber:
  - Keine Anhaltspunkte für Feinstrukturierung
  - Keine Rückkopplung aus Erfahrung in den Entwicklungsprozess
  - Keine Berücksichtigung von Qualitätssicherung
  - Kommunikation mit dem Kunden nur am Anfang
- Unklar, ob Projektfortschritt tatsächlich relevant ist.

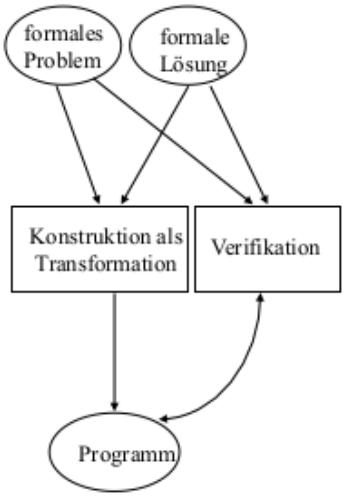
Das Kano-Modell der Kundenzufriedenheit



Einteilung von Programmen nach Lehman: SPE-Programme

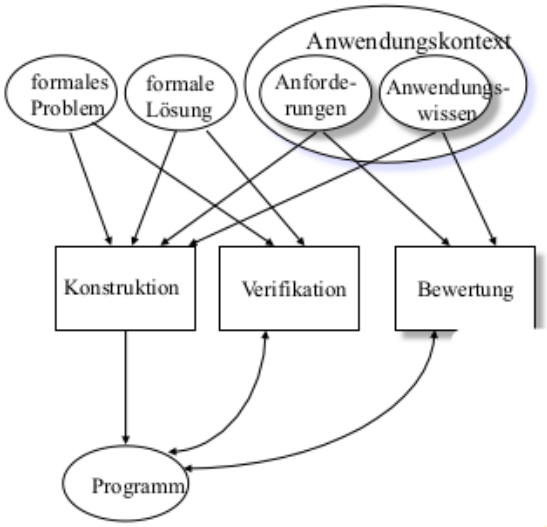
Spezifikationsprogramme (S-Programme)

- besitzen eine vollständige und formale Spezifikation, die die Aufgabenstellung und ihre prinzipielle Lösung beschreibt.
- Es läßt sich unter Verwendung einer mathematischen Methode prüfen, ob die Implementation eines S-Programms vollständig aus seiner formalen Spezifikation abgeleitet werden kann.
  - lassen sich auf der Basis einer schriftlichen Spezifikation erstellen und prüfen. Ihre Korrektheit kann garantiert werden.
- Sie lassen sich getrennt von ihrem Anwendungsbereich entwickeln.
- Beispiel: Berechnung der Fibonacci-Zahlen.



Problemlösende Programme (P-Programme)

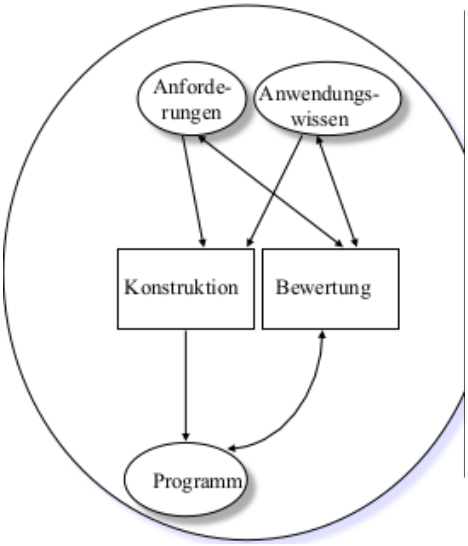
- besitzen eine klar beschreibbare Aufgabenstellung – sie lösen ein bekanntes Problem.
- Die Problemlösung muß mindestens den formalen Bedingungen genügen. Darüber hinaus muß festgelegt werden, wie gut ein P-Programm seine Aufgabe löst.
- Ihre Korrektheit kann garantiert werden aber ihre Verwendbarkeit muß erprobt werden.
- Sie lassen sich nur mit ausreichendem
- Anwendungswissen entwickeln.
- Ihre Anwendung hat keine Rückwirkung auf das ihrer Entwicklung zugrundeliegende Problem.
- Beispiel: Ein Schachprogramm



Eingebettete Programme (E-Programme)

- werden entwickelt, um Unterstützung in einem realen Anwendungsbereich zu bieten.
- Ob und in welchem Umfang ein Zustand als "Problem" angesehen wird und was als Unterstützung benötigt wird, ist abhängig von der Sichtweise des Betrachters.
- Beispiel: Kundenberatung im Wertpapiergeschäft.

„E-Programme sind Programme, die in größere soziotechnische Systeme eingebettet sind. Zwischen den einzelnen Elementen dieser Systeme gibt es Wechselwirkungen, die sich auf die Anforderungen an die Programme auswirken. Werden die Anforderungen durch eine neue Software erfüllt, ändert sich die Situation, und die Anforderungen ändern sich auch. Folglich erfüllt die neue Software die Anforderungen nicht mehr. E-Programme müssen also stetig an die sich ändernden Anforderungen angepasst werden, damit sie brauchbar bleiben.“



Die grundsätzliche Abkehr vom Phasenmodell - Der Autor-Kritiker-Zyklus

- Situationsbestimmung: analysieren
- Zielvorstellung: erarbeiten, bewerten
- Umsetzung: modellieren, konstruieren

Vorgehensmodelle der Software-Entwicklung – Agile Modelle

Grundbegriffe evolutionärer Softwareentwicklung

Evolutionär: Vorgehensweise, die auf Änderungen ausgerichtet ist

Inkrementell: Wachsend, Software-Entwicklung erfolgt in Ausbaustufen

Iterativ: Entwicklungsschritte werden wiederholt

Anwendungsexperten im Entwicklungsprozeß

- Verstärkte Einbeziehung aller beteiligten Gruppen über die gesamte Projektlaufzeit.
- Rollenwechsel vom "Benutzer" zum Anwendungsexperten.
- Ständige Rückkopplung der Ergebnisse durch anwendungsorientierte Dokumente und Prototypen
- Neuorientierung der Entwickler vom Closed Shop zum Dienst am Kunden.

→ Prototypen für Kommunikation mit Kunden

Was ist Prototyping?

Prototyping in der Software-Entwicklung beinhaltet

- bewusst gestaltete Lernprozesse anhand von ausführbaren Vorversionen des geplanten Softwaresystems.
- frühzeitig relevante System-Teile zu entwickeln, erproben und bewerten,
- Rückkopplung für die weitere Entwicklung zu erhalten.
- dient in der Softwaretechnik zur Unterstützung des Entwicklungsprozesses
- dient in der Software-Ergonomie zum besseren Verständnis von Software-Nutzung
- ermöglicht eine auf Experiment und Erfahrung gegründete Vorgehensweise,
- ermöglicht zuverlässige Kommunikation für alle beteiligten Gruppen
- ermöglicht methodische Unterstützung der partizipativen Systementwicklung.

Ein Prototyp sagt mehr als tausend Worte ...

Entwurfsdokumente stoßen an prinzipielle Grenzen:

- Man kann nicht mit ihnen „spielen“.
- Konsequenzen von Entwurfsentscheidungen werden nicht deutlich.
- Technische Durchführbarkeit bleibt ungeklärt.
- Handhabungseigenschaften werden nicht erfahrbar.

Mock-Ups:

- Werden meist aus Pappe oder Papier gebastelt,
- Veranschaulichen die Arbeitsmöglichkeiten mit dem geplanten System
- Unterstützen die Kommunikation mit geringem Aufwand.

Prototypen: ausführbare Vorversionen

- Können demonstriert und benutzt werden
- Sollten leicht änderbar sein
- Werden ggf. in das endgültige System integriert.

Arten des Prototyping

Exploratives Prototyping:

- soll helfen, die Problemstellung aus Anwendersicht zu klären
- wird angewendet, wenn die Problemstellung unklar ist: (wenn man nicht weiß worauf es hinausläuft)
  - Die Entwickler lernen den Anwendungsbereich kennen.
  - Die Benutzer lernen Lösungsmöglichkeiten einschätzen.
- ist der Anforderungsermittlung oder sogar der Projektetablierung zugeordnet:
  - Anforderungen an das Anwendungssystem werden geklärt.
  - Die Anforderungsermittlung (Soll-Konzept) wird konkretisiert.
- dient zur Orientierung über Lösungsmöglichkeiten:
  - Die Kommunikation über die erwünschte Lösung wird abgesichert.
  - Erfordert klare Vereinbarungen über das weitere Vorgehen.
  - Ziel ist in der Regel nicht ein weiter verwendbares System.

Experimentelles Prototyping

- unterstützt die konstruktive Umsetzung der Anforderungen an ein System, indem Risiken des Projektes durch entsprechende Prototypen untersucht werden
- wird angewendet, wenn die technische Umsetzung eines Entwicklungsziels geklärt werden soll:
  - Die Benutzer konkretisieren ihre Vorstellungen über das IT-System.
  - Die Entwickler lernen die Machbarkeit und die Zweckmäßigkeit des Anwendungssystems einschätzen.
- unterstützt den Softwareentwurf:
  - Ausgewählte Teile der Lösung werden überprüft (Oberfläche!).
  - Problematische Entwurfsentscheidungen werden abgesichert.

Evolutionäres Prototyping

- **i**st ein kontinuierliches Verfahren, in dem Anwendungssoftware schrittweise entwickelt und innerhalb einer Organisation an die sich ändernden Randbedingungen angepaßt wird
- bedeutet Konstruktion des Zielsystems in flexibel kombinierbaren kleinen Schritten.
  - Bewährte Prototypen werden graduell erweitert,
  - Keine scharfe Trennung zwischen Prototypen und Ausbaustufen,
- erleichtert schrittweise Umfeldvorbereitung im Einsatzkontext zur Systemeinführung.

