Inhaltsverzeichnis

1. Softwareengineering

1.1. Ziele

☑ Wichtigste Methoden und Verfahren des Software Engineering kennen lernen
☑ Werkzeuge kennen und anwenden können
☑ Einfache SW-Projekte planen und durchführen können
☑ Quellen:
- ☐ http://de.wikipedia.org/wiki/Softwareentwicklung
- ☐ http://www.stefan-baur.de/cs.se.lastenheft.gliederung.html
- ☐ http://pi.informatik.uni-siegen.de/gi/fg211/
- ☐ Helmut Balzert: *Lehrbuch der Software-Technik.* Bd.1. Software-Entwicklung. Spektrum Akademischer Verlag, Heidelberg 1996, 1998, 2001, ISBN 3-8274-0480-0.

1.2. Softwareengineering – Überblick

Die **Softwaretechnik** (engl. *software engineering*) beschäftigt sich mit der Herstellung von Software. Eine Definition von Helmut Balzert beschreibt das Gebiet als

☑ „**zielorientierte** und
☑ systematische **Anwendung** von **Prinzipien, Methoden** und **Werkzeugen** für die
☑ **ingenieurmäßige Entwicklung** von
☑ umfangreichen **Softwaresystemen**." (Lit.: Balzert, S.36)

Hinweis:

☑ Prinzipen … allg. Regeln (zb: Prinzip des Informationhiding)

☑ Methoden … dienen zur Umsetzung der Prinzipien (zb. OOP: Klassen, private, protected, ...)

☑ Die aktuellen Entwicklungen des Fachgebiets werden in der Dokumentation des „Software Engineering Body of Knowledge" (SWEBOK) beschrieben.

1.3. Phasen in der SW-Entwicklung - Teilschritte

Die Software wird Schritt für Schritt fertiggestellt. Diese Schritte werden Phasen genannt und sind während des gesamten Entwicklungsprozesses eng miteinander verzahnt.

1.3.1. Die Phasen - Ein erster Überblick

Trotz zahlreicher Varianten werden im wesentlichen immer die folgenden Phasen unterscheiden:

1. **Planung: Anforderungs- und Problemanalyse**
2. **Analyse: (Pflichtenheft)**
3. **Entwurf: System- und Komponentenentwurf**
4. **Programmierung: und Komponententest**
5. **Systemtest:**
6. **Betrieb: und Wartung**

Für jede der Phasen ist festgelegt, durch welches Ergebnis sie abgeschlossen wird.

Phase	Ziele	Tätigkeiten	Ergebnisse
Planung	☑ Aufgaben/Funktionen (**Was**) bestimmen ☑ Ressourcenbedarf geklärt	Ist-Zustand erheben Problembereich abgrenzen: geplantes System grob skizzieren Umfang und Wirtschaftlichkeit des Projektes abschätzen	☑ **DA-Antrag** ☑ **Projektauftrag** ☑ **grober Projektplan** ☑ **Projektkalkulation** ☑ **Lastenheft**
Analyse	**Vertrag** zwischen Auftraggeber und Hersteller	Pflichtenheft erstellen Projektplan festlegen	☑ **Pflichtenheft** ☑ **genauer Projektplan** ☑ **Benutzungshand-buch V1**
Entwurf	**Wie** die Anforderungen des Pflichtenheftes erfüllt werden.	Systemarchitektur entwerfen: ☑ Komponenten definieren ☑ Schnittstellen entwerfen ☑ Wechselwirkungen festlegen ☑ logische Datenmodell festlegen ☑ algorithmische Struktur entwerfen ☑ Algorithmen und Architektur validieren	☑ **UML-Diagramme** ☑ Beschreibung der Systemarchitektur ☑ **logisches Datenmodell** ☑ Algorithmen ☑ Dokumentation der Entwurfsentscheidungen
Programmierung	Entwurfsergebnisse sind auf einem Rechner ausführbar **Komponententest** bestanden	☑ Codierung in eine Programmiersprache ☑ logisches Datenmodell in physisches übertragen ☑ Komponenten testen	☑ Programm der einz. Module ☑ Protokolle der Modultests ☑ **physisch. Datenmodell** ☑ **Programmier-richtlinien** ☑ **Programm-dokumentation**
System-test	Wechselwirkung der Systemkomponenten unter realen Bedingungen erprobt	☑ Subsysteme integrieren ☑ Installation ☑ **Abnahmetest durch Benutzer** ☑ Leistungstests	☑ Systemtestprotokoll ☑ **auslieferbares Endprodukt**

	☑ **Implementation und -spezifikation stimmen überein**		
Betrieb und Wartung	störungsfreier Betrieb im Feld	☑ im Betrieb entdeckte Fehler beheben ☑ Systemänderungen und -erweiterungen vornehmen	neue Anforderungen

Hinweis:
Neben diesen Hauptphasen sind auch noch folg. Unterstützungsprozesse relevant:

☑ **Projektmanagement**
- Risikomanagement
- Projektplanung
- Projektverfolgung und -steuerung
- Management von Lieferantenvereinbarungen

☑ **Konfigurationsmanagement**
- Versionsverwaltung
- Änderungsmanagement / Veränderungsmanagement
- Release Management
- Application Management (ITIL)

☑ **Dokumentation**
- Software-Dokumentationswerkzeug
- Systemdokumentation (Weiterentwicklung und Fehlerbehebung)
- Betriebsdokumentation (Betreiber/Service)
- Bedienungsanleitung (Anwender)
- Geschäftsprozesse (Konzeptionierung der Weiterentwicklung)
- Verfahrensdokumentation (Beschreibung rechtlich relevanter Softwareprozesse)

Die oben genannten Teilschritte der Softwareentwicklung werden nicht zwangsläufig bei jedem Projekt komplett durchlaufen. Vielmehr werden einzelne Prozesse spezifisch für die jeweilige Anforderung gewählt. Dies ist aus Sicht der Kosten- und Verwaltungsreduzierung notwendig.

