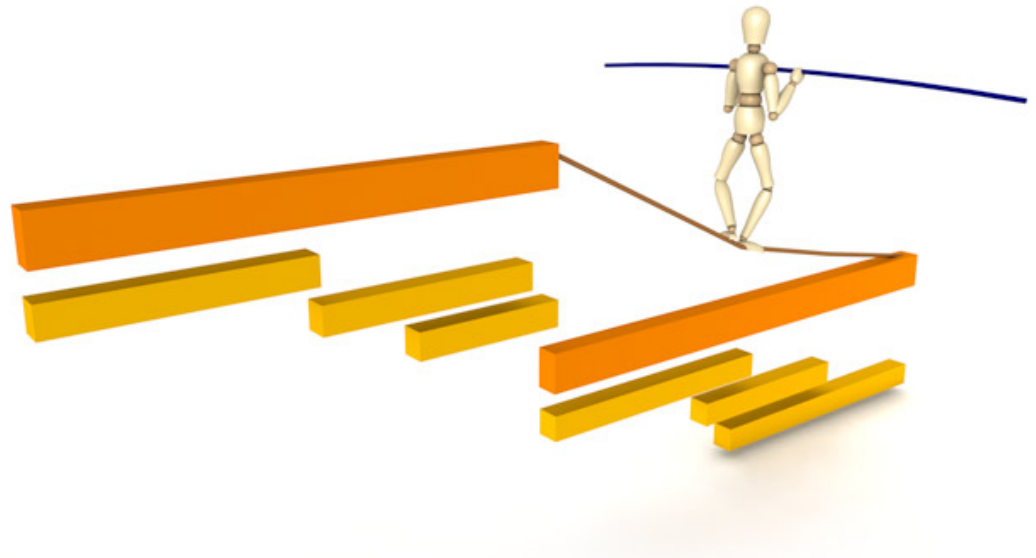


Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion.

Die Inhalte sind urheberrechtlich geschützt.
©2018 Beuth Hochschule für Technik Berlin

SWT - Einführung in die Softwaretechnik



Softwaretechnik - eine Einführung

Lernziele und Überblick

Diese Lerneinheit ist eine einleitende Lerneinheit. Sie enthält auch Übungen, ist im Umfang aber geringer und weniger intensiv als die noch folgenden. Wer sich noch nie mit dem Thema Softwaretechnik beschäftigt hat, sollte diese Lerneinheit durcharbeiten, um ein Gefühl für dieses Fachgebiet zu bekommen.



Gliederung

Themen der Lerneinheit sind:

- Die Begrifflichkeit der Softwaretechnik und des Software-Engineerings.
- Die geschichtliche Entwicklung der Softwaretechnik.
- Ein Blick auf die aktuelle Situation und die Auswirkung von Softwarefehlern - dabei lernen Sie die Bedeutung eines ingenieurmäßigen Vorgehens und der Softwarequalität kennen.
- Best-Practices in der Softwaretechnik.
- Die groben Phasen des Softwarelebenszyklus.
- Die grundlegenden Prinzipien der Softwaretechnik.
- Überblick der Lerneinheiten des Studienmoduls Softwaretechnik.



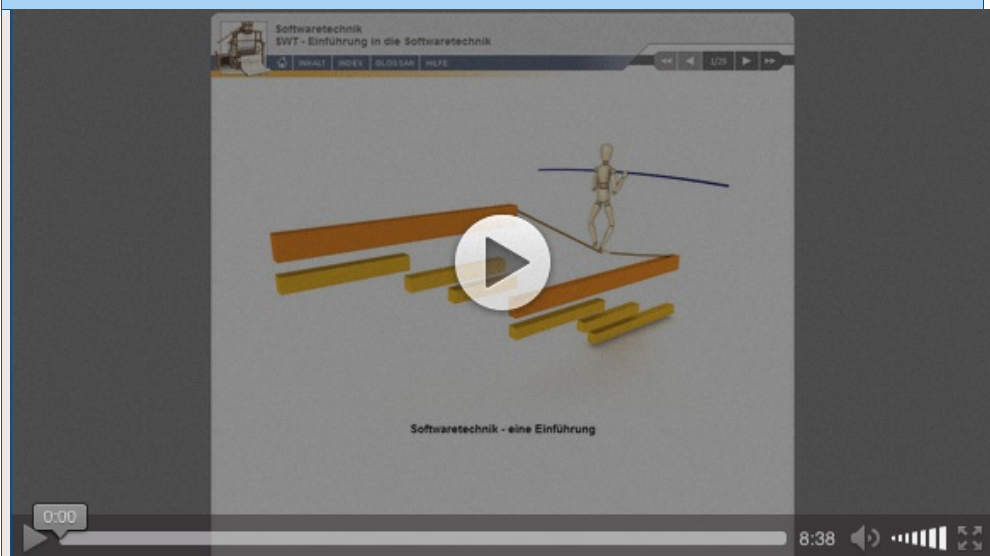
Zeitbedarf

Zum Durcharbeiten dieser Lerneinheit benötigen Sie ca. 60 Minuten und für die Übungen ca. 80 Minuten.



Film

Webkonferenz zur Lerneinheit SWT



© Beuth Hochschule Berlin - Dauer: 8:38 Min. - Streaming Media 13.9 MB

Die Hinweise auf klausurrelevante Teile beziehen sich möglicherweise nicht auf Ihren Kurs. Stimmen Sie sich darüber bitte mit ihrer Kursbetreuung ab.

1 Einführung

Wir wollen zunächst den Begriff Softwaretechnik näher definieren.



Definition

Software

Unter Software kann nicht nur das Endprodukt eines **ausführbaren Programmes** gemeint sein. Zu Software selbst gehört viel mehr. Beispielsweise gehören auch Handbücher, Installationstools, Entwicklerdokumente oder Demonstrationsprogramme dazu. Und dies, obwohl Software im Gegensatz zur Hardware eigentlich eher alle nicht physischen Bestandteile beinhaltet, die auf Computern lauffähig sind. Daher unterscheidet man eher **aktive Daten**, wie ausführbare Programme und **passive Daten**. Letztere sind alle Informationen, die im weitesten Sinne für den erfolgreichen Betrieb des Programmes wichtig sind.



Definition

Technik / Engineering

Unter Technik versteht man, in seiner ursprünglichen Bedeutung, ein **Handwerk** oder eine **Fähigkeit**. Genauer gesagt ist hier eine Ingenieurwissenschaft oder ein Ingenieurwesen gemeint. Als Ingenieur ist man ausgebildet, mindestens ein Fachgebiet sehr gut zu beherrschen. Weiterhin gehören Kenntnisse wie: Praxiserfahrung in der Umsetzung von theoretischem Wissen, analytisches Denkvermögen als auch eine breite Allgemeinbildung dazu.

Der Begriff „**Engineering**“ entspricht dem englischsprachigen Äquivalent um technische und wissenschaftliche Fragestellungen theoretisch und praktisch lösen zu können.

Gesucht wird also **die richtige „Technik“** für die zu erschaffende Software!

Software + technik / Software + engineering

Entsprechend dazu definiert die IEEE Softwaretechnik als:

„The application on a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is the application of engineering of software“

Der Begriff ist vielleicht deshalb auch so schwer zu verstehen, weil er viele andere Gebiete in sich birgt (Mathematik, Produktion, Handarbeit, Kunst, etc.). Vielleicht liegt gerade darin der Reiz, ein Künstler oder Schöpfer zu sein, der etwas erschaffen, zum Leben erwecken und es gleichzeitig Millionen von Menschen sofort zugänglich machen kann.

1.1 Historie

Der Begriff Software Engineering kam in den 60er Jahren auf und wurde auf der von der **NATO** gesponserten **Software Engineering Konferenz** von **1968** in Garmisch-Patenkirchen geprägt.