→ einen kleinen Auschnitt des Zielsystems von dem wir schon ein Bild haben werden wir Schritt für Schritt kontinuierlich einsetzen

Arbeitsschritte beim Prototyping

Funktionsauswahl:

- Horizontales Prototyping: nur Oberfläche
- Vertikales Prototyping: je eine Funktion ganz

Konstruktion:

- Verwendete Hilfsmittel (z.B. Sprachen oder Werkzeuge),
- Aufsetzen auf existierende Systeme

Bewertung:

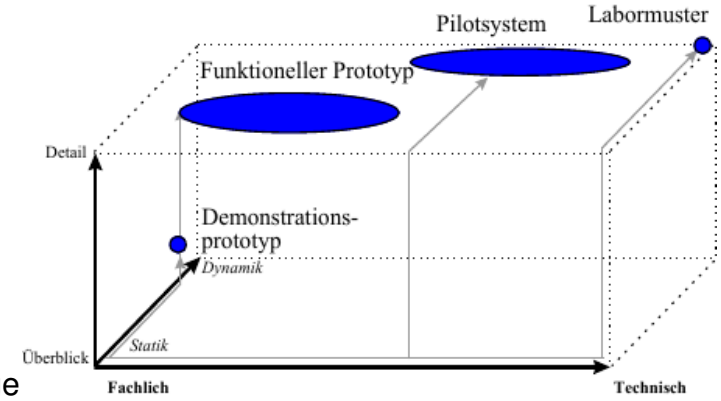
- Wer, mit wem, auf welcher Grundlage?
- Zu unterscheiden: Einarbeitung und Routinebenutzung
- Muss geplant werden

Weiterverwendung:

- Überführung in das Zielsystem (meist erhofft!),
- Wegwerfen (kann sinnvoll sein!).
- Das wichtigste Ziel ist Rückkopplung in den Entwicklungsprozess!

Prototypen

- werden während des gesamten Entwicklungsprozesses erstellt bzw. ein Prototyp wird kontinuierlich weiter bearbeitet.
- Jede Iteration und Etappe wird mit einem lauffähigen Prototyp beendet.
- anhand des Prototyps läßt sich der Fortschritt des Projekts überprüfen.
- Zuerst werden die kritischen Use Cases in dem Prototypen umgesetzt.
- Wir unterscheiden:
  - Demonstationsprototyp
  - funktioneller Prototyp
  - Labormuster
  - T-Prototyp
  - Pilotsystem
- Mit der Konstruktion eines Prototypen wird eine offene Frage beantwortet.



→ durch diese Fragen weiß man welche Art von Prototyping man betreibt und welchen Prototyp man bauen wird

Prototypen im Entwicklungsprozess

- Demonstrationsprototyp (exploratives Prototyping)
  → Zeigt wie ein Anwendungssystem aussehen kann.
- Oberflächenprototyp (experimentelles Prototyping, horizontal)
  → hilft Fragen der Benutzungsoberfläche zu klären,
  → besitzt häufig kaum Funktionalität
- Funktionaler Prototyp (experimentelles Prototyping, vertikal)
  → hilft Entwurfsfragen zu beantworten,
  → zeigt eine Funktion von den Oberfläche bis zur Realisierung.
- Pilotsystem (evolutionäres Prototyping)
  → Technisch „gereifter“ Prototyp,

→ Muss für Routine-Einsatz geeignet sein,

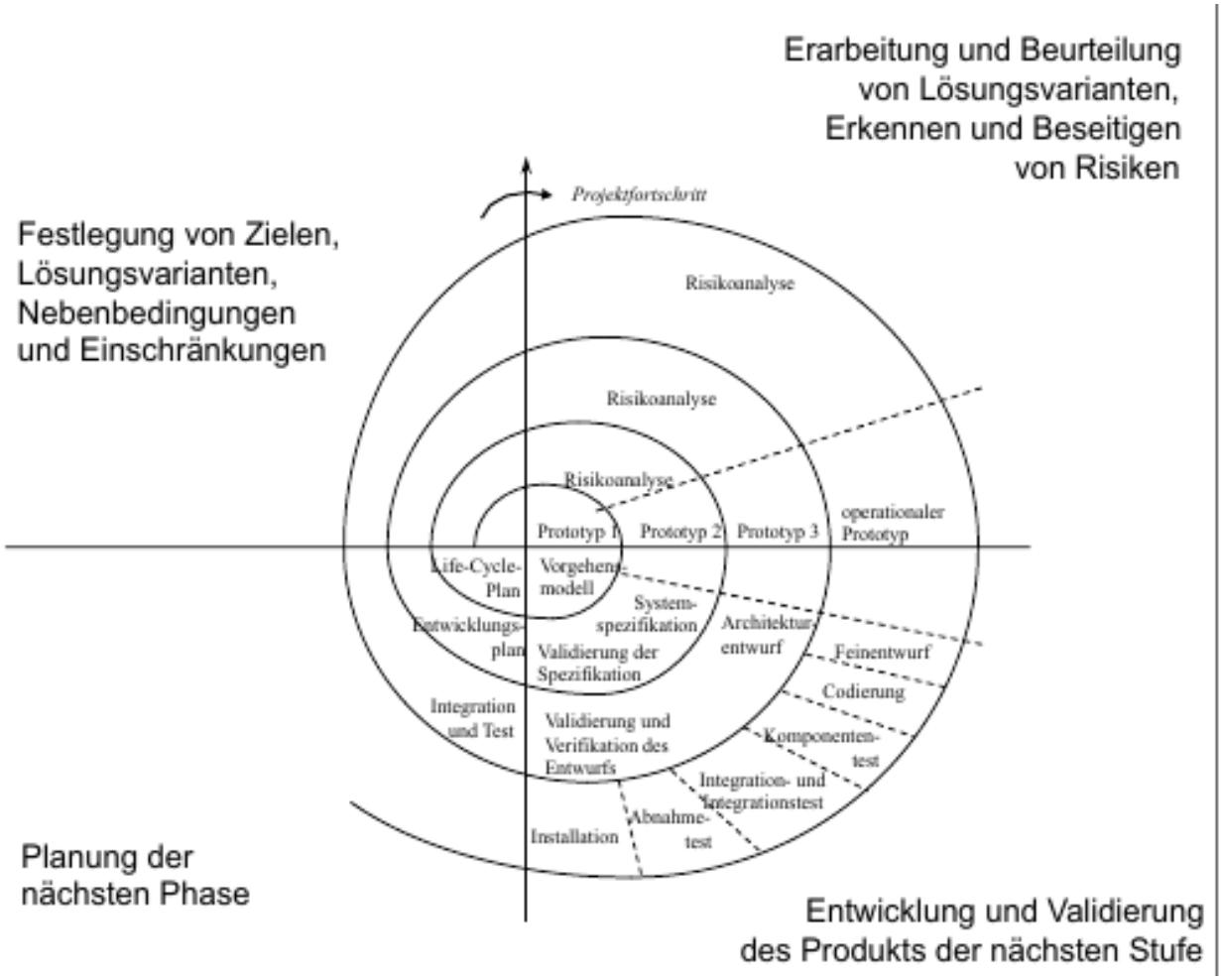
→ wird im Einsatzkontext bewertet,

→ Kern des Anwendungssystems: wird graduell ausgebaut.

Das Spiralmodell

- auch Risk Driven Software Development genannt
- es ist eine Erweiterung des Phasenmodells
- Projektleiter sollten sagen können was die drei größten Risiken des Projektes sind zu anfang des Projektes.
  - Und dann wird ein Prototyp gebaut, um ein/zwei Probleme anzugehen
  - Danach machen wir eine neue Risikoanalayse.
  - Und dann gucken wir ob sich die Risikolage verändert hat
- Am Anfang wollen wir herausfinden was wir bauen wollen → dann Spezifikation → dann Entwicklngsplan
- wenn wir halbwegs sicher sind was wir bauen wollen, können wir uns mit Architektur Entwurf beschäftigen
- Erst danach kommen die Phasen vom Phasenmodell.
- Am schwersten ist es herauszufinden was man bauen will. Das WIE ist einfacher.

Das Spiralmodell als Weiterentwicklung des Wasserfallmodells



Die Phasen bleiben erhalten:

- Sie sind detaillierter beschrieben.
- Sie werden iterativ durchlaufen (Rückgriffe eingeplant!)

Prototypen sind neue Zwischenergebnisse

- Sie bestimmen den Projektfortschritt.
- Sie ermöglichen eine zuverlässigere Evaluierung.

Die Quadranten beschreiben Lernzyklen in der Zeit:

- Definition von Zielen pro Zyklus
- Betonung von Risikoanalyse
- Evaluierung und Rückkopplung explizit

Vorgehensmodell für große Projekte: Unified Process (UP)

Phasen im Unified Process

Inception:

- Vorbereitung, Etablierung und Abgrenzung des Projektes, Festlegung der Zielsetzung, Abschätzung der Projektrisiken.

Elaboration:

- Analyse der Problemdomäne, Ausarbeiten der Architektur, Entwicklung eines Projektplans und Eliminierung des wichtigsten Risikofaktors.

Construction:

- iterative und inkrementelle Entwicklung des kompletten Produktes bis zur Einführungsreife.

Transition:

- Einführung des Produktes und Übergang regelmäßigem Einsatz / Betrieb.

→ Cycle besteht aus vier Phasne, jede Phase hat mehrere Iterationen

Charakteristika des Unified Process

- Use-Case-Driven: Anwendungsfälle (Geschäftsprozesse) bilden den Ausgangspunkt der Modellierung.
  → Erlaubt feinere Projektgliederung im Sinne von Ausbaustufen.
- Architecture-Centric: Software-Architektur zentrales Anliegen im Prozess.
  → Architektur muss im Prozess entwickelt und in jedem Zyklus weiter entwickelt werden.
- Iterative and incremental: Kleine Entwicklungsschritte wiederholt durchgeführt.
  → Orientierung sowohl auf Prototyping als auch auf Ausbaustufen.
  → Bietet auch einen Rahmen für Vorgehensweisen in Agilen Methoden
- Management: Mittelpunkt der Projektaktivitäten.
  → Management als kontinuierlicher Prozess.
  → verweist über das einzelne Projekt hinaus auf den gesamten SoftwareLebenszyklus

Prinzipien Agiler Methoden

- Sehr kurze Zyklen
- Prozess mit wenig Dokumenten
- Prozess steht regelmäßig zur Diskussion
- Collective Code Ownership
- Kunde ist Teil des Entwicklungsprozesses

Die Wertvorstellung

- Individuen und deren Interaktion wiegen mehr als der Prozess und die Werkzeuge
- Lauffähige Software wiegt mehr als ausführliche Dokumentation
- Zusammenarbeit mit dem Auftraggeber wiegt mehr als Vertragsverhandlungen
- Auf Änderungen reagieren wiegt mehr als sich nach einem Plan zu richten

XP-Technik: On-site Customer (Kunde im Team)

- Product Owner muss Entscheidungen treffen
- Anwender im Team.
- Fällt fachliche Entscheidungen.
- Beantwortet Fragen der Entwickler.
- Schreibt Anforderungen (User Stories) oder hilft beim Schreiben.