1.4. Das Wasserfallmodell

Eines der ältesten Modelle ist das Wasserfallmodell, das eine starre Abfolge der einzelnen Phasen annimmt und heute kaum mehr Verwendung findet.

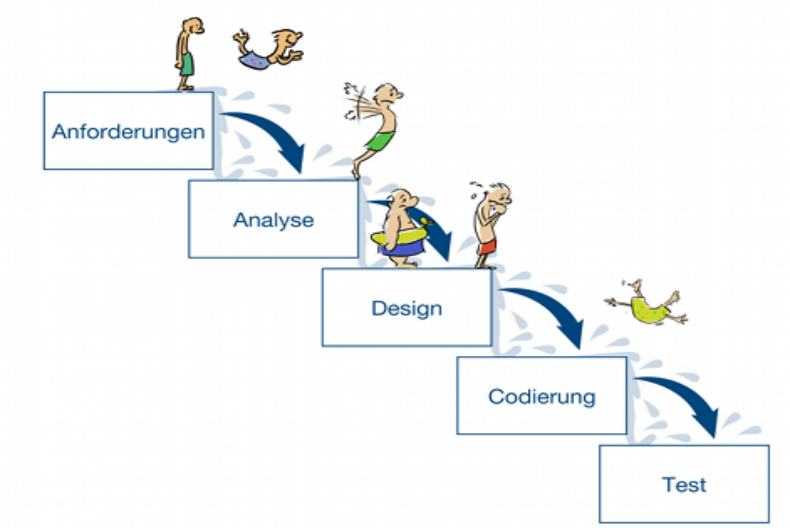

Abbildung 1: Wasserfallmodell (www.scrum-kompakt.de, Okt2015)

1.5. Das V-Modell

Eine Erweiterung, die eine stärkere **Berücksichtigung der Qualitätssicherung** während des gesamten Entwicklungsprozesses vorsieht, ist das sogenannte V-Modell.

Hinweis:

Das V-Modell diente auch als Grundlage für ein Vorgehensmodell, dass **bei der Entwicklung von IT-Systemen** für die Bundeswehr und für Behörden verwendet wird, und inzwischen auch in der Industrie Anwendung findet [Balzert 98, S.103].

Das ursprüngliche V-Modell erweitert das klassische Phasenmodell um die **parallel zu den einzelnen Phasen** durchzuführende Entwicklung von **Testfällen** auf einer semantisch entsprechenden Ebene.

So werden in der Phase der

☑ **Analyse (Anforderungsdefinition) gleichzeitig Testfälle für den Abnahmetest** erstellt (Anwendungszenarien),

☑ während der **Entwurfsphase** werden Testfälle für der **System- und Integrationstest** definiert, und

☑ während der **Implementationsphase** werden **Testfälle für den Modultest** erstellt.

Die folgende Abbildung zeigt eine vereinfachte schematische Darstellung des V-Modells:

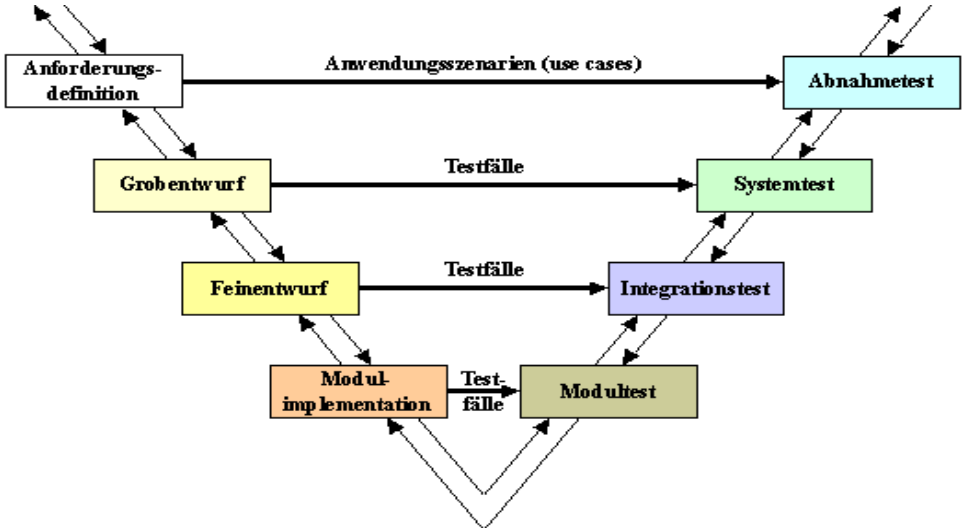

Abbildung 2: V-Modell (winf-wiki.fhslabs.ch Okt2015)

1.6. Prototypenbasierte Modelle

http://de.wikipedia.org/wiki/Prototyping_%28Softwareentwicklung%29

Die prototypenbasierten Prozeßmodelle unterstützen auf systematische Weise die **frühzeitige Erstellung von ablauffähigen Modellen** des Zielprodukts.

Auf diese Weise können die Entwickler Ihre Umsetzungen der gestellten

☑ **Anforderungen** in Kooperation mit den späteren Benutzern **frühzeitig überprüfen** und
☑ mit dem **Entwurf** des Produkts **experimentieren**.

In einem sich wiederholenden Prozess von

1. **Anforderung**sbestimmung,
2. **Umsetzung** im Prototyp und
3. **Evaluation** des Prototypen mit den Benutzern

entsteht aus dem Prototyp das Zielprodukt.