Einige weitere Eckdaten zur Geschichte der Softwaretechnik:

Datum	Ereignis
1950er	Erste Werkzeuge entstehen, wie Macro Assembler. Erste Compiler
1960er	Zweite Generation an Werkzeugen: Optimierende Compiler. Erste Großprojekte mit mehr als 1000 Entwicklern werden durchgeführt.
1968	Dijkstra schreibt über die Probleme der GOTO-Anweisung.  EDSGER W. DIJKSTRA
1968	Der Datumsstandard YYMMDD wird herausgegeben.
1970	<u>Wasserfallmodell</u> von Winston Royce. Unix in den 70ern.
1971	David Parnas: Information Hiding  DAVID PARNAS
1978	Structured Analysis and System Spec (SA-Method)
1988	<u>Spiralmodell</u> von Barry Boehm
1990er	Komponentenbasierte Entwicklung / <u>OO</u> Entwicklung
1994	Design Patterns von Erich Gamma  ERICH GAMMA
1997	<u>UML</u> (Unified Modeling Language)
2004/05	UML 2

Tab.: Geschichte der Softwaretechnik

1.2 Warum Softwaretechnik? Kleine Fehler - große Wirkung!

Weil der Prozess der Softwareerstellung offenbar ziemlich schwierig ist!

Heutzutage können große Softwareprojekte einen Etat in Milliardenhöhe haben - \$ oder auch €. Selbst kleine anmutende Projekte können schnell Kosten in sechsstelliger Höhe erreichen. Dazu kann der zugrundeliegende Projektplan einfach mal mit dem Stundenlohn von Entwicklern von z. B. 50 € pro Stunde hochgerechnet werden. Leider können schon wenige kleine Fehler ein komplettes System lahmlegen oder zerstören.



Beispiel

Rakete

Im Jahre 2005 verursachte ein Softwarefehler, dass sich die zweite Stufe einer russischen Rakete nicht trennte. Die Rakete transportierte einen CryoSat-Satelliten im Wert von 135 Millionen Euro. [1]



Beispiel

Mars Satellit

Ein Mars Satellit verfehlte 1999 sein Ziel, weil ein Programmierer in einer Funktion Gewichtsdaten in Pfund voraussetzte, die Daten jedoch von der aufrufenden Funktion in Kilogramm bekam. Ein Techniker des JPL (Jet Propulsion Laboratories):



„Yesterday, MCO was approaching the planet to pass within 140 km of the surface and all seemed okay. However, in the last six to eight hours before the approach we saw a 100 km drop and we don't understand why.“ (BBC Online).

Der Schaden lag in ähnlicher Höhe.



Beispiel

Arbeitslosengeld II

Das System startete am 1. Januar 2005. Erstaunlicherweise konnte bei 5% der Empfänger kein Arbeitslosengeld berechnet werden. Der Fehler, wie Heise Online berichtete [2], lag darin, dass die Kontonummern falsch behandelt wurden.

Kontonummern wurden standardmäßig 10stellig angesetzt. Bei allen Kontonummern, die kleiner als 10-stellig waren, wurden die Kontonummern rechtsbündig statt linksbündig aufgefüllt. Beispielsweise bedeutete dies für Kontonummern, die weniger als 10 Stellen hatten, dass ein Ergebnis wie 4711000000 herauskam. Richtig wäre allerdings 0000004711 gewesen! Man stelle sich vor, dieser Fehler wäre bei den zu überweisenden Geldern passiert.

Diese Liste lässt sich weiter fortsetzen:

- Am 4. Juni 1996 explodiert die **Ariane 5**, weil eine 64 bit floating point-Zahl in eine 16 bit signed integer umgewandelt werden musste. Die 64 Bit-Zahl hatte einen größeren Wert, als durch den Integer hätte dargestellt werden können. Dies führt zu einer Exception. Entwicklungskosten der neuen Ariane: 8 Milliarden Dollar.
- Für den Divisionsalgorithmus wurde im **Intel Pentium** Prozessor eine Lookup-Table verwendet, die 1066 Einträge hätte haben sollen. Leider wurden nur 1061 geladen!
- Schon **1962** verursachte bei einer Venus Sonde in einer Fortran Do Schleife ein „~“ Zeichen anstatt eines Kommas ein Scheitern der Mission.

Es gibt noch viele Beispiele. Interessierte Leser seien auf die Webseiten [3], [4] und [5] verwiesen. (Vorsicht: Suchtgefahr beim Lesen und oft auch zeitraubend!)

[1] BBC-News: <http://news.bbc.co.uk/1/hi/sci/tech/4381840.stm>

[2] Heise Online: <http://www.heise.de/newsticker/meldung/54690>

[3] Software Bugs: <http://www5.in.tum.de/~huckle/bugse.html>

[4] Software Horror Stories: <http://www.cs.tau.ac.il/~nachumd/verify/horror.html>

[5] Aviation Safety Network: <http://aviation-safety.net>

[6] Therac-25 - Linearbeschleuniger: <http://de.wikipedia.org/wiki/Therac-25>

1.3 Wie viel und was geht schief in Softwareprojekten?

Schauen wir uns zunächst einige Ergebnisse aus Studien an, die auch heute noch gültig sind und die Notwendigkeit von Softwaretechnik unterstreichen.

Studie des DOD (Department of Defense) aus dem Jahre 1989:

- 30% bestellt aber nicht geliefert
- 46% nicht genutzt
- 19% mit großen Änderungen nutzbar
- 3% mit kleinen Änderungen nutzbar
- 2% genutzt wie geliefert

The Standish Group aus dem Jahre 1995 (Chaos-Report USA):

- 31% der Projekte werden abgebrochen
- 53% der Projekte kosten 189% der ursprünglichen Planung
- Nur 16% der Projekte im Zeit und Kostenplan

Wie sieht es in Deutschland aus? Mummert & Partner stellen im August 2000 in einer Studie das Folgende fest:

- 75% aller Projekte erreichen nicht das geplante Ziel
- 40% aller IT-Projekte scheitern

Fazit:
Ein Großteil
der IT-Projekte
scheitern!

Die Standish Group hat einige der Ergebnisse noch grafisch aufbereitet. Hier eine Grafik, die den Erfolg der Projekte in „succeeded=erfolgreich“, „failed=gescheitert“ und „challenged=noch eine Herausforderung“ über die Zeit betrachtet:

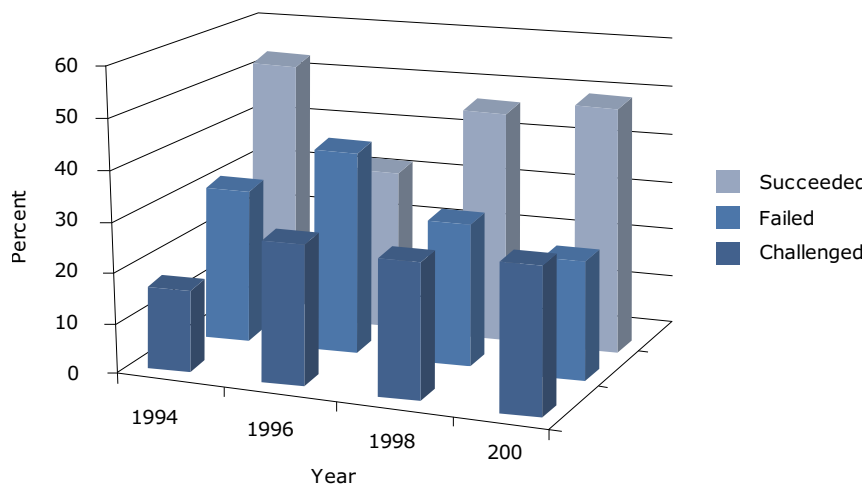


Abb.: Erfolgreiche und gescheiterte Softwareprojekte

[St01]

Interessant ist auch die Feststellung, dass ein signifikanter Zusammenhang zwischen Projektgröße und Projekterfolg besteht. Offensichtlich sind komplexe Projekte nur schwer zu beherrschen.

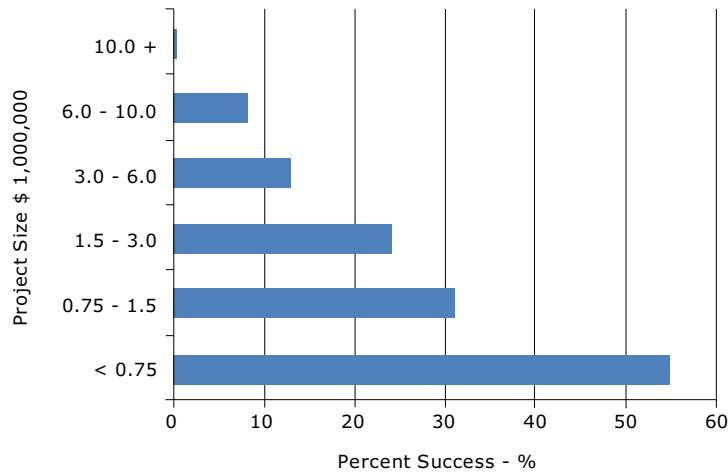



Abb.: Abhängigkeit des Erfolges von der Projektgröße

 [St01]

Ursachen für das Scheitern von Projekten

Die Studie der Standish Group  [Pf98] gibt auch Aufschluss über die Ursachen von Projektfehlern.