XP-Arbeitsmittel: User Stories

- Definieren Anforderungen aus Anwendersicht.
- Kurze, keine vollständigen Spezifikationen.
- „Promise for conversation“
- Schätzbar.
- Testbar.
- Können Akzeptanztests enthalten
- User Stories und Akzeptanztests unterstützen die Fortschrittskontrolle im Projekt.

XP-Technik: Planning Game (Planungsspiel)

- Releaseplanung (1 bis 3 Monate).
- Iterationsplanung (max. 4 Wochen)
- Gemeinsame Aufgabe von Entwicklern und Anwender.
- Entwickler schätzen.
- Anwender priorisieren.

XP-Prinzip: Yesterday‘s Weather (Das Wetter von Gestern)

- Stories werden auf Basis von Erfahrung geschätzt.
- Damit schätzt man die Komplexität der Story ein.
- Annahme: Wir schaffen in der nächsten Iteration genauso viel wie in der vorangegangenen Iteration. Nicht mehr!
- Als Einheiten können auch abstrakte Einheiten genommen werden
  - Beispiele: Aufwandspunkte (AP),Gummibärchen

XP-Technik: Small Releases (Kurze Releasezyklen)

- Release-Zyklus 2-3 Wochen
- Erfahrungen mit dem produktiven Einsatz können schnell in die weitere Entwicklung einfließen.

→ Ein Release-Zyklus von 2-3 Wochen kann für den Kunden zu eng sein. Deshalb: Releases und Iterationen.

- Grob Planung in Release-Zyklen von 2-3 Monaten
- Fein-Planung in Iterationen mit Iterationszyklen von maximal 3 Wochen.

XP-Arbeitsmittel: Story-Cards und Task-Cards

Story-Card

- eine User-Story beschreibt eine fachliche Anforderung, d.h. eine Anforderung aus Kundensicht.
- Aufwandsabschätzung
- Bearbeitungsnotizen auf der Rückseite
- Teil-Ziel eines Release, einer Iteration

Task-Card

- wie Story-Card aber mit technischer Anforderung aus Entwicklersicht.
- Arbeitsaufwand <= 2 Paar-Tage
- wird einer Story-Card zugeordnet
- „entsteht“ oft bei der Bearbeitung einer Story.

XP-Technik: Tuning Workshop

- Ziel: Prozess verbessern.
- Zeitpunkt: Am Ende jeder Iteration.
- Teilnehmer: Alle.

- Vorgehen:
  - Was war gut?
  - Was ist verbesserungsfähig?
  - Wie können Verbesserungen erreicht werden?
- Ergebnis: ToDo-Liste.

XP-Technik: Standup Meeting

- Ziel: Transparenz über Projektzustand und -fortschritt.
- Täglich 15 Minuten treffen.
- Alle stehen.
- Kurzer Statusbericht jedes Teammitgliedes:
  - Was habe ich in den letzten 24 Stunden getan?
  - Welchen Zustand habe ich erreicht?
  - Was werde ich in den nächsten 24 Stunden tun?
  - Welche Probleme habe ich im Moment?
  - Inhaltliche Diskussionen außerhalb des Stand-Up-Meetings

XP-Technik: Metaphor (Metapher)

- Eine einfache, präzise Metapher gibt dem System seine innere und äußere Form.
- Konsistenz des Benutzungsmodells.
- Konsistenz der Architektur.
- Beispiele:
  - Adressbuch
  - Schreibtisch mit Ordnern, Mappen und Dokumenten
  - Einkaufswagen (im Web)
  - Kalkulationsblatt
  - Werkzeug

XP-Prinzip: KISS - Keep It Simple, Stupid (Einfachheit)

- „Do the simplest thing that could possibly work“.
- Einfache Entwürfe sind schneller zu implementieren, einfacher zu verstehen, schneller zu ändern und leichter zu testen

XP-Prinzip: YAGNI - You Arent Gonna Need It (Keine Vorratshaltung)

- Keine Technologie auf Vorrat bauen.

- Ungenutzte Funktionalität wird in Akzeptanztests und im produktiven Einsatz nicht getestet.
- Ungenutzte Funktionalität altert sehr schnell.
- Ressourcen werden zur falschen Zeit gebunden.

XP-Technik: Simple Design (Einfaches Design)

- Design / Architekturentwurf ist sehr anspruchsvoll.
- Die Realisierbarkeit wird erst bei der Realisierung geprüft.
- Bewertung von Design ist kontextabhängig.
- Kontext kann und wird sich ändern:
  - Anwendungsbereich
  - Handhabung und Präsentation
  - Technologie.
- Was heute ein gutes Design ist, kann morgen unzureichend oder gar ungeeignet und „schlecht“ sein.

XP-Technik: Collective Ownership (Gemeinsame Verantwortlichkeit)

Der gesamte Code und alle erstellten Dokumente gehören dem Team.

- Jeder darf jederzeit alles ändern.
- Wenn etwas kaputt ist, ist das gesamte Team verantwortlich.
- Spezialwissen wird über Coaching und Schulungen ins Projekt transferiert.

The „Truck-Factor“

Der Truck-Faktor beschreibt die Wahrscheinlichkeit, mit der ein Projekt scheitert, wenn ein Projektmitglied von einem Truck überfahren wird.

→ Der höchste Truck-Faktor (1,0) wird bei einem Spezialistenteam erreicht.

→ Collective Ownership minimiert den Truck-Faktor.

XP-Technik: Continuous Integration (Fortlaufende Integration)

- Mehrmals täglich integrieren.
- Es existiert ständig ein lauffähiges System auf dem Server.
  - Jeder kann sofort von neuem Code profitieren.
  - Neuer Code wird sofort von anderen Entwicklern getestet.
  - Risiko von Doppelentwicklungen wird reduziert.

    - Jederzeit kann ein lauffähiges System erstellt werden.
- Wenn sich Änderungen nicht mehr am selben Tag integrieren lassen, kann es sinnvoll sein, die Änderungen zu verwerfen und am nächsten Tag erneut zu beginnen.
- Ein Integrations-Tool muss optimistische Strategien bei der Entnahme und Rückstellung von Dokumenten unterstützen.

XP-Technik: Test First (Erst Testen; test-infiziert arbeiten)

- Unit-tests (Komponententests) und Akzeptanztests
- Test-First Ansatz als Ideal
- Erster Ansatz: Zu jeder Klasse eine Testklasse, jede Operation einmal aufrufen.
- Ziel: 100% Anweisungsüberdeckung.
- Tests müssen automatisch ablaufen, sie sollen ausprogrammiert sein.
- Tests werden ständig ausgeführt (Minutentakt).
- Der Server-Bestand muss immer alle Tests bestehen.
- Immer, wenn ein Fehler gefunden wurde, erst die Testklasse so erweitern, dass sie den Fehler aufdeckt. Dann die Korrektur durchführen.

XP-Technik: Refactoring

- Refactoring ist die interne Umstrukturierung von Code, ohne die Funktionalität des Systems zu ändern.
- Es ist keine Schande, schlechten Code zu schreiben. Es ist allerdings schändlich, diesen nicht zu verbessern.
- „Never change a running system“ gilt nicht mehr.
- Refactorings müssen in kleine Einzelschritte zerlegt werden (bei größeren Refactorings sind explizite Pläne notwendig)
- Jeder Refactoring-Schritt wird mit Tests abgesichert.

XP-Technik: Pair-Programming (Programmieren in Paaren)

- Immer zwei Entwickler an einem Rechner: Driver und Copilot.
- Ständiges Peer-Review.
- Höhere Code-Qualität:
    - weniger Bugs
    - bessere Schnittstellen
    - weniger Code (einfacheres Design)

    - besseres Design
    - weniger Redundanzen
- Wissen verteilt sich schnell im Team.
- Qualifikationen verteilen sich schnell im Team.
- Der Qualifikationsunterschied sollte nicht zu groß sein.
- Allerdings: Pair-Programming ist gut geeignet für die Ausbildung von Team-Mitgliedern.

Pair-Programming: Arbeitsstile

- Tastatur wird häufig gewechselt. Das muß ohne Sitzplatzwechsel leicht möglich sein.
- Wenn es dem „Zuschauer“ zu bunt wird, nimmt er sich einfach die Tastatur. (Konfliktfähigkeit)
- Der „Besitzer“ der Tastatur erläutert jeweils was er tut. Der andere betrachtet die Entwicklung mit etwas Abstand und gibt aus dieser Perspektive Anregungen und Tipps. (Kommunikationsfähigkeit)
- Paare werden nicht fest zusammengestellt, sondern bilden sich situationsabhängig neu (während der Planung).
- Es können problemlos viele Paare (z.B. 10) gleichzeitig in einem Raum arbeiten.

XP-Technik: Sustainable Pace (Ausdauerndes Tempo)

- Softwareentwicklung hat den Charakter eines Marathon.
- Sprints sind nur kurzfristig durchzuhalten und erzwingen anschließende Ruhephasen.

XP-Technik: Coding Standards

- Richtlinien für die äußere Form des Source-Code braucht man in jedem Projekt.
- Sie erleichtern das Programmieren in Paaren und die gemeinsame Verantwortlichkeit.
- Bei Java gibt es solche Guidelines von Sun bereits fertig definiert.
- Richtlinien können um projektspezifische Eigenheiten ergänzt werden.