Die folgende Abbildung zeigt den Ablauf dieser Prozeßmodelle:

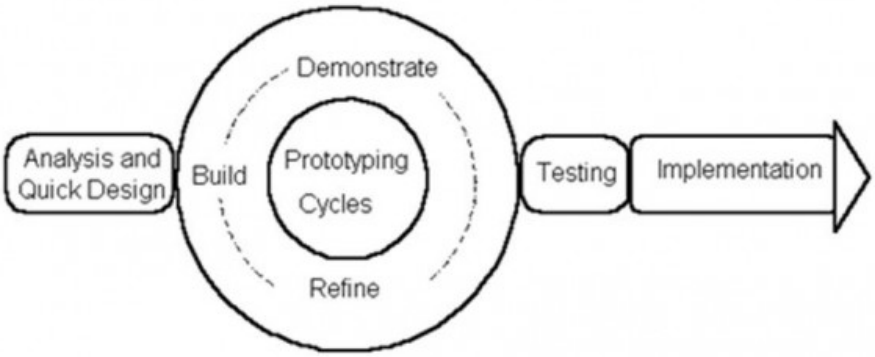

Abbildung 3: http://www.leanready.com/2014/05/validate-your-product-faster-with-rapid-prototyping/

Vorteile

☑ Die **Anforderungen** der Anwender können laufend **präzisiert** und verifiziert werden. Damit sinkt das Risiko einer Fehlentwicklung.
☑ Der Fertigstellungstermin ist besser verifizierbar.
☑ Die **Qualitätssicherung** kann frühzeitig eingebunden werden.

Nachteile

☑ Prototyping verführt oft dazu, Anforderungen weder korrekt zu erheben noch sauber zu **dokumentieren**. Der Entwicklungsprozess kann sich dadurch erheblich verlangsamen.

1.6.1. Exploratives Prototyping

☑ Oft auch **Oberflächenprototyp** genannt.
☑ Frühzeitiges **Feedback** der **zukünftigen Nutzer**
☑ Konzentration auf die Funktionalitäten des Systems
ausgehend vom Entwurf der Oberfläche bzw. der Menüeintrage, ermittelt man die Funktionalität des Produktes.
☑ Ziel ist es **nachzuweisen, dass die Anforderungen oder Ideen tauglich sind.**

1.6.2. Evolutionäres Prototyping

☑ zur evolutionären Softwareentwicklung
☑ **schrittweise Erweiterung** der Funktionalität
☑ Prototyp wird stets lauffähig gehalten und bis zur Produktreife weiterentwickelt.

1.6.3. Experimentelles Prototyping

☑ zu Forschungszwecken bzw. der **Suche nach Möglichkeiten zur Realisierung**
☑ mit einem experimentellen Prototyp wird eine **sehr umfangreiche Problemanalyse und Systemspezifikation(Pflichtenheft)** durchgeführt
☑ gewonnene Erkenntnisse können anschließend in einem richtigen Produkt verwertet werden

1.7. +Das RUP Vorgehensmodell: Rational Unified Process

Der Rational Unified Process (RUP) ist ein **objektorientiertes Vorgehensmodell** zur Softwareentwicklung.

Der RUP **benutzt die Unified Modeling Language(UML)** als Notationssprache und basiert auf mehreren Verfahren:

☑ 1. Erstellung von **Anwendungsfällen** (Use Cases) erstellen

☑ 2. Erstellung einer **System-Architektur** (**Klassendiagramme**)

☑ 3. inkrementelles und iteratives **Entwickeln**
- ☐ (**Sequenzdiagramme, Interaktionsdiagramme, Zustandsübergangsdiagramme**, ...)

Anmerkung: UML-Diagramme werden in einer eigenen Lerneinheit bearbeitet.

Die erste Version des RUP aus dem Jahre 1999 führte die Vorschläge der drei Begründer (Booch, Jacobson, Raumbough) für eine einheitliche Modellierungsmethode zusammen.

Hinweis:

RUP (Rational Unified Process) ist eine Vorgehensweise beim Softwareentwicklungsprozess mit **eher schwergewichtiger** Methodologie, vielen formalen Definitionen und Dokumenten, iterativ, architekturzentriert, Use-Case-getrieben, wohldefiniert und sehr strukturiert.

Zu jedem Zeitpunkt bietet RUP Planungshilfen, Leitlinien, Checklisten und Best Practices.

Die konsequente und komplette Nutzung von RUP macht erst bei Teams mit über 10 Personen Sinn. Es sind über 30 Rollen für über 130 Aktivitäten vorgesehen und es werden über 100 verschiedene Artefakttypen (Arbeitsergebnistypen) vorgeschlagen, die erzeugt, dokumentiert und verwaltet werden müssen.

1.7.1. Statische Aspekte

Der RUP legt grundlegende Arbeitsschritte fest:

Arbeitsschritte

1. Geschäftsprozessmodellierung (englisch *Business Modeling*)
2. Anforderungsanalyse (englisch *Requirements*)

3. Analyse & Design (englisch *Analysis & Design*)
4. Implementierung (englisch *Implementation*)
5. Test (englisch *Test*)
6. Auslieferung (englisch *Deployment*)

1.7.2. Dynamische Aspekte

In jedem der obigen Arbeitsschritte werden folgende 4 Phasen mehr oder weniger intensiv angewendet:

1. **Konzept**ionsphase (englisch *Inception*)
2. **Entwurf**sphase (englisch *Elaboration*)
3. **Konstruktion**sphase (englisch *Construction*)
4. **Übergabe**phase (englisch *Transition*)

Jede dieser Phasen werden pro Arbeitsschritt mehrfach (Iteration) durchlaufen.

RUP unterstützt das **iterative** Paradigma:

Vorteile
☑ +**Risiken** können früher erkannt werden
☑ +wechselnde **Anforderungen** können besser berücksichtigt werden
☑ +inkrementelle Auslieferung wird erleichtert

Nachteile
☑ –Mehrarbeit
☑ –komplexeres Projektmanagement
☑ –schwerer messbar

Weiteres siehe http://www.ibm.com/software/awdtools/rup.