- **13,1%** Unvollständige / ungenaue Anforderungen
- **12,4%** Mangelnde Einbeziehung der Beteiligten
- **10,6%** Ressourcenmanagement
- **9,9%** Unrealistischere Erwartungen
- **9,3%** Mangelnde Unterstützung vom Management
- **8,7%** Sich häufig ändernde Anforderungen / Spezifikationen
- **8,1%** Mangelhafte Planung
- **7,5%** Wird nicht mehr benötigt
- **6,2%** Mangelndes IT-Management
- **4,3%** Mangelndes Technologiewissen



Hinweis

Außergewöhnlich ist, dass die inhärente Projektkomplexität oder die von der durch menschliches Handeln herbeigeführte Komplexität von den beteiligten Personen sehr selten als Ursache für das Scheitern von Projekten angeführt wird.

Einige der genannten Punkte kann die Softwaretechnik behandeln und Lösungsvorschläge unterbreiten. Im Studienmodul Softwaretechnik werden diese Punkte ebenfalls betrachtet. Softwaretechnik kann ihnen zwar nicht mehr Unterstützung vom Management verschaffen, aber es verdeutlicht die Bedeutung von Managementunterstützung und es bietet eine gute Hilfestellung zu den Themen: Anforderungen, Ressourcenmanagement, Planung, IT-Wissen und Technologiewissen.

Fehlerquellen

Abschließend noch ein Blick auf die häufigsten Fehlerquellen in Softwareprojekten:

- **32%** Fehler in der Systemlogik
- **19%** Dokumentationsfehler
- **18%** Abarbeitungsfehler am Rechner
- **11%** Fehler im Code
- **6%** Datenbehandlungsfehler
- **5%** Fehler in den Anforderungen
- **5%** Prozessverlaufsfehler
- **4%** Hardwarefehler

2 Worum geht es?

Wie lassen sich die ingenieurmäßigen Hinweise, Ratschläge und Wissenseinheiten der Softwaretechnik kategorisieren? Was lernen wir in den folgenden Lerneinheiten?

Kategorien






DUMKE  [Du01] nimmt eine Einteilung in fünf Kategorien vor:

- **Methoden** (development methods): Richtlinien, Strategien, und Technologien für eine systematische, d. h. phasen- oder schrittweise Entwicklung von Software
- **Werkzeuge** (tools): rechnergestützte Hilfsmittel zur Software-Entwicklung und -Anwendung
- **Maßsysteme** (set of measurements): Menge von Software-Maßen zur Bewertung und Messung der Eigenschaften der zu entwickelnden Software hinsichtlich Eignung, Qualität und speziell des Leistungsverhaltens
- **Standards** (standards): Menge von Richtlinien für die einheitliche und abgestimmte Form der Software-Entwicklung und des zu entwickelnden Software-Systems
- **Erfahrungen** (experiences): (quantifizierte) Kenntnisse über die Entwicklung der Software sowie das Entwicklungsergebnis selbst hinsichtlich des Einsatzes, der Qualität und des Nutzens (als Ingenieurwissen)

Es gibt noch weitere Themen, wie das Projektmanagement, Simulationen, Implementierung, Muster etc., die mehr oder weniger in das Gebiet der Softwaretechnik fallen. Auch diese haben mitunter eine große Relevanz für die Erstellung eines Softwaresystems.

Gremien

Für die oben aufgeführten **Standards** gibt es eine Reihe von Gremien und Organisationen, die für die Softwaretechnik von Bedeutung sind:

- **ANSI**: American National Standard Institute  www.ansi.org
- **DIN**: Deutsches Institut für Normung  www.din.de
- **DoD**: Department of Defense  www.defenselink.mil
- **ACM**: Association of Computing Machinery  www.acm.org
Standards Committee  www.acm.org/tsc
- **ISO**: International Organisation for Standardization  www.iso.org
- **IEEE**: Institute of Electrical and Electronic Engineers
Standards Association  standards.ieee.org
- **DEN**: European Committee for Standardization  www.cenorm.be
- **GI**: Gesellschaft für Informatik e.V.  www.gi-ev.de

2.1 Best Practices

In der Regel geht es bei dem Softwareprojekt um so genannte „Best Practices“. Das sind mehr oder weniger „objektive“ Erfahrungswerte, die ein Projekt zum Erfolg führen. Dabei werden Methoden, Werkzeuge oder Maßsysteme genutzt, die sich aufgrund der Erfahrungen als sehr erfolgreich erwiesen haben.


Methoden

Best Practices in Bezug auf Methoden stellen, beispielsweise die Vorgehensmodelle dar. Diese können beschreiben, welche Vorgehensweise sich bisher vorteilhaft erwiesen hat. Sie geben Aufschluss über spezielle Tätigkeiten oder Dokumente die in einer bestimmten Phase zu erstellen sind oder auch welche Werkzeuge einsetzbar sind.

Werkzeuge

Werkzeuge sind beispielsweise Entwicklungsumgebungen (IDE, Integrated Development Environments), CASE-Tools (Computer Aided Software Engineering), Design- oder UML-Tools (Unified Modeling Language), Werkzeuge zur Anforderungserhebung und -verwaltung sowie Werkzeuge für das Versions- / Buildmanagement.

Standards

Unter Standards fallen beispielsweise Standards für Programmiersprachen wie Java oder C, aber auch Designstandards wie UML oder Codingstandards wie Code Conventions (z. B. für Java  <http://java.sun.com>)

Erfahrungen

Zu den schwerer fassbaren Erfahrungen der Softwaretechnik gehören Punkte der allgemeinen Projekt- / Praxiserfahrung, die man selbst oft erst nach Durchführung von mehreren Projekten erfahren kann. Dazu ein Beispiel von **PETER COAD** [1]:

- **No big bang:** niemals den gesamten Code mit einem Mal zusammenschreiben
- **Tiny step:** in kleinen Schritten entwickeln und die Ergebnisse jeweils genau prüfen
- **Frankenstein principle:** Code, der in einer Nacht „zusammengehackt“ wurde, kann einen noch Jahre beschäftigen
- **Good politics principle:** zunächst sollten die Systemelemente entwickelt werden, die den Auftraggeber am meisten „beeindrucken“ [1]
- **25 words or less:** versuche stets den Sinn des zu entwickelnden Systems in 25 oder weniger Worten auszudrücken
- **Don't touch the WHISKEY principal:** vermeide das WHISKEY-Prinzip (Why in the H— Isn't someone Koding Everything Yet?) (übersetzt vielleicht: „Warum in aller Welt sind noch nicht alle beim Kodieren.“)

1) Unter Kent Becks XP wurde das Prinzip so umformuliert, dass der größte Business Value als erstes entwickelt werden sollte.

3 Der Softwarelebenszyklus

Die Betrachtung des Softwarelebenszyklus ermöglicht dem Softwaretechniker die Elemente der Softwaretechnik besser einzuordnen und den entsprechenden Sprachgebrauch kennen zu lernen. Was ist also der Softwarelebenszyklus?



Definition

Softwarelebenszyklus

Unter Softwarelebenszyklus versteht man die Menge aller unterscheidbaren Phasen, die das Softwareprodukt und die bei der Erstellung beteiligten Personen durchleben. Dies geht üblicherweise von der ersten Idee bis hin zu kompletten Installation und sogar bis zur Ablösung des Softwareproduktes.

Die genannten unterscheidbaren Phasen können wie folgt beschrieben werden:



Definition

Phasen des Softwarelebenszyklus

Eine Phase im Softwarelebenszyklus ist ein klar umrissener Zeitabschnitt, in dem überwiegend ähnliche Tätigkeiten wie Anforderungsanalyse, Implementierung oder Wartung durchgeführt werden. Am Ende einer solchen Phase liegen bewertbare Ergebnisse vor, die unter Umständen sogar getestet werden können.

Betrachten wir einige der „wichtigsten“ Phasen im Software-Entwicklungsprozess.