Agile mit Scrum und Kanban

Das Scrum Team:

- Product Owner
- Scrum Master
- Development Team

Events

- Sprint Planning (8h*)
  - Arbeit für den akutellen Sprint planen
  - das ganze Team arbeitet zusammen
- Daily Scrum (15min)
  - Update über den aktuellen Stand, Arbeit & Hindernisse
  - Nur das Entwicklungsteam
- Sprint Review (2h*)
  - Präsi des Inkrements: Diskussion wie es weiter gehen kann
  - das ganze Team + ausgewählte Stakeholder, Eingeladen vom Product Owner
- Sprint Retrospective (0.5 -2h*)
  - Erstellen eines Verbesserungs-Plans in Bezug auf Personen, Beziehungen, Prozesse und Tools
  - Das ganze Team

Artefakte in Scrum

- Product Backlog
  - geordnete Liste der Anforderungen, Scrum spricht von Items
  - wird kontinuierlich angepasst
  - verantwortet vom Product Owner
- Sprint Backlog
  - eine Menge von Items aus dem Product Backlog
  - enthält die vom Dev Team für den Sprint nötigen Infos
  - verantwortet vom Dev Team
- Increment
  - ein am Ende des Sprints aus lieferbarer Zustand, der der Definition von „Done“ entspricht

Kanban

- Agile Methode für evolutionäres Change-Management
- viele kleine Änderungen/Verbesserungen schrittweise umzusetzen
- Hohe Transparenz (Kanban Board)
- Pull Prinzip
- Durchlaufzeiten der Tasks minimieren
- WIP – Limits festlegen

	Scrum	Kanban
Objekt(e)	Produkt	Aufgaben
Steuerung	effektive Takte (Sprints)	kontinuierlicher Fluss
Elemente	Stories, Aufgaben	Aufgaben
Kontext	Projektmanagement	Aufgabenmanagement
Verbesserung von	Projektergebnis, Prozess	Prozess
Anwendung	Entwicklungsprojekte	Persönlich, Teams, Projekte

Erst Scrum dann Kanban

- Scrum
  - Zweiwöchige Iterationen erschien zu lang
  - Planungsaufwand in Relation zur Iteration bereits zu hoch
  - Aufwendiger Prozess des Planens
  - Wir entwickeln kein Produkt
- Kanban
  - Extrem flexibel, jede Woche neue Stories besprechen, Refinement
  - Durchlaufzeiten der einzelnen Tasks minimieren
  - geringer Planungsaufwand, aber alle zwei Wochen Review
  - Schätzen um einheitliches Verständnis der Tasks zu erreichen

Schätzen

- ist ein kontinuierlicher Prozess
- findet im Sprint-Planning Meeting statt bei Scrum
- im Refinemeeting bei Kanban

Festpreis und agile Methoden

Plan (SOLL)

- Pflichtenheft als Basis für den Festpreis: beschreibt die Leistungsanforderungen
- Meilenteine

IST Situation

- Backlog: enthält die Anforderungen aus Pflichtenheft, Raum für Veränderungen
  - Begeisterungsanforderungen
  - vergessene Basisanforderungen
- Sprint-Backlog
  - Was ist im nächsten Inkrement enthalten

Softwarearchitektur

Das Problem: Wie soll die Lücke zwischen Anforderungen und dem lauffähigen System (Code) überwunden werden?

→ Funktionale Dekomposition und Objektorientierung

Die Rolle der Softwarearchitektur (Brücke zwischen Anforderungen und Code)

- Grobes Systemkonzept
- Systemweite Abstraktion
- Struktur der Strukturen des Systems
- Wiederverwendung abstrakter Ideen

Vorteile

- Langfristige Planung!
- Verstehbar!
- Analysierbar!
- Wiederverwendbar!

Was ist Architektur?

- Abstraktion: abstrakte Komponenten mit Eigenschaften und Beziehungen (jeweils unterschiedlicher Art)
- Ein System beinhaltet viele Strukturen, nicht nur eine. Für bestimmte Zwecke/Analysen sind jeweils bestimmte Sichten auf ein System relevant.
- Jedes System hat eine Architektur aber sie muss nicht bekannt sein (Reverse Engineering)

- Architektur beschreibt nur Außensicht der Elemente, nicht deren Innenleben (Wirkung auf andere Elemente)

Software-Architektur

Als (Software-)Architektur bezeichnet man die Architektur eines konkreten Softwaresystems. Eine Softwarearchitektur beschreibt den Aufbau des Systems, indem die einzelnen Elemente des Systems und ihre Beziehungen untereinander dargestellt werden.

→ Definition für Softwarearchitektur:

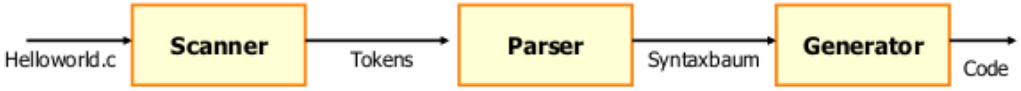
> Eine Softwarearchitektur bezeichnet die Modelle und die konkreten Komponenten eines Softwaresystems in ihrem statischen und dynamischen Zusammenspiel. Sie kann selbst als explizites Modell dargestellt werden. Eine Softwarearchitektur beschreibt ein konkretes System in seinem Anwendungskontext.

Architekturstil

- Als Architekturstil wird eine prinzipielle Lösungsstruktur bezeichnet, die für ein Softwaresystem durchgängig und unter weitgehendem Verzicht auf Ausnahmen angewandt werden sollte
- Die Lösung nach einem Architekturstil schränkt die möglichen Architekturelemente und Beziehungen ein. Programmiersprachlich zulässige Beziehungen oder Schnittstellen werden durch Architekturregeln verboten. Ein Architekturstil ist eine Einschränkung des Designraums.
- Unsere Definition für Architekturstil:
  - Ein Architekturstil legt Arten von Architekturelementen fest und gibt Architekturregeln an, die die Schnittstellen und Beziehungen der Architekturelemente einschränken.

Beispiel für einen Architekturstil

Typischer Architekturstil für Compiler:



Entwickler profitieren von dieser Architektur, weil sie wohlverstanden ist

- Was sind die Bestandteile?
- Welche Verantwortungen haben sie

- Wie sind die Bestandteile miteinander verbunden?

Merkmale:

- Verständlich: „gerader“ Datenfluss durch die Stationen
- Realisierbar: Realisierung jeder Station durch ein Team à Koordination aus Architektur ableitbar
- Erweiterbar: neue Station kann leicht hinzugefügt werden
- Skalierbar: parallele Stationen möglich

Referenzarchitektur (Modellarchitektur)

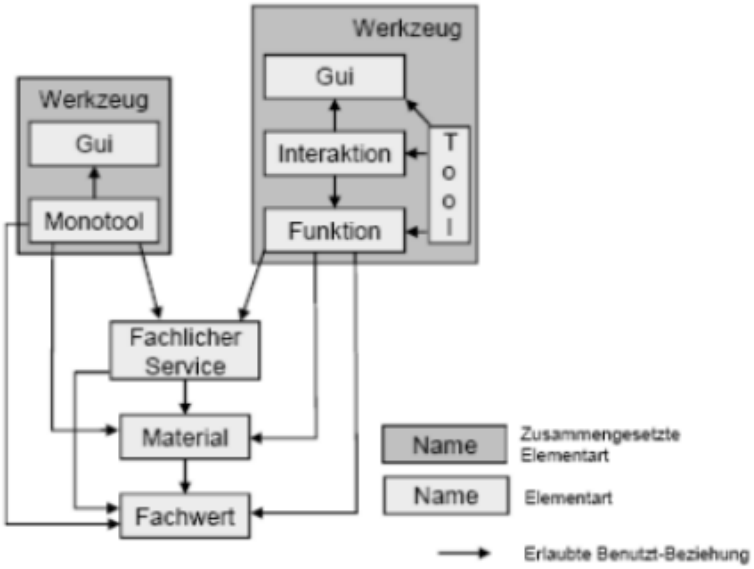
Für flexiblere Softwaresysteme werden aktuell komplexere Architekturstile diskutiert und eingesetzt. Solche Architekturstile definieren Architekturelemente auf der Klassen-Ebene und enthalten Regeln, die sich nicht auf Schichten oder eine festgelegte Menge an Subsystemen und deren Beziehungen und Schnittstellen abbilden lassen.

- Eine Modell- (aktuell: Referenzarchitektur, hz) beschreibt das Soll-Konzept einer Architektur. Sie gibt Regeln vor, die von Softwaresystemen eingehalten werden müssen, die nach ihr entworfen wurden. Eine Architektur, die eine Modellarchitektur als Vorlage hat, kann man als „Exemplar“ dieser Modellarchitektur bezeichnen. Die Regeln der Referenzarchitektur definieren Elemente, die in einem Exemplar dieser Modellarchitektur vorkommen dürfen und welche Beziehungen zwischen diesen Elementen erlaubt sind.
- Definition für Modellarchitektur
    - Eine Modellarchitektur beschreibt die allgemeinen Prinzipien hinter einer Softwarearchitektur. Sie umfasst die grundlegenden Elemente, deren Verknüpfungen und die Regeln, die für eine Softwarearchitektur gelten. Eine Modellarchitektur gibt Anleitung bei der softwaretechnischen Realisierung eines Softwaresystems.

Ausschnitt einer WAM-Referenzarchitektur

(statische Sicht)

es ist nicht veranschaulicht wie es sich zur Laufzeit verhält



Regeln der WAM-Referenzarchitektur

- Materialien dürfen keine FKs kennen
- Die FK bearbeitet das Material
- Zyklische Strukturen sollen vermieden werden
- Kontext-IAKs kennen die volle Schnittstelle ihrer Sub-IAKs
- Die Beziehung zwischen Kontext- und Sub-IAKs ist nicht immer vonnöten
- Die Aufgaben der FK werden in einer Klasse gekapselt
- Es muss möglich sein, zu fragen, ob ein Fachwert nur eine endliche Anzahl von Werten annehmen kann
- Die Tool-Klasse erzeugt die FK
- Werkzeuge sollten die Materialien, die sie bearbeiten unter einem Aspekt kennen
- Die IAK hat im Regelfall keinen direkten Zugriff auf das Material. In einigen Fällen (z.B. um komplexe tabellarische Materialien zu handhaben), darf die IAK Lesezugriff auf ein Material haben

Warum ist Architektur wichtig?