Das absolute Einhalten der Iterationen (Rückkopplungen in den jeweiligen Phasen, Validierungsphasen, ...) kann zu Nachteilen führen. Aus diesem Grund wurden weitere Methoden entwickelt, die noch sehr jung sind. Dazu gehören die agile od. Adaptiven SW-Prozesse.

1.8. Agile und adaptive Vorgehensmodelle

Unter einem **agilen/adaptiven Vorgehensmodell** versteht man eine Weiterentwicklung des iterativen Paradigmas, bei der die **Planung der Iterationen dynamisch** erfolgt.

Charakteristikum:

☑ **kontinuierliche Anpassung an Änderungen**

1.8.1. EXE-KOOP-UPDATE-Prinzipien:

☑ **EXE: Ausführbare SW**	VOR	vollständiger **Dokumentation**
☑ **KOOP: Zusammenarbeit mit Kunden**	VOR	**Vertrags**verhandlung
☑ **UPDATE: Berücksichtigung von Änderungen**	VOR	**Beharren auf Plan**

Bekannte Vorgehensweisen:

☑ XP: Extreme Programmierung
  ☐ **Einfachheit**, **Kommunikation**, Mut, Feedback
☑ Scrum, Kanban:
  ☐ **Visualisierung des Projektverlaufes** (s. scrum.pdf)

1.8.2. Extreme Programming: Einfachheit und Kommunikation

http://de.wikipedia.org/wiki/Extreme_Programming

Auszug aus: http://www.frankwestphal.de/ExtremeProgramming.html

Extreme Programming (XP) ist eine

☑ **agile(EXE-KOOP-UPDATE) Vorgehensweise zur SW-Entwicklung**
☑ **für kleine Teams**.

Der Prozess ermöglicht, **langlebige Software zu erstellen** und während der Entwicklung auf vage und sich **rasch ändernde Anforderungen zu reagieren**.

XP-Projekte schaffen **ab Tag 1 Geschäftswert** für den Kunden und lassen sich fortlaufend und außergewöhnlich stark durch den Kunden steuern.

Im Kern beruht XP auf den Werten**, Einfachheit, Kommunikation, Feedback, Mut**, Lernen, Qualität und Respekt.

XP **erfordert Disziplin.**

1.8.3. Techniken für ein XP-TEAM

☑ XP-PERSONEN
  ☑ **2 bis etwa 12 PROGRAMMIERER**,
  ☑ einem **AUFTRAGGEBER/Kunde** oder mehreren Ansprechpartnern auf Kundenseite und
  ☑ **MANAGER**.

☑ XP-UMGEBUNG/Ablauf

☑ **Offene Arbeitsumgebung**
Das Team arbeitet zusammen in einem größeren Raum oder eng aneinander grenzenden Räumen. Typischerweise ist der "**Kriegsraum**" mit **Wandtafeln** und unzähligen **Flipcharts** ausgestattet. Die Arbeitstische stehen meist dicht beieinander im Kreis mit den Monitoren nach außen gerichtet und sind so gestaltet, dass **zwei Programmierer zusammen bequem an einem Computer** arbeiten können.

☑ **Kurze Iterationen**
Die Entwicklung erfolgt in **Perioden von ein bis drei Wochen**. Am Ende jeder Iteration steht ein funktionsfähiges, getestetes System mit neuer, für den Kunden wertvoller Funktionalität.

☑ **Gemeinsame Sprache**
Das Team entwickelt in seiner Arbeit ein gemeinsames Vokabular, um über die Arbeitsweisen und das zu erstellende System diskutieren zu können. Die Kommunikation im Team erfolgt stets **offen und ehrlich**.

☑ **Reviews/Rückblick:**
Jede Iteration endet damit, in einem Rückblick über die eigenen Arbeitsweisen kritisch zu reflektieren und im Team zu diskutieren,

☐ was gut lief und
☐ was in Zukunft anders angegangen werden muss:
(wer kann womit helfen?)
☐ Zeitplan/Fertigstellungsgrad
(was muss noch alles gemacht werden?)

Typischerweise werden aus den Dingen, die während dieser **Team-Reviews** zur Oberfläche kommen, **Regeln** generiert, vom Team akzeptiert, **auf Poster geschrieben** und im Projektraum zur Erinnerung an die Wand geheftet.

Ein- oder zweimal jährlich macht das Team für zwei Tage einen gemeinsamen Ausflug, um in einem Offsite-Meeting formal vor- und zurückzublicken.

☑ **Tägliches Standup-Meeting**
Der Tag beginnt mit einem Meeting, das **im Stehen** gehalten wird, damit es kurz und lebendig bleibt.

Jedes Teammitglied berichtet reihum, an welcher Aufgabe es **gestern** gearbeitet hat und was es **heute** machen wird. **Probleme werden genannt aber nicht gelöst. Die meisten Teams treffen sich vor der Wandtafel ihrer Iterationsplanung**.

1.8.4. Techniken für die Kunden

Erstellung von User-Story-Karten und Akzeptanztests.

☑ **Anforderungsdefinition im Dialog: Kunde ↔ Programmierer**
Das für die anstehenden Programmieraufgaben nötige Verständnis der Anforderungen wird

fortlaufend in der Konversation mit den Kunden geprüft und vertieft. In kurzen Designsessions wird unter Umständen auf eine der Wandtafeln ein wenig UML gemalt. Während der gesamten Entwicklung dienen die **Kunden als direkte Ansprechpartner** zur Bewältigung fachlicher Fragen. Die verbleibende Zeit verbringen die Kunden mit dem Schreiben und Ergründen neuer Benutzergeschichten und Akzeptanztests.