3.1 Phasen des Software-Entwicklungsprozesses

Entwicklung

Die Software-Entwicklung hat die Aufgabe ein Produkt zu erstellen, das die geforderten Qualitätseigenschaften besitzt. Der Entwicklungsprozess wird im Allgemeinen in eine Reihe von Aktivitäten aufgeteilt, deren Ergebnisse einzelne Teilprodukte sind. Solche Aktivitäten werden nach zeitlichen, begrifflichen, technischen und/oder organisatorischen Kriterien zu Phasen zusammengefasst.

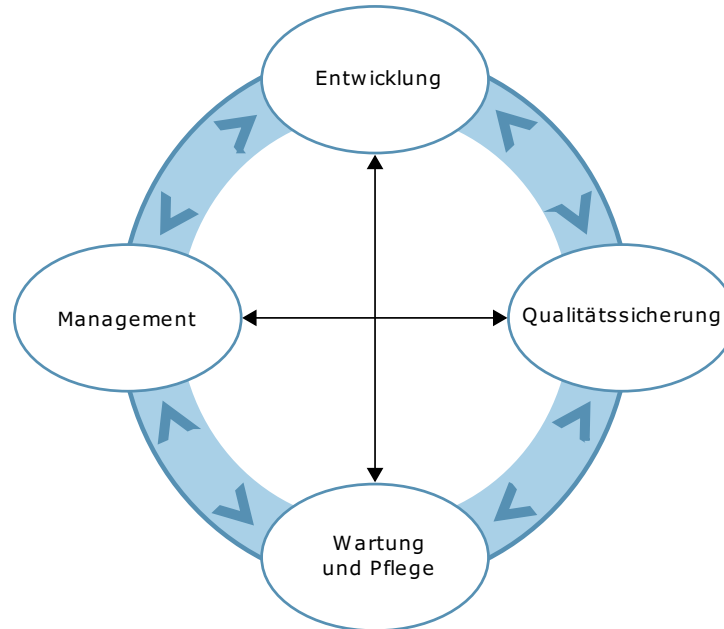


Abb.: Phasen im Software-Entwicklungsprozess

Management

Das Software-Management ist erforderlich, um den Entwicklungsprozess zu **planen**, zu **organisieren**, zu **leiten** und zu **kontrollieren**. Entwicklung und Management im Software-Entwicklungsprozess sind vielfach voneinander abhängig. Die Einführung neuer Methoden und Tools kann zu Veränderungen in den organisatorischen Strukturen führen.

Qualitätssicherung

Die Sicherstellung einer geforderten Softwarequalität muss durch eine entwicklungsbegleitende Software-Qualitätssicherung erreicht werden. Dazu sind eine Reihe von konstruktiven und analytischen Maßnahmen durchzuführen. Die Maßnahmen der Qualitätssicherung werden dabei sowohl von der Entwicklung (die verwendeten Methoden, Tools und Programmiersprachen erfordern bestimmte Sicherungs- und Prüfmaßnahmen), als auch vom Management durch organisatorische Zuordnung der Qualitätssicherung zum Entwicklungsprozess beeinflusst.

Wartung und Pflege

Nachdem ein Softwareprodukt zur Anwendung freigegeben wurde, beginnt die Wartung und Pflege, d. h. es gilt alle nach Inbetriebnahme auftretenden Fehler zu beseitigen, das Anwendungssystem an veränderte Bedingungen anzupassen und es bei Vorliegen neuer Anforderungen weiterzuentwickeln. Art und Umfang der Wartungstätigkeiten sind auch von der Gewährleistung der Qualitätssicherung abhängig, d. h. eine schlechte Produktqualität führt zwangsläufig zu häufigeren Fehleranteilen.

3.2 Phaseneinteilung

Im Softwarelebenszyklus kann die Aufgliederung der Phasen unterschiedlich vorgenommen werden. Nachfolgend für das Studienmodul wird in Anlehnung an BALZERT [Bal98] eine erste grobe Phaseneinteilung vorgeschlagen werden.

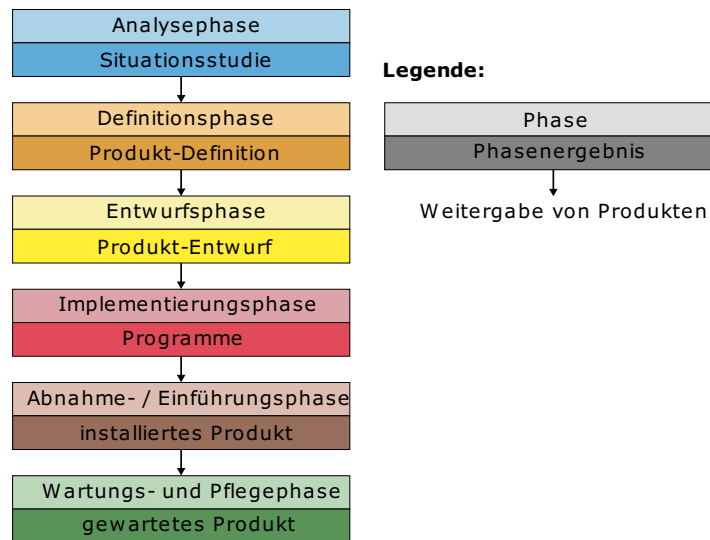


Abb.: Phaseneinteilung nach Balzert.

Wie aus der Abbildung ersichtlich wird, ist jede Phase gekennzeichnet durch entsprechende **Ergebnisse**, die sie hervorbringt. Weiterhin lassen sich jeder Phase notwendige Aktivitäten / Arbeitsschritte und Kriterien zur Überprüfung des erfolgreichen Abschlusses zuordnen. Zur Beschreibung des Software-Entwicklungsprozesses mit Hilfe definierter Phasen werden in der Fachliteratur verschiedene Phasenmodelle (Software-Life-Cycle-Modelle) verwendet.

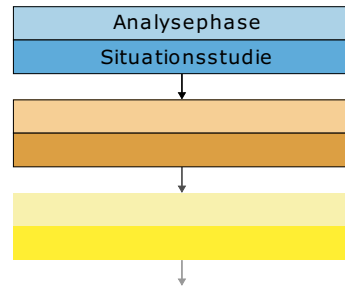
Auf die Problematik der unterschiedlichen Modelle soll hier nicht näher eingegangen werden. Phasen- und Vorgehensmodelle werden in anderen Lerneinheiten behandelt.

Wie bereits erwähnt gibt es mehrere Sichtweisen auf die Zyklen im Softwarelebenszyklus. Andere Autoren definieren beispielsweise vor dem Entwurf die Phasen: Problemdefinition, Anforderungsanalyse und Spezifikation.

Beachten Sie bei der obigen Abbildung, dass die Pfeile zwischen den einzelnen Phasen nur dann gelten, wenn es sich um ein streng „wasserfallartiges“ Vorgehen handelt. In anderen Modellen **vermischen sich** diese **Phasen** ständig, reihen sich beispielsweise in Spiralen mit Rücksprüngen ein oder es werden alle Phasen quasi kontinuierlich gelebt. Es sind also auch gut Doppelpfeile oder eine komplette Vernetzung aller Phasen denkbar. Schließlich steht der Begriff „Zyklus“ in Softwarelebenszyklus ja auch für etwas Wiederkehrendes.

Im Folgenden werden für jede Phase die Ziele, die erforderlichen Aktivitäten und die Ergebnisse in kurzer Form dargestellt.