- Verstehen und Kommunikation
  → Gemeinsame Sprache für alle Stakeholder
- Wiederverwendung
  → Etablierung einer A. verursacht Kosten à erhalten!
  → Z.B. Referenz-Architekturen, architektonische Stile
- Konstruktion
  → Grober „Bauplan“ für Entwickler
  → Resultat frühzeitiger Entscheidungen

- Evolution
  → A. ist Resultat der Überlegungen über erwartete Änderungen
  → Festlegung des erlaubten Rahmens für Änderungen
  → Verstehen unterstützen (50% des Wartungsaufwands)
- Analyse
  → Erlaubt die frühzeitige Analyse und Priorisierung hinsichtlich Qualitätszielen
- Management
  → Erreichen einer Architektur ist wichtiger Meilenstein
  → A. ist Grundlage für Projektorganisation
- Kommunikationsplattform
  - Grobe Beschreibung des Systems
  - Kann von allen Beteiligten verstanden werden
  - Nutzt verschiedene Sichten (Reduktion der Komplexität)
  - Zuordnen und Priorisieren von Anforderungen
  - Alternative Lösungen abwägen
- Bauplan für Design & Implementierung
  - Festlegen der wesentlichen Komponenten, Schnittstellen und Interaktionen
  - Systematische Wiederverwendung
  - Rahmen für Projektplanung

Sichten

- Es gibt viele Sichten auf ein System, jeweils mit unterschiedlichen Schwerpunkten
- Je nach Sicht treten bestimmte Aspekte in den Vordergrund, andere werden vernachlässigt
- Die Spezialisierung der Sichten erlaubt es, die Komplexität besser zu handhaben
- Die Sichten sich nicht völlig unabhängig – bestimmte Informationen können mehrfach auftreten
- Zwei Sichten sind konsistent, wenn sie sich nicht widersprechen
- Diskussion darüber, welche Sichten insgesamt existieren und deren Bezeichnungen sind noch im Fluss.
- Generelle Beschreibungsform: Graph
  - Knoten: Einheiten (z.B. funktional, organisatorisch, Ausführung)
  - Kanten: Beziehungen (z.B. Datenfluss, Abhängigkeiten)

Sichten auf Software-Architektur

Es gibt vier Sichten die wir brauchen um eine Software in Referenzarchitektur zu erstellen.

- Fachliche Sicht: Wie kriegen wir raus, wofür eine Software gut ist? Wie sieht das Anwendungsmodell aus?
    - Einsatzkontext
    - Anforderungen
    - Kooperation
    - Benutzbarkeit
- Statische Sicht: Architekturelemente die man statisch betrachten kann,
    - Subsysteme
    - Schnittstellen
    - Schichten
    - Beziehungen
    - Verantwortlichkeiten
- Verteilungssicht: An welchen Stellen ist es möglich Dienste in die Cloud zu verlangern?
    - Rechner
    - Prozesse
    - Netzwerke
- Laufzeitsicht: Welche Einheiten interagieren miteinander?
    - Interaktion
    - Synchronisation
    - Datenaustausch

Darstellung von Software-Architekturen: UML-Dokumenttypen

- Statisch:
    - Klassendiagramme
    - Komponenten- und Package-Diagramme
    - Zustandsdiagramme
    - Kommunikationsdiagramme

- Laufzeitsicht:
  - Objektdiagramme
  - Sequenzdiagramme
  - Interaktionsübersichtsdiagramme
  - Zeitdiagramme
- Verteilung:
  - Verteilungs- (Deployment) Diagramme
- Fachlich:
  - Szenarios mit Sequenz- und Aktivitätsdiagrammen
  - Anwendungsfalldiagramme (UseCase)

Von der Software-Architektur zur Anwendungslandschaft

- In Unternehmen und Organisationen werden heute dutzende/hunderte von Anwendungssystemen eingesetzt. Diese sind mit unterschiedlichen Technologien und Architekturen realisiert. Sie sind meist stark miteinander und mit Systemen außerhalb der Organisation vernetzt.
- Dieses Netz aus Anwendungssystemen eines Unternehmens bezeichnen wir als Anwendungslandschaft.
  - Bsp Anwendungslandschaft: Man sieht welche Außenbeziehungen es gibt, Welche Teile meiner Anwendung wären betroffen wenn ich ein Teil ändere?
- Für diese Größenordnung von Anwendungssystemen geeignete Darstellungsformen werden oft Software-Landkarten genannt.

Softwarekartographie:

Entwicklung eines grundlegenden Begriffsapparats und anschaulicher kartographischer Techniken, um komplexe betriebliche Informationsinfrastrukturen zu beschreiben, zu bewerten und langfristig zu gestalten.

SOLID++ Architektur- und Designprinzipien

Softwarequalität: Motivation

„There is software that‘s easy to write. That would be software that is small, written by one person over a short period of time, used by that one person, used once and then thrown away. Everything else is difficult.“

Zusätzliche Motivation: Software verändert sich!

- Software ist keine Prosa, die – einmal geschrieben – unverändert bleibt.
- Software wird erweitert, korrigiert, gewartet, portiert, adaptiert, …
- Diese Arbeit wird von unterschiedlichen Personen vorgenommen (manchmal über Jahrzehnte).
- Für Software gibt es deshalb nur zwei Optionen: Entweder wird sie gewartet. Oder sie stirbt.

Externe und interne Qualität in ISO 9126

Anforderungen an Entwurfs- und Konstruktionseinheiten

- **Zerlegbarkeit:** Entwurfsprobleme in kleine, weniger komplexe Teilprobleme zerlegen und diese als Konstruktionseinheit abbilden. Konstruktionseinheiten bilden eine einfache Struktur und können weitgehend unabhängig konstruiert werden.
- **Kombinierbarkeit:** Durch die Verwendung und neue Kombination von bestehenden Entwurfs- und Konstruktionseinheiten lassen sich neue Softwaresysteme aus anderen Anwendungsbereichen konstruieren.

- **Verständlichkeit:** Jede Entwurfs- und Konstruktionseinheit sollte weitgehend unabhängig von anderen verständlich sein.
- **Kontinuität:** Eine Anforderungsänderung (technisch oder fachlich) sollte immer nur die Änderung an einer oder wenigen Entwurfs- und Konstruktionseinheiten erfordern.

Symptome eines „verrottenden“ Systems

- **Rigidity – Starrheit:** Das System ist unflexibel gegenüber Änderungen. Entwickler sind sich an vielen Stellen über Abläufe im System im Unklaren, es besteht eine Unbehaglichkeit gegenüber Änderungen. Kleine Refactorings führen zu großen Aufwänden.
- **Fragility – Zerbrechlichkeit:** Änderungen am System führen zu Fehlern, die keinen offensichtlichen Bezug zur Änderung haben. Die Scheu vor Änderungen wächst.
- **Immobility – Unbeweglichkeit**: Es bestehen Entwurfs- und Konstruktionseinheiten, die eine ähnliche Aufgabe bereits lösen, können aber nicht wiederverwendet werden, da zu viel “Gepäck” an dieser Einheit hängt. Eine generische Implementierung oder das Herauslösen ist nicht möglich. Meist wird der benötigte Code kopiert.
- **Viscosity – Zähigkeit:** Bei der Umsetzung einer Änderung ist der “Hack” immer die geeignetste Lösung. Unterschied Zähigkeit der Entwicklung und Zähigkeit der Umgebung

Entwurf nach Zuständigkeiten

- ist eine Entwurfsphilosophie, die von Rebecca Wirfs-Brock Ende der 80er Jahre formuliert wurde.
- Jedes Objekt in einem objektorientierten System sollte für eine klar definierte Aufgabe zuständig sein.
- Dieser Ansatz geht u.a. auf die Forderung nach „Separation of Concerns“ von Dijkstra zurück.

Geheimnisprinzip

- Nur relevante Merkmale einer Entwurfs- und Konstruktionseinheit sollten für Klienten sichtbar und zugänglich sein.
- Details der Implementierung sollten verborgen bleiben.
- Schon allein das Wissen über Interna kann missbraucht werden!

OO-Design Prinzipien: S.O.L.I.D

S ingle Responsibility Principle

O pen Closed Principle

L iskov Substitution Principle

I nterface Segregation Principle

D ependency Inversion Principle

Single Responsibility Principle (SRP)

Jede Klasse soll nur eine fest definierte Aufgabe erfüllen

Gegenbeispiel (Verletzung des SRP):

```
interface Modem {
        public void dial(String pno);
        public void hangup();
        public void send(char c);
        public char recv();
}
```

- Das Modem implementiert zwei Zuständigkeiten:
  - Connection Management (dial und hangup)
  - Data Communication (send und receive)
- Die zwei Zuständigkeiten können sich aus unterschiedlichen Motiven ändern.
-

Open Closed Principle (OCP)

- Komponenten sollen offen für Erweiterungen sein, aber geschlossen für Veränderungen
- Komponenten sollen „wie ausgeliefert“ benutzt werden können, trotzdem erweiterbar sein
- Realisiert mit Polymorphie
- Zukünftige Erweiterungen sollen ohne Änderung des bestehenden Quellcodes möglich sein
- Nur Abhängigkeit von Abstraktionen erlauben, nicht Abhängigkeit von konkreten Implementierungen

Liskov Substitution Principle (LSP)

- Unterklassen erfüllen alle Verträge ihrer Oberklassen, überschriebene Methoden besitzen keine stärkeren Vorbedingungen und keine schwächeren Nachbedingungen
- Unterklassen sollen nicht mehr erwarten und nicht weniger liefern als ihre Oberklassen
- wenn man es nicht einhält hat es dramatische Auswirkungen

  → wenn Teile der Software 2015 entwickelt wurden und 2018 erbt eine neue Klasse von einer klasse von damals, bricht das System zusammen wenn es Änderungen in der alten Klasse gibt und die Verbindung mit der neuen Klasse nicht beachtet wird

Vererbung nur dann benutzen wenn...