☑ **Benutzergeschichten (user-story-karte): Kunde**

Die Kunden halten ihre **Anforderungen in Form einfacher Geschichten(Story)** auf gewöhnlichen Karteikarten fest. Jeder geschriebenen Story-Karte kommt das Versprechen nach, den genauen Funktionsumfang zum rechten Zeitpunkt im Dialog mit den Programmierern zu verfeinern und zu verhandeln.

Zudem werden pro story Akzeptanztests entwickelt. (siehe später Lerneinheit: Testen)

☑ **+Iterationsplanung: Kunde ↔ Programmierer**

Jede Iteration beginnt mit einem **Planungsmeeting**, in dem das

☐ Kundenteam seine **Geschichten** erzählt und mit dem Programmierteam diskutiert.

☐ Die Programmierer **schätzen** den **Aufwand** grob ab, den sie zur Entwicklung jeder einzelnen Geschichte benötigen werden.

☐ Die **Programmierer zerlegen** die geplanten Geschichten am Flipchart in technische Aufgaben, **übernehmen Verantwortung** für einzelne Aufgaben und schätzen deren Aufwände vergleichend zu früher erledigten Aufgaben.

☑ **Akzeptanztests**

Die Kunden spezifizieren während der Iteration funktionale **Abnahmekriterien**. Typischerweise entwickeln die Programmierer ein kleines Werkzeug, um diese Tests zu kodieren und automatisch auszuführen. Spätestens zum Ende der Iteration müssen die Tests erfüllt sein, um die gewünschte Funktion des Systems zu sichern.

☑ **Kurze Releasezyklen**

Nach **ein bis 3 Monaten** wird das System an die wirklichen Endanwender ausgeliefert, damit das Kundenteam wichtiges Feedback für die Weiterentwicklung erhält.

1.8.5. Techniken für die Programmierer

☑ **Programmieren in Paaren**

Die Programmierer arbeiten stets zu zweit am Code und diskutieren während der Entwicklung intensiv über Entwurfsalternativen. Sie wechseln sich **minütlich an der Tastatur ab** und **rotieren stündlich ihre Programmierpartner**. Das Ergebnis ist eine höhere Codequalität, grössere Produktivität und bessere Wissensverbreitung.

☑ **Gemeinsame Verantwortlichkeit**

Der gesamte Code gehört dem Team. Jedes Paar soll jede **Möglichkeit zur Codeverbesserung** jederzeit wahrnehmen. Das ist kein Recht sondern eine **Pflicht**.

☑ **Erst Testen**
Gewöhnlich wird jede Zeile Code durch einen Testfall motiviert, der zunächst fehlschlägt. Die **Unit Tests werden gesammelt, gepflegt und nach jedem Kompilieren ausgeführt.**

☑ **Design für jetzt**
Jeder Testfall wird auf die einfachst denkbare Weise erfüllt. Es wird keine unnötig komplexe Funktionalität programmiert, die momentan nicht gefordert ist.

☑ **Refactoring**
Das Design des Systems wird fortlaufend in kleinen, funktionserhaltenden Schritten verbessert. Finden zwei Programmierer Codeteile, die schwer verständlich sind oder unnötig kompliziert erscheinen, verbessern und vereinfachen sie den Code. Sie tun dies in disziplinierter Weise und führen **nach jedem Refactoring-Schritt die Unit Tests aus**, um keine bestehende Funktion zu zerstören.

☑ **Fortlaufende Integration**
Das System wird mehrmals **täglich** durch einen **automatisierten Build-Prozess** neu gebaut. Der entwickelte Code wird in kleinen Inkrementen und spätestens **am Ende des Tages in die Versionsverwaltung eingecheckt** und ins bestehende System integriert.

Die **Unit Tests** müssen zur erfolgreichen Integration zu **100% laufen**.

1.8.6. Techniken für das Management

☑ **Akzeptierte Verantwortung/ richtiges Delegieren**
Das Management
☐ **schreibt** einem XP-Team **niemals vor**, was es zu tun hat.
☐ Stattdessen **zeigt** der Manager lediglich **Probleme auf** und
☐ **läßt** die Kunden und Programmierer selbst **entscheiden**, was zu tun gilt.

☑ **Information durch Metriken**
Eine der Hauptaufgaben des Managements ist es, dem Team den Spiegel vorzuhalten und zu zeigen, wo es steht. Dazu gehört unter anderem das Erstellen einfacher Metriken, die den **Fortschritt** des Teams oder zu lösende Probleme **aufzeigen**. Es gehört auch dazu, den Teammitgliedern regelmässig in die Augen zu schauen und **herauszufinden, wo Hilfe** von Nöten ist.

☑ **Ausdauerndes Tempo**
Softwareprojekte gleichen mehr einem Marathon als einem Sprint. Viele Teams werden immer langsamer bei dem Versuch, schneller zu entwickeln. **Überstunden sind keine Lösung** für zuviel Arbeit.
☐ Wenn Refactorings und Akzeptanztests aufgeschoben werden, muss der Manager dem Team stärker den Rücken freihalten.
☐ Wenn Teammitglieder müde und zerschlagen sind, muß der Manager sie nach Hause schicken.

1.8.7. +Podcast zu XP: Chaos Radio

http://chaosradio.ccc.de/cre028.html

Extreme Programming (XP) ist eine seit einigen Jahren immer populärer werdende Methode zur Entwicklung von Software **in kleineren Teams** (3-6 Personen, Projektdauer. 6-12 Monate). Die teilweise radikalen Änderungen im Vergleich zur "traditionellen" Vorgehensweisen erfordern umfangreiches Umdenken in technischen und sozialen Prozessen, bieten aber die Möglichkeit der Beherrschung zuvor schwer zu bändigender Dynamiken.