3.3 Analysephase



Ziele:

- Beschreibung der Ausgangssituation
- Definition der Ziele für die Softwarelösung im Anwenderbereich
- Ressourcenplanung (personell, finanziell, zeitlich und technisch)

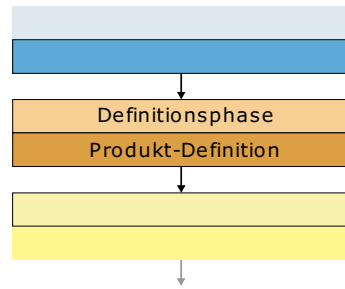
Aktivitäten:

- Voruntersuchung durchführen (u.a. Marktanalysen, Trends, Kundenanfragen)
- Erhebung der Ist-Situation im Untersuchungsbereich
- Erstellen einer groben Übersicht über die Bestandteile des geplanten Systems
- Durchführbarkeitsuntersuchungen
- Erstellen eines Projektplans mit Ressourcen und Projektschritten
- Wirtschaftlichkeitsbetrachtung

Ergebnisse:

- Situationsstudie, Projektkalkulation / Projektplan

3.4 Definitionsphase



Definitionsphase : Ziele:

- Definition der Anforderungen an das geplante Softwaresystem (Festlegung, was es leisten soll) aus der Sicht des Anwenders
- Definition von Prämissen für die Realisierung des Softwaresystems

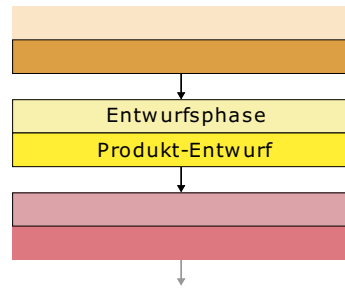
Aktivitäten:

- Anforderungen ermitteln
- Anforderungen festlegen und beschreiben
- Validierung der definierten Anforderungen (Prüfung auf Vollständigkeit, Konsistenz und technische Durchführbarkeit)

Ergebnisse:

- Datenmodell, Funktionsmodell, Benutzerschnittstelle
- und Projektplan
- Produktdefinition / Produktanforderungen beziehen sich auf Daten, Funktionen, Verhalten und Benutzeroberfläche des geplanten Anwendungssystems

3.5 Entwurfsphase



Ziele:

- Erarbeitung eines softwaretechnischen Entwurfs

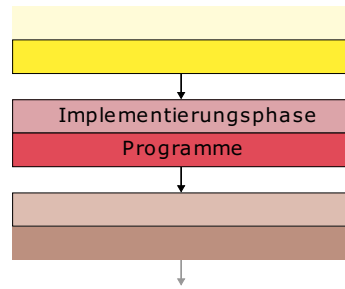
Aktivitäten:

- Analyse der Einsatzbedingungen sowie der Umgebungs- und Randbedingungen
- Definition der Systemkomponenten
- Entwurf der Systemarchitektur (Komponenten, deren Zusammenwirken und Anordnung)
- Entwurf der Schnittstellen und Festlegen ihrer Wechselwirkungen
- Validierung der Systemarchitektur

Ergebnisse:

- Beschreibung der Struktur des Software-Entwurfs
(z. B. Zusammensetzung des Programmes, Programmfluss, Hierarchie, Module, etc.)
- Entwurf der Systemkomponenten

3.6 Implementierungsphase



Ziele:

- Umsetzung der Entwurfsergebnisse in eine am Rechner ausführbare Form
- Sicherung der Richtigkeit und Fehlerfreiheit der Ergebnisse der Systemimplementierung

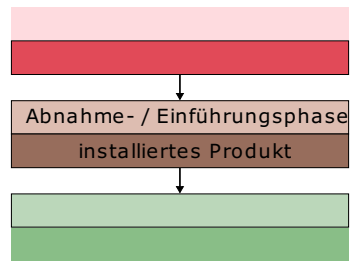
Aktivitäten:

- Verfeinerung der Algorithmen für die einzelnen Systemkomponenten
- Kodierung / Generierung der Algorithmen
- Überführung des logischen Datenmodells in ein physisches DB-Konzept
- Prüfung der semantischen Richtigkeit der Systemkomponenten
- Test der Syntax und gegebenenfalls Korrektur der fehlerhaften Systemkomponenten
- Prüfung des Zusammenwirkens der Systemkomponenten unter realen Bedingungen
- Feststellen und Beseitigen der Fehler im Softwaresystem

Ergebnisse:

- Quelltexte für die Systemkomponenten
- Protokolle der Komponententests, Systemtests
- Generierte Datenobjekte und Datenstrukturen

3.7 Abnahme- und Einführungsphase



Ziele:

- Abnahme des fertiggestellten Softwareproduktes und Einführung beim Anwender

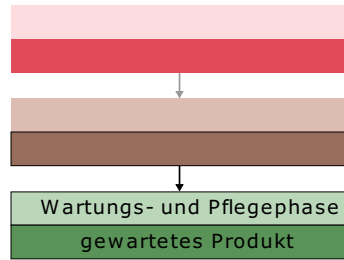
Aktivitäten:

- Übergabe des Softwareproduktes einschließlich Dokumentation an den Auftraggeber
- Durchführung eines Abnahmetests
- Installation des Softwareproduktes in dessen Zielumgebung
- Schulungen der späteren Benutzer der Software
- Inbetriebnahme des Produktes

Ergebnisse:

- Übergebene Softwareprodukte
- Dokumentationen

3.8 Wartungs- und Pflegephase



Ziele:

- Gewährleistung der produktiven Nutzung des Anwendungssystems
- Fehlerbeseitigung am Softwaresystem in der Betriebs- bzw. Nutzungsphase
- Realisierung notwendiger Änderungs- und Erarbeitungsarbeiten

Aktivitäten:

- Wartung: Stabilisierung und Optimierung der Produktinstallation
- Pflege: Anpassungen und Erweiterungen

Ergebnisse:

- Zufriedene Kunden

4 Prinzipien des Software Engineerings

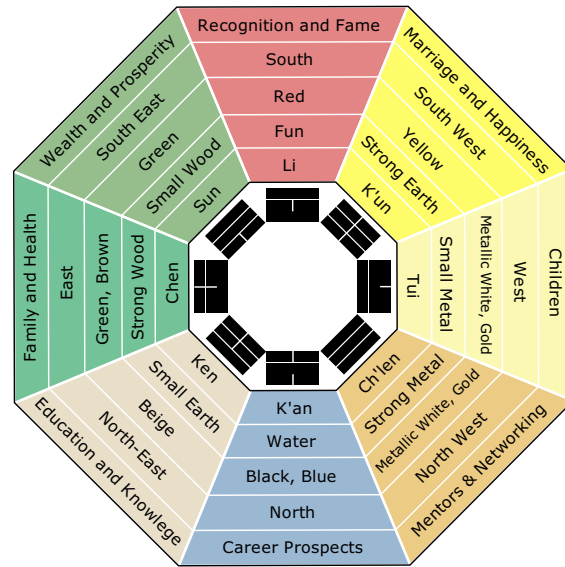


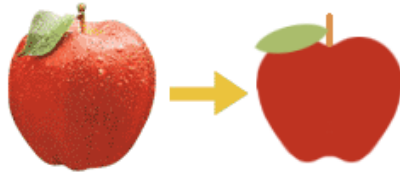
Abb. : Prinzipien
des Feng Shui

Software Engineering zeichnet sich durch eine hohe Innovationsgeschwindigkeit aus. Das gilt insbesondere für die Entwicklung von Tools und Methoden. Im Vergleich dazu sind **Prinzipien** **allgemeingültig** und **abstrakt**. Sie bilden die theoretischen Grundlagen, z. B. für die Methodenentwicklung.

In diesem Kapitel werden einige ausgewählte Prinzipien vorgestellt, um eine Basis für das Verständnis der, in den weiteren Lerneinheiten, behandelten Methoden zu schaffen.

Im Allgemeinen gelten diese Prinzipien für die Ingenieurwissenschaft, haben aber für die Softwareentwicklung eine besondere Bedeutung, da es darum geht **komplexe und interagierende Systeme** zu entwickeln. Diese lassen sich um so besser beschreiben, modellieren, entwickeln und testen, je besser man es schafft Dinge zu abstrahieren oder zu strukturieren. Das ist etwas, was vielen Informatikern schwerfällt - es ist häufig zu reizvoll Dinge einfach sofort zu entwickeln.

4.1 Prinzip der Abstraktion



Die Abstraktion ist eines der wichtigsten Prinzipien im Software Engineering. Abstraktion beinhaltet **Verallgemeinerung**, **das Erfassen des Wesentlichen** sowie **das Weglassen von Besonderem und Einzelem**. Abstraktion ist das Gegenteil zur **Konkretisierung**.

Das Ergebnis der Abstraktion ist ein Modell. Solch ein Modell der realen Welt entsteht durch Abstrahieren vom Konkreten und Weglassen von Nebenaspekten. Abstraktion und Konkretisierung sind nicht absolut sondern relativ, das bedeutet, dass das Ziel der jeweiligen Abstraktion (Modellbildung) bestimmt, welche Aspekte wesentlich und welche unwesentlich sind. Es wird auch der Begriff der Abstraktionsebenen benutzt, um Abstufungen hervorzuheben. Bei der Software-Entwicklung findet ein ständiges Wechselspiel zwischen Abstrahieren und Konkretisieren statt.