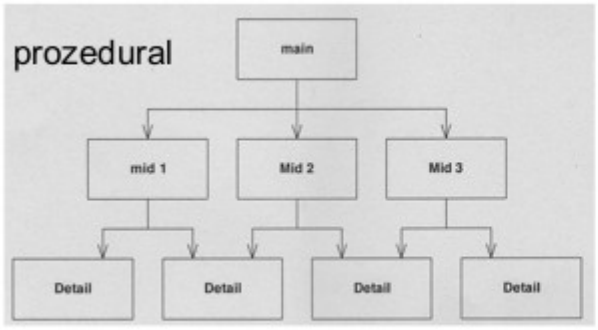
- Beide Klassen in der selben logischen Domäne sind
- Die Unterklasse ein Untertyp der Oberklasse ist
- Die Implementierung der Oberklasse nötig oder angemessen ist für die Unterklasse
- Die Erweiterungen der Unterklasse vor allem Funktionalität hinzufügen

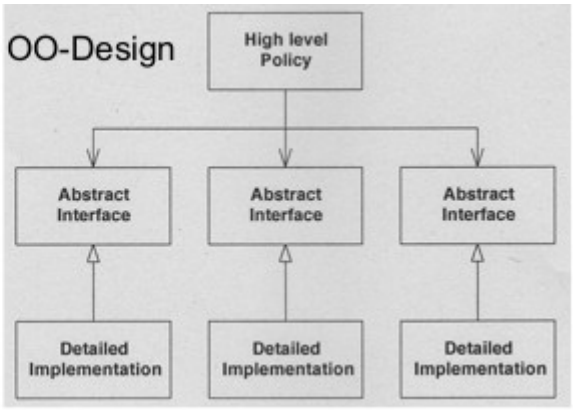
Interface Segregation Principle (ISP)

- Wenn eine Klasse A mehrere Klienten hat, sollten die Methoden des Klienten in ein Interface extrahiert werden. Die klientspezifischen Interfaces werden dann von der Klasse A implementiert.
- „Klientspezifisch“ bedeutet, dass ähnliche Klienten kategorisiert werden, die dann ein spezifisches Interface benutzen. Ansonsten entstünde eine bizarre Abhängigkeit des Services zu jedem einzelnen Klienten.
- Wenn zwei oder mehr Klient-Typen dieselbe Methode verwenden, sollte die Methode zu allen Interfaces hinzugefügt werden.

Dependency Inversion Principle (DiP)

Es sollte keine Abhängigkeit zu konkreten Klassen geben





Inversion of Control (IoC)

- Hollywood-Principle: „Don‘t call us, we‘ll call you“
- Kontrollfluss geht vom Framework aus: Framework ruft eine Methode des Systems auf, nicht umgekehrt.
- Beispiel: ActionListener in Swing („actionPerformed“)

Dependency Injection (DI)

- Abhängigkeiten zu anderen Bausteinen werden an der Schnittstelle eines Bausteins explizit gemacht, indem es Methoden zum Übergeben von Referenzen auf die benötigten Bausteine gibt.
- Durch Dependency Injection wird Software:
  - Gut modularisiert,
  - Leicht veränderbar und
  - Einfach zu testen.
- Ward Cunningham: „DI is a key element of agile architecture“

Vier Varianten von Dependency Injection (DI)

- Bei DI werden Abhängigkeiten von außen gesteuert:
  - Field Injection (mit Metadaten)
  - Interface Injection

    - Setter Injection (mit oder ohne Metadaten)
    - Constructor Injection (mit oder ohne Metadaten)
- Auflösen der Abhängigkeiten kann auf zwei Arten erfolgen:
    - Explizite Verknüpfungen
    - Automatische Konfiguration

Dependency Injection (DI) Framework

Ein DI-Framework ist dazu gedacht, die Konfiguration der Anwendung zu steuern. DI-Frameworks bieten in der Regel eine Reihe von Funktionen, die über das reine Zusammenstecken der konfigurierten Objekte hinausgehen:

- Einheitliche Konfigurationsmechanismen für Systemproperties, Konfigurationsdateien, etc.
- Unterschiedliche Lebenszyklen (Scopes) für die erzeugten Objekte
    - Singleton, Prototype, aber auch Request, Session, Conversation
- Aspect Oriented Programming (AOP) durch Erweiterungen der erzeugten Objekte (Proxies, Weaving)
    - Transactions
    - Security
    - Caches

Release Reuse Equivalency Principle (REP)

- Komponenten, die wiederverwendet werden sollen (Klassen, Packages, Subsysteme) müssen ein Release Management haben.
- Komponenten müssen Versionsnummern haben, die für einen bestimmten Zeitraum abwärtskompatibel sind, weil der Aufwand für ein Update durch den Kunden hoch ist (Import der neuen Bibliothek, Integrationstests, Verifikation der neuen Komponente)
- Deshalb sollten Klassen, die eine Wiederverwendungseinheit bilden auch eine Releaseeinheit sein

Common Closure Principle (CCP)

- „Classes that change together, belong together “
- In grossen Entwicklungsprojekten werden Packages und Subsysteme von Klassen entworfen.
- Das bauen, testen und verteilen der gesamten Anwendung ist nicht trivial.
- Je mehr Subsysteme in einem Release verändert werden, so größer wird auch der Aufwand für Rebuild, Test und Deployment des gesamten Systems.
- Deshalb ist es hilfreich, die Anzahl der veränderten Subsysteme zu beschränken.
- Um dies zu erreichen, werden Klassen zusammen in Subsystemen gruppiert, die bei fachlichen oder technischen Anforderungsänderungen gleichzeitig angepasst werden müssen.

Common Reuse Principle (CRP)

- „Classes that aren ‘ t reused together should not be grouped together “
- Eine Abhängigkeit von einem Package oder Subsystem ist auch eine Abhängigkeit aller Elemente dieser Einheit.
- Wenn der Klient nur von wenigen Klassen der Einheit abhängt und diese keiner Änderung unterliegen, hat er trotzdem den Aufwand der Installation und Revalidierung des Subsystems (Releases).

Acyclic Dependencies Principle (ADP)

„The Dependency between packages must not form cycles “ .

Average Component Dependency (ACD) Metrik

- ACD-Metrik trifft eine Aussage über den Kopplungsgrad der Komponenten
  - Jede komponente hat mind. Grad 1 da es von sich selbst abhängig ist
- Zyklen verschlechtern die ACD Metrik

Don‘t repeat yourself (DRY)

- Auch „Single Point of Truth“ and „Single Point of Maintenance“
- Programmcode soll nicht dupliziert werden, weil
  - es die Änderbarkeit erschwert, da an vielen Stellen Änderungen nachgezogen werden müssen.
  - es zu Inkonsistenzen führen kann.
  - es die Lesbarkeit erschwert, da es den Programmcode „aufbläht“.

Law of Demeter - Original

- Ursprüngliche Definition (als Law of Demeter for Functions/ Methods) war bereits lediglich eine Entwurfsrichtlinie:
- Eine Methode m eines Objektes o sollte ausschließlich Methoden von folgenden Objekten aufrufen:
  - von o selbst;
  - von Parametern von m;
  - von Objekten, die m erzeugt;
  - von Exemplarvariablen von o.
- Umgangssprachlich: „Sprich nur mit Deinen engsten Freunden!“

Law of Demeter - Heute

- Zeitgemäße Anpassungen:
  - If you delegate, delegate fully.
  - Don‘t message your delegate object‘s objects.
  - Don‘t be aware of how some object works. Don‘t work at a lower level than is necessary.
  - Es soll ein Objekt nicht seinem internen Zustand (weitere Objekte als Attribute) gefragt werden. Ich sage dem Objekt, was es machen soll.
- Wenn wir das Traversieren von Objektgeflechten „ausimplementieren“, koppeln wir die Methode zu stark an eine Struktur. Dadurch wird deren Änderung erschwert.

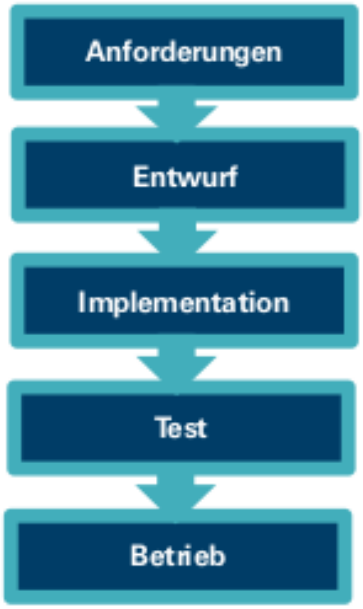
Simple Design

- Wert in extreme Programming
- KISS - Keep it simple and stupid
- Was bedeutet einfach?
- Einfaches Design ist schwer zu entwerfen. Frage: Gibt es einen Entwurf, der bei gleichen Anforderungen an Wartung und Erweiterbarkeit die Aufgabe auch löst?
- Herausforderung bei inkrementeller Vorgehensweise
- Einfache Entwürfe sind aber einfacher zu verstehen und lassen sich auch besser erweitern.
- Steht in Beziehung mit YAGNI - You aren‘t going to need it yet“
- Wieviel Design zum jetzigen Zeitpunkt

Performance im Entwicklungsprozess

Qualitätsmerkmale von Softwaresystemen (ISO 9126)

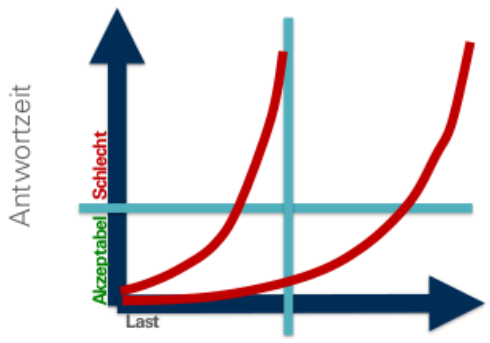
- Funktionalität
  - Angemessenheit
  - Richtigkeit
  - Interoperabilität
  - Ordnungsmäßigkeit
  - Sicherheit
- Zuverlässigkeit
  - Reife
  - Fehlertoleranz
  - Wiederherstellbarkeit
- Benutzbarkeit
  - Verständlichkeit
  - Erlernbarkeit
  - Bedienbarkeit

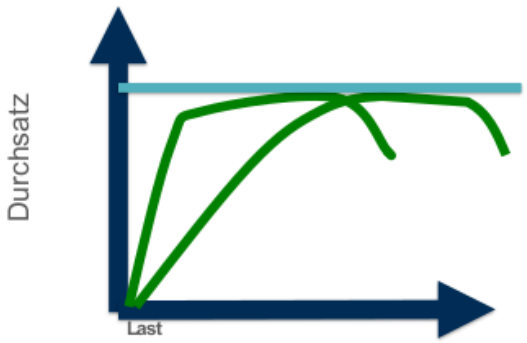


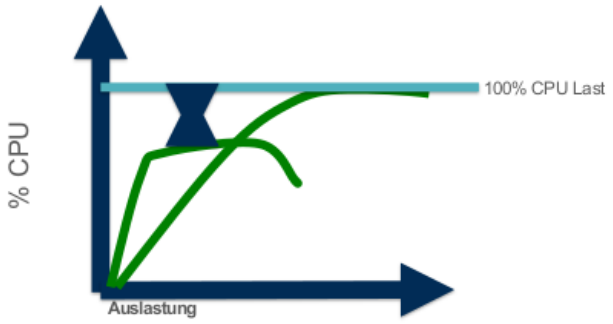
- Effizienz
  - Verbrauchsverhalten
  - Zeitverhalten
- Änderbarkeit
  - Analysierbarkeit
  - Modifizierbarkeit
  - Stabilität
  - Prüfbarkeit
- Übertragbarkeit
  - Anpassbarkeit
  - Installierbarkeit
  - Austauschbarkeit