Ängste, mangelnde Offenheit und Kommunikation sowohl auf Seiten des Auftraggebers als auch der Entwickler sind häufig bereits der Anfang vom Ende jeder erfolgreichen Softwareentwicklung. Extreme Programming begegnet diesen Problemen durch kooperative Entwicklungsmodelle (**Pair Programming**), **iterative Verfeinerungen der Aufgabenstellung** und Konzentration auf **schnelle Releasezyklen** und Testbarkeit von Systemen.

Pavel berichtet aus seinen jahrelangen und mehrheitlich positiven Erfahrungen in der konkreten Anwendung von Extreme Programming im Unternehmen und erläutert, welche Schritte nötig waren, um diese Umstellung zu einem Erfolg zu führen, welche langfristigen Effekte das hatte.

1.9. Phasenmodelle – Eine Zusammenfassung

Einen ausführlichen Überblick bietet:

☑ http://de.wikipedia.org/wiki/Liste_von_Softwareentwicklungsprozessen

1.10. Dokumente

1.10.1. Lastenheft

http://de.wikipedia.org/wiki/Lastenheft

Ein **Lastenheft** beschreibt die unmittelbaren Anforderungen **durch den Besteller** eines Produktes.

Gemäß DIN 69905 (Begriffe der Projektabwicklung) beschreibt das Lastenheft die „vom Auftraggeber festgelegte Gesamtheit der Forderungen an die Lieferungen und Leistungen eines Auftragnehmers innerhalb eines Auftrages".

Das Lastenheft beschreibt in der Regel also, **was und wofür** etwas gemacht werden soll (Fachkonzept). Die Adressaten des Lastenhefts sind der (externe oder firmeninterne) Auftraggeber sowie die Auftragnehmer.

In der Softwaretechnik ist das Lastenheft das **Ergebnis der Planungsphase** und wird in der Regel einvernehmlich von den Bestellern und den Entwicklern als **Vorstufe des**

Pflichtenhefts überarbeitet.

Um ein Lastenheft übersichtlich zu halten, wird es vorzugsweise in knapp orientierendem Text gefasst und mit Detaillierungen beispielsweise in tabellarischer Form, mit Zeichnungen oder Grafiken ergänzt. Es gibt dazu auch formalisierende Ansätze, wie die Beschreibungssprache UML.

1.10.2. Lastenheft: Gliederung

Ein Lastenheft lässt sich auf verschiedene Weise gliedern. Folgende Angaben sollten berücksichtigt werden:

1. Ausgangssituation und Zielsetzung
2. Produkteinsatz
3. Produktübersicht
4. Funktionale Anforderungen
5. Nicht funktionale Anforderungen
    - Benutzbarkeit
    - Zuverlässigkeit
    - Effizienz
    - Änderbarkeit
    - Übertragbarkeit
    - Wartbarkeit
6. Risikoakzeptanz
7. Skizze des Entwicklungszyklus und der Systemarchitektur oder auch ein Struktogramm
8. Lieferumfang
9. Abnahmekriterien

1.10.3. Lastenheft: Beispiel

http://www.stefan-baur.de/cs.se.lastenheft.html

1.10.4. Pflichtenheft

Das **Pflichtenheft** (auch **Fachspezifikation**, **fachliche Spezifikation**, **Fachfeinkonzept**, **Sollkonzept**, **Funktionelle Spezifikation**, oder **Feature Specification**)

☑ beschreibt die unmittelbaren Anforderungen durch den Besteller
☑ in der Interpretation des Herstellers für sein Produkt.

Das Pflichtenheft ist die **vertraglich bindende**, detaillierte Beschreibung eines zu erstellenden Werkes, zum Beispiel des Aufbaus einer technischen Anlage, der Konstruktion eines Werkzeugs oder auch der Erstellung eines Computerprogramms. Die dazu erforderliche Arbeit liegt allein in der Verantwortung des Herstellers oder Auftragnehmers

Das **Pflichtenheft wird vom Auftragnehmer (Entwicklungsabteilung/-firma) formuliert und auf dessen Wunsch vom Auftraggeber bestätigt**.

Idealerweise sollten erst nach dieser Bestätigung die eigentlichen Entwicklungs-/Implementierungsarbeiten beginnen.

Im Gegensatz zum technischen Design (auch technische Spezifikation – Wie wird es umgesetzt?) beschreibt das Pflichtenheft die geplante technische Lösung – in unserem Beispiel die Software – als Black Box (**Was wird umgesetzt**?). Entsprechend enthält es in der Regel nicht die betriebliche Lösung der Aufgabenstellungen des Auftraggebers. Schon gar nicht beschreibt es die (hier beim Softwarebeispiel) Implementierungsprobleme, sondern allenfalls die Schnittstellen, deren sorgfältige Beschreibung solche Probleme vermeiden soll.

Es ist bewährte Praxis, bei der Erstellung eines Pflichtenheftes das **Ein- und Ausschlussprinzip** zu verwenden, d. h., konkrete Fälle explizit ein- oder auszuschließen.

Bei Lieferung der Software wird formell eine Abnahme vollzogen, die die Ausführung des Werkvertrages oder auch des Kaufvertrages beschließt. Diese Abnahme wird häufig über einen Akzeptanztest ausgeführt, der feststellt, ob die Software die Forderungen des Pflichtenheftes in dem Verständnis des Bestellers erfüllt.