Für den Software-Entwicklungsprozess heißt das, sich einerseits dem System von einer sehr abstrakten Ebene her zu nähern und es in seiner Modellierung und Beschreibung immer feiner zu konkretisieren. Andererseits müssen Komponenten, deren genaue Struktur klar ist, auf einer höheren / abstrakteren Ebene beschrieben werden, um überhaupt darstellen und kommunizieren zu können.

Klassenbildende- und Komplexbildende Abstraktion

Im Rahmen der Modellbildung sind insbesondere zwei Formen der Abstraktion bekannt.

1. Klassenbildende Abstraktion

Das Verallgemeinern von Elementen durch Weglassen der sich unterscheidenden Merkmale nennt man klassenbildende Abstraktion.

2. Komplexbildende Abstraktion

Komplexbildend zu abstrahieren bedeutet, mehrere Elemente als Teile eines neuen Ganzen zu verstehen und unter einem gemeinsamen Begriff zusammenzufassen.

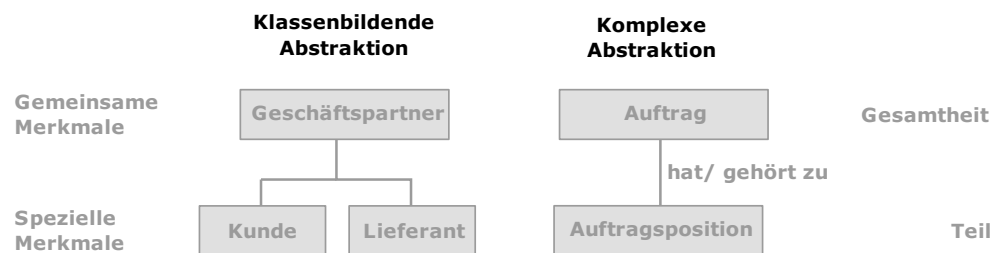


Abb.: Formen der Abstraktion

4.2 Prinzip der Strukturierung

Das Prinzip der Strukturierung hat sowohl für das Softwareprodukt als auch für den Entwicklungsprozess eine große Bedeutung. „Die Struktur (im allgemeinen Sinne) ist ein Gefüge, das aus Teilen besteht, die wechselseitig voneinander abhängen.“ [Bal98] . Strukturieren bedeutet, für ein komplexes System **eine reduzierte Darstellung** zu finden, die den Charakter des Ganzen mit seinen spezifischen Merkmalen wiedergibt.

Die Abbildung stellt beispielhaft eine einfache PNP-Struktur dar. Ein Transistor, eine Diode oder ein NPN sind real viel komplexer. Die Informationen können jedoch reduziert und strukturiert werden.

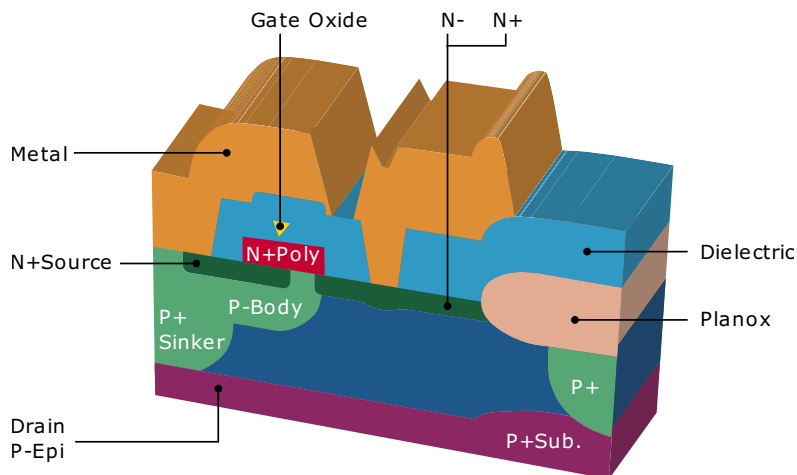


Abb.: PNP-Struktur

Struktur

Das Prinzip der Strukturierung hilft bei der Bewältigung von Komplexität. Es bietet:

- Ansätze zur Zerlegung in kleine, überschaubare Einheiten
- eine Basis für arbeitsteiliges Vorgehen
- eine schnelle Einarbeitung in einen Aufgabenkomplex

Die Anwendung des Prinzips der Strukturierung bringt weitere Vorteile [Bal98] :

- Erhöhung der Verständlichkeit (für Produkt und Prozess)
- Verbesserung der Wartbarkeit des Softwareproduktes
- Erleichterung der Einarbeitung in ein fremdes Softwareprodukt

Struktur ist die Art der Zusammensetzung der Elemente

Baum- und Netzstrukturen

Im Rahmen der Software-Entwicklung sind als Strukturierungsformen vor allem Baum- und Netzstrukturen verbreitet. Baumstrukturen dienen zur Darstellung hierarchischer Beziehungen, wie der Spezifikation von Prozessen, Daten und Programmen (Beispiele: Funktionshierarchie-Diagramme, Datenstruktur-Diagramme).

Netzstrukturen können zur Darstellung von Funktionsabläufen, Steuerflüssen und zur Darstellung von Beziehungen zwischen Daten und Funktionen bzw. zwischen Datenobjekten verwendet werden (Beispiele sind Informationsfluss- oder Entity-Relationship-Diagramme).

Gruppen / Clustern

Aber auch das Bilden von Gruppen / Clustern ist ein wichtiges Element der Softwaretechnik, wenn beispielsweise Funktionen oder GUI-Elemente gruppiert werden. Es zeigt sich daher, dass die Struktur eines Systems ganz allgemein die Art der Zusammensetzung der Elemente ist, wobei Elemente mit gleichen Eigenschaften zusammengehören oder sonst wie geometrisch nach Eigenschaften oder Fähigkeiten organisiert oder gruppiert werden können.

4.3 Prinzip der Hierarchisierung

Das Prinzip der Hierarchisierung ist ein **Spezialfall der Strukturierung**. Die Hierarchie bezeichnet eine Rangordnung, eine Über- und Unterordnung.

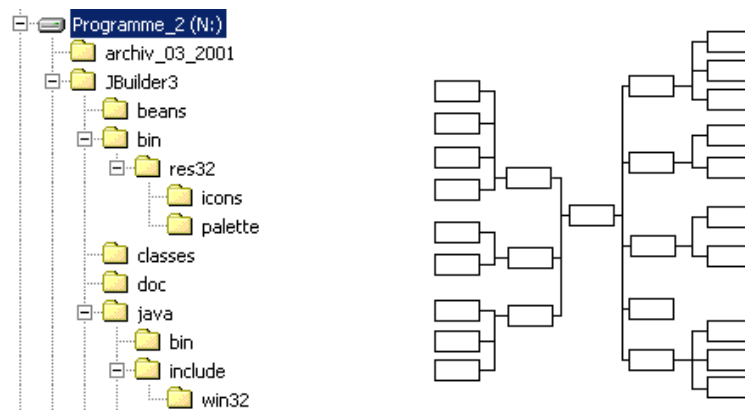


Abb.: Hierarchisierung

Ein System verfügt über eine Hierarchie, wenn seine Elemente nach einer **Rangordnung** zueinander in Beziehung stehen. Elemente der gleichen Rangordnung stehen auf derselben Stufe und bilden eine Ebene der Hierarchie. Rangordnungen können anhand von **Bedeutungen**, **Eigenschaften** oder zeitlichen Zusammenhängen der Elemente eines Systems festgelegt werden. Das Prinzip der Hierarchisierung ist ein wichtiges Prinzip zur Strukturierung von Softwareprodukten und von Software-Entwicklungsprozessen. Dem Prinzip der Hierarchisierung liegen beispielsweise **Baumstrukturen** zu Grunde.

4.4 Prinzip der Modularisierung

Aufgliederung

Das Prinzip der Modularisierung beinhaltet die **Aufgliederung** eines Softwaresystems in eine Menge übersichtlicher, eindeutig abgegrenzter und leicht austauschbarer **Bausteine** (Module). Ein Modul ist kontextunabhängig und **in sich abgeschlossen**.