Grundbegriffe

- Antwortzeit: Zeit, die für die Ausführung einer Transaktion benötigt wird
- Durchsatz: Verarbeitete Transaktionen pro Zeiteinheit
- Last: Eingehende Transaktionen pro Zeiteinheit
- Verfügbarkeit: Zeit, die ein System für den Endanwender verfügbar ist
- Auslastung: Genutzter Anteil einer Systemressource







Littles Gesetz: Anzahl der Transaktionen im System = Antwortzeit * Durchsatz

Performanceanforderungen

- Für welche Teile des System sollten Performanceanforderungen definiert werden?
- Wann sollten Performanceanforderungen definiert werden?
- Um diese Fragen kompetent beantworten zu können muss die Anwendungsdomäne verstanden werden
- Premature optimization is the root of all evil
- Beispiel: „Das Versenden einer Email darf nicht länger als 2sek dauern → wäre unzureichend

Aspekte von Performance Anforderungen

- Anwendungsfall
  - Welche Transaktion betrachte ich?
- Kriterium
  - Antwortzeit, Durchsatz, CPU-Auslastung?
  - Durchschnitt, Maximum, Perzentil?
- Last
  - Bei welcher Last soll das Kriterium erfüllt sein?
  - Welche Transaktionen werden wie häufig auf dem System ausgeführt?
- Daten
  - Für welche Daten muss das Kriterium erfüllt werden?
  - Größe Menge Qualität
- Umgebung
  - Auf welcher Umgebung muss das Kriterium erfüllt werden?

Berücksichtigung der Performance beim Entwurf

- Mengengerüst
  - Was sind die dynamischen Größen?
- Skalierbarkeit
  - Horizontal vs. Vertikal
  - An welchen Stellen sollte das System skalierbar sein?
- Keine „Premature Optimization“
  - Performance vs. Wartbarkeit
  - Man weiß nicht wo die Schwachstellen liegen

Performanceaspekte bei der Implementierung

Programmiersprachenspezifisch

- Es gibt eine Menge von „Tips und Tricks“
- kleine Methoden, StringBuilder statt Kokatenation usw
- Wichtig: keine Premature Optimization

Performance-Tests

- Prüfung ob der Anwendungsfall die Anforderungen erfüllt
  - Messbarkeit herstellen
  - Möglichst reale Daten
  - möglichst reale Last
  - Testumgebung möglichst äquivalent zur Produktivumgebung
  - Fremdsysteme möglichst äquivalent zu produktiven Fremdsystemen
- Je näher Daten, Last und Umgebung an der produktiven Realität sind, desto aussagekräftiger sind die Tests
- Optimal sind die Performancetests automatisiert

Performance-Test Typen

- Lasttest
  - Wie verhält sich das System bei erwarteter Last?
  - Werden die Anforderungen erfüllt?
- Stresstest
  - Wie verhält sich das System bei sehr hoher Last?
  - Ist das System stabil?
- Spike Test
  - Wie verhält sich das System bei plötzlicher Veränderung der Last?
- Endurance Test
  - Wie verhält sich das System bei lang anhaltender Last?

Testaufbau

Testen des Tests

- Verstehen der Testumgebung
- Genauigkeit des Timers?
- 0-Test

Lastgenerierung

- Benutzermuster erstellen
  - Anfangen mit den wichtigsten Fällen
- Caching
  - Nutzung von produktionsnahen Daten
  - Möglichst ähnliches Verhalten der Caches wie in Produktion
- Ramp up
- Apache Jmeter: Open Source Lasttestumgebung
  - Unterstützte Protokolle: HTTP, SOAP, FTP, SMTP ….

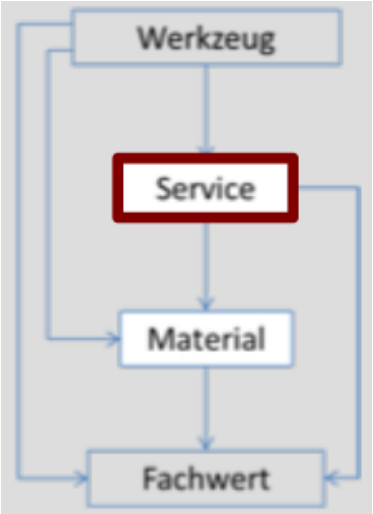
Laufzeitanalyse

- Blick in das System zur Laufzeit
- Verschiedenste Werkzeuge
  - OS-Counter
  - Loganalyse
  - Monitore auf Schnittstellen der Laufzeitumgebung
  - Profiling

Lösungen für Performanceprobleme

- Hängen stark von der Problemursache ab
- Vertikal skalieren
  - Mehr CPU, Speicher, Threads
- Horizontal skalieren
  - Mehr Knoten
- ggf. zunächst Skalierbarkeit herstellen
- Effizientere Algorithm implementieren
- Nutzung von Asynchronität
- Verteilungsschnitt verschieben
- Nutzung von Redundanz (Caching)

Fachliche Services



- kapseln und vergegenständlichen fachliches Wissen
- implementieren Anwendungslogik interaktions- und präsentations**unabhängig**
- Service-Schnittstelle kann in mehrere fachliche oder technische Aspekte unterteilt sein
- können mit anderen fachlichen oder technischen Services zusammenarbeiten (z.B. in hierarchischer Beziehung)
- werden entworfen in Autor-Kritiker-Zyklen im Zusammenspiel mit den Anbindungen der Benutzerschnittstellen
    - dabei ist auf "Reinhaltung" des Service von Benutzungsschnittstellen-Spezifika zu achten
    - sehr gute Autoren-Trennung Fachlicher Service / Benutzungsschnittstelle möglich

Definition Referenzarchitektur

- beschreibt das Soll-Konzept einer Architektur. Sie gibt Regeln vor, die von Softwaresystemen eingehalten werden müssen, die nach ihr entworfen wurden
- Die Regeln der Referenzarchitektur definieren Elemente, die in einem Exemplar dieser Referenzarchitektur vorkommen dürfen und welche Beziehungen zwischen diesen Elementen erlaubt sind
- Eine Architektur, die eine Referenzarchitektur als Vorlage hat, kann man als „Exemplar“ dieser Referenzarchitektur bezeichnen

Ein Blick in die Unternehmenslandschaft: Hostsysteme

- Enthalten viel Fachwissen
- Unterstützen nicht adäquat aktuelle Anforderungen (Reaktivität, Kundenorientierung, Multi-Channeling)
- nicht von heute auf morgen abzulösen
- **Problem:** Host-Programme mit neuen Technologien integrieren

Ein Blick in die Unternehmenslandschaft: Komponenten von Drittherstellern

- SAP etc
- Fertigungstiefe verringern
- auf das Kerngeschäft konzentrieren (Make-or-Buy)
- **Problem:** Eingekaufte Komponenten nahtlos miteinander sowie mit eigenen Komponenten integrieren

Ein Blick in die Technologielandschaft: Unterschiedliche Verteilungstechnologien

- (Web-)Sockets, RMI, Restful Services, EJB, WebServices
- **Problem:** Welche Technologie für welches Anliegen?

Motivation für Service Oriented Architectures (SOA) – Anwendungsfall

- im Unternehmen sind viele (technisch) unterschiedliche Anwendungssysteme im Einsatz
- um einen Geschäftvorteil abzuwickeln, werden mehrere dieser Anwendungen benötigt
- Die Anwendungen müssen hierfür Infos untereinander austauschen
- die einzelnen Anwendungen kommunizieren auf unterschiedliche Weise miteinander
- Wenn der Ablauf geändert wird, müssen unter Umständen andere Anwendungen miteinander kommunizieren
- Diese Anwendungen müssen ggf. so erweitert werden, dass sie sich untereinander verstehen
- Bei unterschiedlichen Kommunikationsarten führt das zu erheblichem Wartungsaufwand

Motivation für Service Oriented Architectures (SOA) – Einheitliche Kommunikation

- zum Verringern des Wartungsaufwand soll eine einheitliche Art der Kommunikation gewählt werden
- Jede Anwendung stellt die benötigten Schnittstellen als Dienst zur Verfügung

- **Hoffnung:** Es entsteht nun deutlich weniger Aufwand, wenn sich der Ablauf ändert, da jede Anwendung zumindest technisch die Voraussetzung zum Datenaustausch mit jeder anderen Anwendung hat
- da die Dienste von jeder Anwendung einzeln zur Verfügung gestellt werden, müssen die jeweiligen Adressen allen Kommunikationspartnern bekannt gemacht werden
- bei Änderungen an einem System (z.B. bei einem Umzug auf einen neuen Server) müssen alle Partner angepasst werden

Motivation für Service Oriented Architectures (SOA) – Service Registry

- um die Anwendungen weiter zu entkoppeln, werden alle Dienste in einem zentralen Verzeichnis eingetragen
- Wenn eine Anwendung einen Dienst benutzt, holt sie sich die Adresse aus diesem Verzeichnis
- Wenn sich jetzt etwas ändert muss es nur im zentralen Verzeichnis geändert werden

Motivation für Service Oriented Architectures (SOA) – Fachliche Services

Anwendungen stellen feingranulare, technische Services bereit:

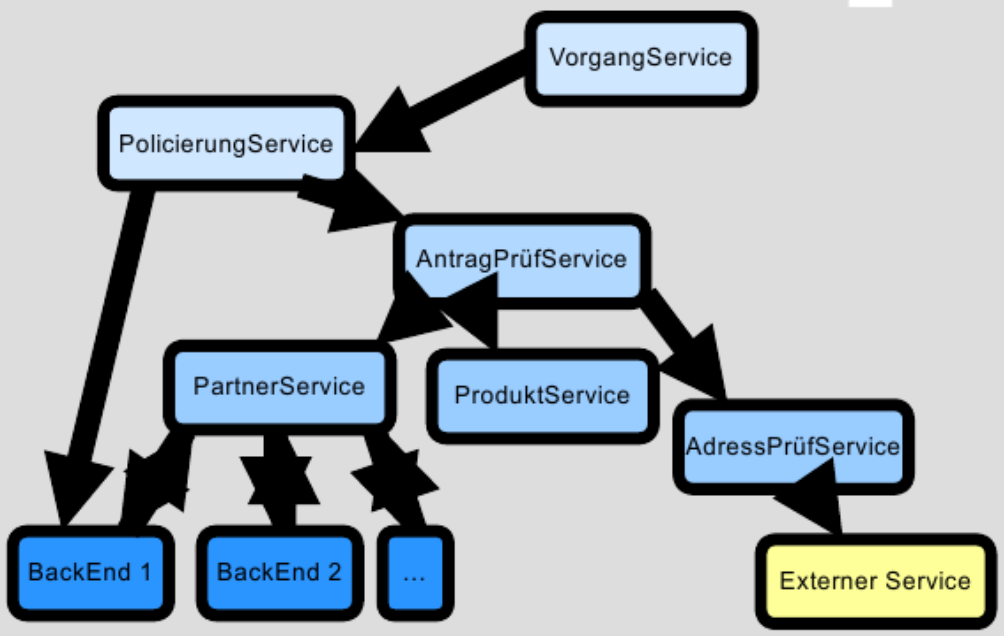
- aus fachlicher Sicht atomare Funktionen benötigen häufig mehrere Dienste aus unterschiedlichen Systemen
- Zum Anlegen eines neuen Kunden könnten folgende Schritte nötig sein:
  - Prüfen der Bonität
  - Vergabe einer Kundennummer
  - Erfassen der Stammdaten
- der Vorgang ist nur erfolgreich, wenn alle Schritte abgeschlossen werden

Technische Services werden zu fachlichen Services zusammengefasst:

- fachliche Services verbergen alle Details zur Implementierung ihrer Funktionalität
- es ist nicht länger nötig zu wissen, welche Datenbanken und Systeme verwendet werden

- Der fachliche Service trägt Sorge, dass die Datenbestände in den verwendeten Systemen konsistent bleiben

Beispiele für Services aus einem Versicherungsprojekt – Übersicht



Zusammenhang – Fachliche Services & SOA

SOA ist ein Architekturstil, bei dem die Funktionalität von Software-Anwendungen in fachliche motivierten Komponenten zsmgefasst wird, die Services (Dienstleister) genannt werden.

- z.B. BLZ-Prüfung
- Adressprüfung
- Tarifrechner

Derselbe Service kann zur Laufzeit parallel von verschiedenen Anwendungen benutzt werden:

- Der Service wird ein Teil der Infrastruktur
- eine andere Form von Wiederverwendung als in Klassenbibliotheken oder Frameworks

→ *Schlussfolgerung: SOA erfordert keine zusätzlichen Investitionen in Tools, Frameworks, Hardware*

Dienstleister/fachlicher Service

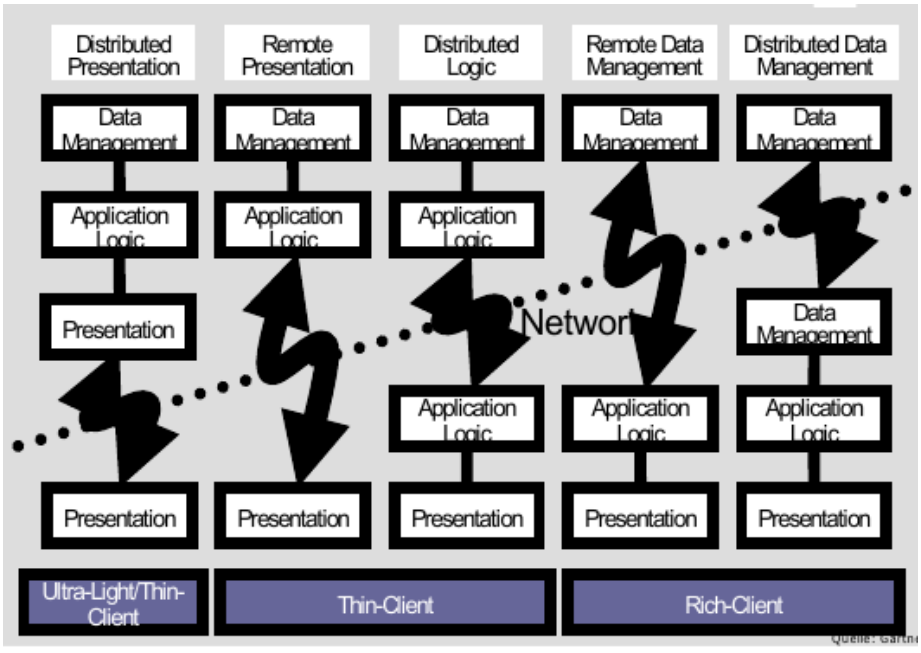
- ein Dienstleister bieter einer Zielgruppe ein Dienstleistungsbündel an
- die Dienstleistung wird synchron oder asynchron erbracht (Warten in der Schlange, Bestellung und Lieferung)
- ein Dienstleister kann für die Realisierung der Dienstleistung interne Materialien sowie weitere Dienstleister verwenden
- ein Dienstleister kann ein oder mehrere Materialien dies Klienten verwenden und modifizieren

Client-Server-Schnitte

Gründe für Verteilung

- Fachliche Anforderungen
  - Multi-Channeling (Desktop, Webtop, mobiles Endgerät)
- Performance
  - Performance durch mehr Prozessoren
  - Skalierung/Lastverhalten bei vielen Benutzern
  - zentrales Caching
- Administration
  - vereinfachtes Deployment (Ausbreitung)
  - zentrale Administration

Mögliche Client/Server-Schnitte



Zusammenspiel der Sichten

Für die Verteilungssicht sind die Fragen zu klären:

- Welche Arten der Verteilung passen zu welchem Einsatzkontext?
- Welche Auswirkung hat die gewählte Verteilung auf die Konstruktion in der statischen Sicht?
- Wie performant ist die gewählte Verteilung?
- Hier in dem Fall ist die Verteilungssicht relevant

Anforderungen an die Verteilungsarchitektur

Ein möglichst einfaches Verteilungsmodell für die Konstruktion von

- Oberfläche als Web- oder Swing-Lösung
- Businesslogik
- Persistenzschicht mit Transaktionsmanagement und Logging-Strategie auf der DB sowie
- Schnittstellen zu Fremdsystemen mit synchronem und/oder ansynchronem Zugriff

damit

- die Komplexität für das Entwicklungsteam gering ist,
- das Entwicklungsteam produktiv bleibt bzw. produktiver wird,
- die Testbarkeit nicht leidet

und sich das Softwaresystem für die Benutzer gleichzeitig

- erwartungskonform und
- performant

verhält.

ThinClient

- Präsentation
- Businesslogik: Einsatz: Implizite Kooperation mit Datenkonsistenz durch Transaktionen, z.B. Antrags- oder Auftragsbearbeitung im Rahmen von Geschäftsprozessen

- Persistenz/Fremdsysteme: DB + Fremdsysteme müssen nicht auf demselben Server laufen wie die Businesslogik, sodass ein weiterer Verteilungsschnitt entsteht

RichClient

- Präsentation
- Businesslogik: Einsatz: Übertragung der Materialien auf den Client gehört zum Benutzungsmodell, z.B. Entwicklungsumgebungen, Textverarbeitung, explizite Kooperation
- Persistenz/Schnittstellen zu Fremdsystemen

Mischform: Werkzeuge auf dem Client

- Materialien und Services auf dem Server
- die Werkzeuge werden über das Netz durch Datentransportklassen versorgt
- **Nachteil:**
  - keine Konsistenz der Datentransportklassen auf dem Client
  - Werkzeuge müssen häufig Anfragen an Services und Materialien auf dem Server stellen

    → Werkzeuge benötigen eigentlich Materialien und Services auf dem Client

Mischform: Werkzeuge und Materialien auf dem Client

- Services auf dem Server
- Transparenter Transport der Materialien auf den Client oder Auschecken bzw. explizites Öffnen im Benutzungsmodell
- Nachteil:
  - große Materialien nur teilweise laden (Performance)
  - Materialien müssen von dem persistenten Kontext getrennt werden, um verteilte Transaktionen zu vermeiden (detach/attach Problematik)

    → LazyLoad und Transaktionen übers Netz sind komplex

RIAS – Integration in Client-Umgebung

- bezüglich der Möglichkeiten in der UI-Konstruktion sind RIAs und Desktopanwendungen vergleichbar
- Aber: Die Integration von RIAs in die Infrastruktur des Clients ist nicht unmittelbar möglich
    - kein unmittelbarer Zugriff auf Drucker/HDD/3D-Engine
    - Zugriff möglich via Plugins: Java(Applets)/AdobeReader
    - Zugriff möglich über lokalen Web-Server
    - Der Browser ist die VM

RIAs – Vorteile gegenüber Desktop-Anwendungen

- Flexibilität
    - minimale Systemvoraussetzungen: Typisch ist Internetzugang und gängiger Browser und AdobeReader
    - Bindung an einen bestimmten Rechner (und damit ggf. an einen bestimmten Ort) entfällt
    - Plattformunbhängigkeit
- Unmittelbarkeit
    - „Bookmarken“ statt „installieren“
    - direkte Updates
    - direktes Provisioning
    - Raubkopien von Web-Anwendungen (fast) unmöglich

ULC Konzept der Firma Canoo

- Swing Elemente sind in zwei Hälften aufgeteilt
- ULC übernimmt Kommunikation und Synchro
- Halb Object and Protocol Design Pattern (HOPP)
- Extension API für spezielle Widgets
- Nachteile
    - JRE auf dem Client bzw. im Browser

    - Entwickler müssen sich ein bisschen um Verteilung kümmern

Zusammenfassung

- Verteilungssicht rückt in Zeiten der Web-Anwendungen in den Fokus
- Verteilung muss zur fachlichen Sicht passen
- Verteilung führt zu Konstruktionsvarianten in der statischen Sicht
- Durch RIA werden Web-Anwendungen zu RichClient-Anwendungen

Software-Architekt

Aufgaben

- Kommunizieren mit Anwendern: Standardisierte Notation + Struktur (Syntax, Semantik & Gliederung)
    - feste Strukturen machen sie schneller, senken Risiko und erhöhen Verständlichkeit
- konstruieren (entwerfen)
- Anforderungen verstehen: mit Stakeholdern arbeiten
- implementieren
- bewerten