1.10.5. Pflichtenheft: Gliederung

Ein Pflichtenheft sollte wie folgt gegliedert sein:

1. Zielbestimmung
    1. Musskriterien: für das Produkt unabdingbare Leistungen, die in jedem Fall erfüllt werden müssen
    2. Sollkriterien: die Erfüllung dieser Kriterien wird angestrebt
    3. Kannkriterien: die Erfüllung ist nicht unbedingt notwendig, sollten nur angestrebt werden, wenn noch ausreichend Kapazitäten vorhanden sind.
    4. Abgrenzungskriterien: diese Kriterien sollen bewusst nicht erreicht werden
2. Produkteinsatz
    1. Anwendungsbereiche
    2. Zielgruppen
    3. Betriebsbedingungen: physikalische Umgebung des Systems, tägliche Betriebszeit, ständige Beobachtung des Systems durch Bediener oder unbeaufsichtigter Betrieb
3. Produktübersicht: kurze Übersicht über das Produkt
4. Produktfunktionen: genaue und detaillierte Beschreibung der einzelnen Produktfunktionen
5. Produktdaten: langfristig zu speichernde Daten aus Benutzersicht
6. Produktleistungen: Anforderungen bezüglich Zeit und Genauigkeit
7. Qualitätsanforderungen
8. Benutzungsoberfläche: grundlegende Anforderungen, Zugriffsrechte
9. Nichtfunktionale Anforderungen: einzuhaltende Gesetze und Normen, Sicherheitsanforderungen, Plattformabhängigkeiten
10. Technische Produktumgebung
    1. Software: für Server und Client, falls vorhanden

    2. Hardware: für Server und Client getrennt
    3. Orgware: organisatorische Rahmenbedingungen
    4. Produktschnittstellen
11. Spezielle Anforderungen an die Entwicklungsumgebung
    1. Software
    2. Hardware
    3. Orgware
    4. Entwicklungsschnittstellen
12. Gliederung in Teilprodukte
13. Globale Testszenarien, Abnahmekriterien
14. Glossar: In dem eventuelle Fachausdrücke für Laien erläutert werden.

1.10.6. Pflichtenheft: Beispiel

http://www.stefan-baur.de/cs.se.pflichtenheft.beispiel.html

1.10.7. RDP: Diplomarbeits-Antrag

https://info.htl-salzburg.ac.at/Lehre/elektronik/Diplomarbeiten/

1.10.8. RDP: Zitierrichtlinien

https://info.htl-salzburg.ac.at/Lehre/elektronik/Diplomarbeiten/

https://www.zotero.org/

http://www.citavi.com/

1.10.9. RDP: Projektmanagement

http://www.projekthandbuch.de/it_model_qm_review.htm

1.10.10. RDP: Aufbau Muster_Diplomarbeit

https://info.htl-salzburg.ac.at/Lehre/elektronik/Diplomarbeiten/

1.10.11. RDP: Tagebuch

https://info.htl-salzburg.ac.at/Lehre/elektronik/Diplomarbeiten/

Datum,von,bis, Wer, Tätigkeiten, Bemerkungen

1.10.12. RDP: Besprechungsprotokolle

https://info.htl-salzburg.ac.at/Lehre/elektronik/Diplomarbeiten/

Datum, von,bis,
Ort
Thema
Teilnehmer
Agenda
Beschlüsse

1.10.13. RDP: Diplomarbeit_Beurteilungsraster

abteilungsvorstand\Diplomarbeit_Beurteilungsraster.pdf

1.11. RDP: Dokumentation

www.htl-innovativ.at

1.12. Werkzeuge

1.12.1. Test, Verifikation und Validierung

JUNIT,… (siehe: SENG/testen)
http://www.eclipse.org/tptp/

1.12.2. UML

http://www.highscore.de/uml/
siehe Lerneinheit: SENG/uml

1.12.3. Versionsverwaltung

https://git-scm.com/doc

1.12.4. Programmdokumentation: Ungarische Notation, Doxygen

Zitat www.wikipedia.de:

> Bei der ungarischen Notation handelt es sich um eine von Programmierern verwendete Konvention zur Benennung von Variablen. Die ungarische Notation wurde durch den Ungarn Charles Simonyi entwickelt und bekam aufgrund seiner Nationalität ihren Namen.
> Kern der ungarischen Notation ist es, den Datentyp einer Variable im Variablennamen zu verdeutlichen. Hierzu werden verschiedene Präfixe definiert, die dem Variablennamen vorangestellt werden.

Präfixe für Variablensichtbarkeiten:

Präfix	Sichtbarkeit	Beispiel
m_	Member-Variable	m_szLastName
g_	globale Variable	g_pVertexBuffer
s_	statische Variable	s_iInstanceCount

Präfixe für Datentypen:

Präfix	Datentyp	Beispiel
n oder i	Integer	nSize
l	Long	lBigSize
u	unsigned (in Kombination mit n,i oder l)	uiPositiveNumber
f	Float	fPrecision
d	Double	dMorePrecision
c	Character	cChar
b	Boolean	bBusy
sz	null-terminierter String	szLastName
p	Zeiger	pMemory
a	Array	aCounter
r	Referenz	rReferenz
p	Pointer	pVertexBuffer
v	Vektor	vCross
m	Matrix	mProjection

Anmerkungen:

- Die einzelnen Präfixe lassen sich auch kombinieren. So definiert paszTabelle einen Zeiger auf ein Array null-terminierter Strings.
- Da enum's in C zu int's konvertiert werden können, ist es sinnvoll entsprechende Variablen auch so zu deklarieren.
  Beispiel:

  enum FARBE {

```
  ROT,
  GRUEN,
  BLAU,
};
FARBE iFarbe = ROT;
```

- Die Variablen-Konventionen gelten natürlich nicht für innermethodische Deklarationen, aber auch hier sollte man vielleicht nicht nur r,s,w,u und Konsorten als Variablennamen verwenden, da dadurch die Verständlichkeit deutlich sinkt.