Kontextunabhängigkeit beinhaltet, dass ein Modul von seiner Umgebung unabhängig entwickelbar, wartbar, prüfbar und verständlich ist.



Das Ergebnis der Modularisierung ist die Darstellung eines Softwaresystems als hierarchische Anordnung von Modulen. Das Prinzip der Modularisierung schließt ein, dass ein Modul festgelegte Schnittstellen zu anderen Modulen hat, d. h. der Datenaustausch erfolgt über Schnittstellen. Das Prinzip der Modularisierung wird u.a. durch folgende Konzepte und Methoden unterstützt:



- Komponenten- bzw. Subsystembildung beim objektorientierten Vorgehen
- hierarchische Informationsfluss-Diagramme

Modularität im Großen ist eng mit dem Prinzip der Abstraktion verknüpft, da die Modularisierung gleichzeitig das Bilden von Abstraktionsebenen erfordert.

Information hiding

Im engeren Zusammenhang zum Prinzip der Modularisierung steht das Geheimnisprinzip („information hiding“ nach **PARNAS**), welches Abstraktionsebenen erfordert 📖 [Par72] . Das Geheimnisprinzip bedeutet, dass für den Anwender bzw. Benutzer einer Systemkomponente bzw. eines Subsystems die Interna der Systemkomponente bzw. des Subsystems verborgen, d. h. nicht sichtbar sind. Die Anwendung des Geheimnisprinzips bedeutet für ein Modul, das kontextunabhängig sein soll und über eine definierte **Schnittstelle kommuniziert**, dass nur diese definierte Schnittstelle von außen **sichtbar** ist.

Aus der Anwendung des Prinzips der Modularisierung ergeben sich einige Vorteile:

- Höhere Änderungsfreundlichkeit (leichter Austausch und leichte Erweiterbarkeit)
- Verbesserung der Wartbarkeit
- Bessere Strukturierung
- Erleichterung der Standardisierung
- Verbesserung der Überprüfbarkeit

4.5 Prinzip der Standardisierung

Das Prinzip der Standardisierung ist erforderlich, um sowohl den Entwicklungsprozess als auch das Produkt zu vereinheitlichen. Das Prinzip der Standardisierung lässt sich durchsetzen über:



- Einhalten überbetrieblicher Standards
- Festlegen und Einhalten zweigspezifischer und betrieblicher Richtlinien
- Einsatz interner Standards

Vorteile

Die Verwirklichung des Prinzips der Standardisierung bringt folgende Vorteile:

- **Vereinheitlichung** des Entwicklungsprozesses und des Softwareproduktes
- Einheitliche Gestaltung der Dokumentation und Programme
- Erleichterung der **Einarbeitung und Wartung** von Softwareprodukten
- Erleichterung der **Anpassung** an veränderte Bedingungen im Basisprozess

Kosten

Ein wichtiger Aspekt der Standardisierung sind heutzutage die **Kosten**. In fast allen Branchen können große Projekte nur noch mit Hilfe von Standards kostengünstig realisiert werden. In der Kommunikationstechnik bezieht sich das beispielsweise auf **Protokolle**, aber auch im Consumer-Bereich zeigt es sich, beispielsweise beim Kampf um Videoformate und Datenträger (HD-DVD gegen Blu-ray). Die Design-Notation **UML** wurde auch durch den Wunsch der Industrie viel schneller zum Standard, damit unter anderem auch einheitliche UML-Werkzeuge entwickelt und verkauft werden können.

5 Überblick zum Studienmodul Softwaretechnik

Auf den folgenden Seiten wird ein Überblick über die Lerneinheiten des Studienmoduls Softwaretechnik I gegeben.

Das Studienmodul enthält 12 Lerneinheiten, die besonders ab der Lerneinheit Design Patterns nicht mehr in der vorgegebenen Reihenfolge gelehrt oder gelesen werden müssen.



5.1 Lerneinheiten 01 - 05

Lerneinheit SWT

Einführung in die Softwaretechnik

Was Sie gerade lesen, stimmt Sie auf die Thematik ein, klärt grundlegende Definitionen, stellt den Softwarelebenszyklus vor, stellt Prinzipien vor und macht Sie auf die kommenden Module neugierig.

Lerneinheit VOR

Vorgehensmodelle

In dieser Lerneinheit werden alle relevanten Vorgehensmodelle / Prozessmodelle vorgestellt und besprochen. Dies sind beispielsweise RUP, V-Modell, OEP und auch einige agile Modelle wie XP, Scrum oder Crystal Clear. Auf den RUP Prozess von Rational wird weniger umfassend eingegangen, weil dies Teil von Softwaretechnik II ist.



Lerneinheit REQ

Requirements Engineering

Hier werden die frühen Phasen von Softwareprojekten betrachtet. Bei der Anforderungserhebung und dem Anforderungsmanagement, handelt es sich hauptsächlich um den Menschen und seine Bedürfnisse und Wünsche, die sich auf die Softwaretechnik beziehen. Im Blickpunkt steht die Ergründung aller Randbedingungen - wie es sich aus Erfahrung zeigt ist dies in vielen Projekten eine schwierige Aufgabe.

Lerneinheit ANA

Analyse

In der Analysephase geht es darum, ein möglichst klares Bild des zukünftigen Systems zu entwickeln und dieses zu dokumentieren. Dies gilt von der Systemidee über Geschäftsprozesse bis hin zur konkreten Beschreibung der Anwendungsfälle.



Lerneinheit UML

Unified Modeling Language

Die UML ist mittlerweile der, von der OMG, dominierende Industriestandard für die Modellierung, mit anderen Worten das beschreibende Design von IT Systemen. Sie sieht für viele Elemente der Software-Entwicklung, wie Klassen, Module, Komponenten, Abläufe, Interaktionen etc., graphische Elemente vor und wird von sehr vielen Werkzeugen unterstützt.

 <http://www.jeckle.de/umltools.htm>

5.2 Lerneinheiten 06 - 08

Lerneinheit OOD

Objektorientiertes Design

Nachdem in der Analyse eine umfassende Vorstellung davon gewonnen worden ist, was zu tun ist, muss jetzt das System konkret entworfen werden. Das bedeutet: Es muss die Anwendungsarchitektur entwickelt, alle Komponenten definiert sowie alle Schnittstellen analysiert und festgelegt werden. Des Weiteren muss man sich bereits hier um Tests Gedanken machen, bis hin zur konkreten Definition der Klassen.



Es wird hier also quasi das Skelett der Anwendung entwickelt, um danach in der Implementierungsphase das Softwareprodukt passend mit „Fleisch“ füllen zu können.

Lerneinheit ARC

Objektorientierte Architekturen

Die vorigen Lerneinheiten haben die Grundlagen des Designs und damit auch einige Grundlagen der Architektur von IT-Systemen gelegt.

In dieser Lerneinheit werden komplexere Architekturen und deren Prinzipien vorgestellt, um einen Einblick über die Bedeutung und Anwendung von Software-Architektur zu bekommen.

Lerneinheit TST

Objektorientiertes Testen und Test-Driven Development

Ein wichtiges Gebiet der Softwaretechnik beschäftigt sich mit der Sicherstellung der **Qualität** von Software. Wie die Einführung gezeigt hat, können (auch schon kleinste) Fehler im Softwaresystem, ungeheure Kosten verursachen. Daher hat sich heute eine Vielfalt an pfiffigen Vorgehensweisen und Werkzeugen etabliert, die das Testen von Software sicherstellen. Und das beste ist, es macht Spaß!



Der Aufwand für Softwaretests ist minimiert worden und das Feedback für den Entwickler kann sehr individuell gestaltet werden. Es wird hier aber nicht nur Praxis mit den führenden **Testwerkzeugen** wie JUnit, TestNG vorgestellt, sondern auch etwas **Theorie** mit den Grundlagen und Schichten des Testens.

Weitere Themen sind **Test-Driven Development** und Test-Hilfen wie **Mock**-Objekte, die für fortgeschrittene Softwaretechnologen wichtig sind. Nicht ohne Grund ist dies eine der umfangreichsten Lerneinheiten.

5.3 Lerneinheiten 09 - 12

Lerneinheit REF

Refactoring, Refactoring to Patterns

Mittlerweile ist das **Umgestalten oder Ändern** von Software ein wichtiges Forschungsgebiet und ein wichtiger Wettbewerbsvorteil in der Software-Entwicklung geworden. In einer schnellen, sich agil ändernden Umgebung müssen Entwicklerteams sofort auf neue Anforderungen reagieren können und den Code oder das Design ihres Softwaresystems IDE gestützt anpassen können.

Es werden in dieser Lerneinheit daher die Prinzipien des Refactorings, der Kataloge und der Werkzeuge vorgestellt, mit denen der Code umgestellt werden kann. Refactoring geht aber noch viel weiter. Das Umstellen einer Architektur - wie z. B. die Anpassung oder Integration von Design Patterns in Code - erfordert viel mehr Erfahrung und wird hier ebenfalls nur angerissen.

Lerneinheit ANT

Buildmanagement

Heutzutage ist es unmöglich, alle Tätigkeiten wie kompilieren, kopieren, komprimieren, testen, codechecks, deployen, Prozesse starten, Reports erstellen und Code auschecken in einem großen Softwareprojekt per Hand durchzuführen. All diese Prozesse werden in einen Buildprozess integriert, der es per Knopfdruck ermöglichen soll, zu integrieren und **jederzeit** schnell eine **laufende Anwendung zu erstellen**.



Lerneinheit SVN

Versions- und Fehlermanagement

In dieser Lerneinheit wird das Versionsmanagement betrachtet. Was steckt dahinter? Was ist Checkout? Was sind Branches? Welche Werkzeuge gab es? Welche Werkzeuge sind heute State-of-the-Art? Und natürlich sind die beiden Platzhirsche **CVS** und **Subversion** Kern des Lernmoduls. Diese und andere kommerzielle Versionsmanagementsysteme sind Teil eines jeden Softwareentwicklungshauses, die alle Erfahrung mit diesen Systemen voraussetzen. Aber allein der sichere Boden unter den Füßen bei der Entwicklung bedeutet heutzutage schon ein viel schnelleres und aggressiveres Refactoring.

Abgeschlossen wird diese Lerneinheit durch einen Blick in Aufgaben von Fehlermanagementsystemen, von denen die wichtigsten kommerziellen und Open-Source Lösungen vorgestellt werden.

Lerneinheit MET

Software- und Architekturmetriken

Ein Teilaspekt der Softwaretechnik beschäftigt sich auch mit der **Softwaremessung**. So ermöglicht es bestimmte Maßzahlen für Aufwand und Qualität, gute Rückschlüsse auf den Code oder die noch zu leistende Arbeit zu liefern. Zu diesem Fachgebiet gehört auch Code-Coverage und konkrete Werkzeuge wie JDepend, McCabe.



Zusammenfassung

- Unter Software unterscheidet man aktive Daten wie ausführbare Programme und passive Daten.
 - Große Softwareprojekte können heutzutage Millionen an Kosten verursachen und schon wenige Fehler können ein komplettes System lahmlegen oder zerstören.
 - Es besteht ein signifikanter Zusammenhang zwischen Projektgröße und Projekterfolg.
 - Softwaretechnik kann in 5 Kategorien eingeteilt werden: Methoden, Werkzeuge, Maßsysteme, Standards und Erfahrungen.
 - Best Practices sind mehr oder weniger „objektive“ Erfahrungswerte die ein Projekt zum Erfolg führen.
 - Zu den schwer fassbaren Erfahrungen der Softwaretechnik gehören Punkte der allgemeinen Projekt- / Praxiserfahrung.
 - Der Softwarezyklus ermöglicht dem Softwaretechniker, die Elemente der Softwaretechnik besser einzuordnen und den entsprechenden Sprachgebrauch kennen zu lernen.
 - Die Sicherstellung einer geforderten Softwarequalität muss durch eine entwicklungsbegleitende Software-Qualitätssicherung erreicht werden.
 - Entwicklung und Management im Software-Entwicklungsprozess sind vielfach voneinander abhängig.
 - Zur Beschreibung des Software-Entwicklungsprozesses, mit Hilfe definierter Phase, werden in der Fachliteratur verschiedene Phasenmodelle als Software-Life-Cycle-Modelle verwendet.
 - Die Prinzipien des Software Engineerings sind allgemeingültig und abstrakt. Sie bilden die theoretischen Grundlagen, z. B. für die Methodenentwicklung.
 - Die Abstraktion ist eines der wichtigsten Prinzipien im Software Engineering. Es findet ein ständiges Wechselspiel zwischen Abstrahieren und Konkretisieren statt.
 - Strukturieren bedeutet für ein komplexes System eine reduzierte Darstellung zu finden, die den Charakter des Ganzen mit seinen spezifischen Merkmalen wiedergibt.
 - Ein System verfügt über eine Hierarchie, wenn seine Elemente nach einer Rangordnung zueinander in Beziehung stehen.
 - Das Ergebnis der Modularisierung ist die Darstellung eines Softwaresystems als hierarchische Anordnung von Modulen.
 - Das Prinzip der Standardisierung ist erforderlich, um sowohl den Entwicklungsprozess als auch das Produkt zu vereinheitlichen.
-

Wissensüberprüfung

Naturgemäß kann in einer Einführung noch nicht so viel Wissen oder Praxis abgefragt werden, aber ein paar kleine Checks helfen vielleicht, diese Lerneinheit besser zu verinnerlichen:



Formulieren

Übung SWT-01

Softwaretechnik allgemein

Was ist Softwaretechnik? Was wird da gelehrt? Worum geht es?

Was antworten Sie?

Bearbeitungszeit: 10 Minuten



Formulieren und Skizzieren

Übung SWT-02

Softwarelebenszyklus

Sie werden in einem Bewerbungsgespräch gebeten, die wesentlichen Teile des Softwarelebenszyklus an dem Whiteboard zu skizzieren und zu erläutern. Können Sie das?

Bearbeitungszeit: 15 Minuten



Formulieren

Übung SWT-03

Prinzipien

Geben Sie bitte zu jedem Prinzip der Softwaretechnik (Abstraktion, Strukturierung, Hierarchisierung, Modularisierung, Standardisierung) ein Beispiel an. Wo ist Ihnen das schon begegnet und wo könnte das in der Softwaretechnik angewendet oder wirksam werden?

Bearbeitungszeit: 10 Minuten



Recherchieren

Übung SWT-04

Recherche

Recherchieren Sie bitte im Internet über Softwaretechnik oder Softwareengineering. Welche Bereiche interessieren Sie ganz besonders?

Bearbeitungszeit: 30 Minuten



Formulieren

Übung SWT-05

Wissensfragen zur Lerneinheit

Versuchen Sie die Fragen schriftlich zu beantworten ohne noch einmal in der Lerneinheit nachzuschlagen.

1. An welche Best Practices erinnern Sie sich oder welche haben Sie verinnerlicht?
2. Warum ist es sinnvoll Softwarefehler früh zu erkennen?
3. Was geht in Softwareprojekten typischerweise schief?
4. Kennen Sie (Standardisierungs-)Organisationen, die sich mit Software beschäftigen?

Bearbeitungszeit: 15 Minuten



Zuordnung

Übung SWT-06

Fehlerquellen

Häufige Fehlerquellen in Softwareprojekten sind:

häufig



selten

32%

19%

18%

11%

6%

5%

5%

4%

Fehler im Code

Prozessverlauf

Datenbehandlungsfehler

Hardwarefehler

Fehler in den
Anforderungen

Fehler in der Systemlogik

Abarbeitungsfehler am
Rechner

Dokumentationsfehler

?

Test wiederholen

Test auswerten



Zuordnung

Übung SWT-07

Phaseneinteilung

Ordnen Sie die Phasen und Ergebnisse des Softwarelebenszyklus in der richtigen Reihenfolge an.

Phase	<input type="text"/>
Ergebnis	<input type="text"/>
↓	
Phase	<input type="text"/>
Ergebnis	<input type="text"/>
↓	
Phase	<input type="text"/>
Ergebnis	<input type="text"/>
↓	
Phase	<input type="text"/>
Ergebnis	<input type="text"/>
↓	
Phase	<input type="text"/>
Ergebnis	<input type="text"/>

Wartungs- und Pflegephase	installiertes Produkt
Definitionsphase	Situationsstudie
Analysephase	Produkt-Entwurf
Produkt- Definition	Programme
Abnahme- / Einführungsphase	gewartetes Produkt
Implementierungsphase	Entwurfsphase

? Test wiederholen Test auswerten