Zusätzliche Konventionen:

- Header-Dateien haben die Dateiendung .h, Quelldateien die Endung .cpp
- Klassennamen werden groß geschrieben
  `class MemoryManager {...}`
- Variablen werden klein geschrieben
  `MemoryManager* pmm;`
- Konstanten werden konstant groß geschrieben
  `static const int MEM_SIZE=100`
- Verzeichnisse werden klein geschrieben
  `./grafik/MemoryManager.hpp`
- Methoden werden ausschließlich in den Quelldateien implementiert
  `MemoryManager.h:`

```
#ifndef MEMORY_MANAGER
#define MEMORY_MANAGER

class MemoryManager {
  static const int MEM_SIZE;
  int m_nSize;

  MemoryManager();
  virtual ~MemoryManager();
  int free();
}
#endif // MEMORY_MANAGER
```

`MemoryManager.cpp`

```
#include "MemoryManager.h:"

int MemoryManager::MEM_SIZE=100;

MemoryManager::MemoryManager() {...}
MemoryManager::~MemoryManager(){...}

int
MemoryManager::free(){...}
```

Ausnahme hierbei sind natürlich die Dateien für Template-Klassen, da hier die Implementierung vollständig in der Header-Datei stehen muss.

Kommentare sind im doxygen-Format (siehe www.doxygen.org) zu schreiben, zumindest sind alle Methoden (inkl. Parameter), Member-Variablen und ähnliche Dinge, die in den Header-Files implementiert sind zu dokumentieren.

1.13. Programm Dokumentation: Doxygen

http://www.cypax.net/tutorials/doxygen/index?language=de

Lerneinheit: doxygen und javadoc

1.14. Programm Dokumentation: Javadoc

Lerneinheit: doxygen-javadoc

Quelle: truetigger@chello.at

http://members.chello.at/truetigger/webstart/dokus/javadoc/javadoc7.html

1.14.1. Javadoc Übersicht

JavaDoc erkennt besondere Kommentare innerhalb von Java-Quelltexten und generiert daraus eine API-Dokumentation. Richtig angewendet ergibt sich damit eine sehr gute Code-Dokumentation, die einerseits beim Lesen der Quelltexte selbst hilfreich ist, anderseits auch unabhängig vom Source weitergegeben werden kann und damit für Bibliotheken eine ausgezeichnete API-Dokumentation ergibt, die eine effiziente Verwendung der Bibliotheken gestattet.

Jeder JavaDoc-Kommentar beginnt mit /** - nur in diesen Blöcken werden die sogenannten Tags ausgewertet wie Autor und Version einer Klasse, Variablen, Rückgabewerte und Ausnahmen bei Methoden. Aus den Beschreibungen und den Daten der Tags wird eine API-Dokumentation im HTML-Format generiert, die das Programm in Packages und Klassen unterteilt, einen Index bietet, einen Klassenbaum, zu jeder Klasse die Schnittstellen und noch vieles mehr.

1.14.2. Wer profitiert von einer sauberen Dokumentation mittels JavaDoc?

Der Programmierer selbst:

Spätestens nach ein paar Monaten steckt man nicht mehr so tief im Code drin, und wenn man dann wieder mit dem selbstgeschriebenen Code zu tun hat, muss man sich fast ebenso mühsam einlesen wie in fremden Code. Hier hilft eine Dokumentation der Klassen, Methoden und Variablen.

Das Team:

Programmiert man im Team, ist eine ordentliche Dokumentation eine gewaltige Hilfe. So lässt sich viel Zeit sparen, weil es zu grossen Teilen entfällt, sich in fremden Code einlesen zu müssen. Übrigens ist es gerade im Team einfach, Code- und Dokumentations-Richtlinien durchzusetzen, die man sich allein aus Bequemlichkeit eher schenken würde.

Third-Party-Programmierer:
: Egal ob Closed-Source oder Open-Source, egal ob ein kleines Programm oder ein mächtiges Framework - es kann immer passieren, dass sich aus einem winzigen Projekt eine riesige Sache entwickelt. Ob nun die eigenen Hilfs-Bibliotheken später kommerziell vertrieben werden, fremde Programmierer Erweiterungen oder Verbesserungen für das bestehende Produkt entwickeln wollen - ist eine durchgängige API-Dokumentation vorhanden, erleichtert es die Arbeit um Grössenordnungen.

Code-Reviewer:
: Code altert. Selbst wenn keine Bugs als Notfälle repariert werden müssen, so kommt der Tag, an dem das Programm gegen neu erkannte Sicherheits-Schwachstellen geprüft oder in Sachen Geschwindigkeit optimiert werden muss. Zwar ist bei einem Code-Review im Gegensatz zu den drei anderen Fällen ein genauer Blick in den Source immer nötig, doch mit einer guten Dokumentation kommt man Programmierfehlern viel schneller auf die Schliche, als wenn man erst aufgrund von - möglicherweise fehlerhaftem - Code die Absichten des Programmierers erraten muss.

1.14.3. Vorteil von Dokumentation im Quelltext

Einen Teil der Programm-Dokumentation entsteht unweigerlich außerhalb vom Quelltext: Lasten- und Pflichtenhefte, verwendete Standards, Protokolle und Ideenpapiere. Auch die Beschreibung der API (der Programmierschnittstelle) kann in einem externen Arbeitsschritt erfolgen.

JavaDoc geht den Weg, bestimmte Kommentare innerhalb des Quelltextes auszulesen und zu verarbeiten. Dadurch ergeben sich einige Vorteile:

- Bei Arbeiten direkt am Quelltext hat man auch ohne nachzuschlagen stets die Dokumentation parat.
- Es ist einfacher, Dokumentation und Programm-Teil zugleich zu erstellen.
- Bei Änderungen an der API wird das Abgleichen von Code und Dokumentation nicht so schnell vergessen (vorkommen kann es leider immer noch).
- moderne IDEs unterstützen JavaDoc direkt und bauen die Dokumentation on the fly als Tooltips ein (Eclipse zum Beispiel).

Der einzige Nachteil von JavaDoc: Es kostet während des Programmierens etwas an Zeit, zusätzlich die Dokumentation zu schreiben. Denn Dokumentieren gilt nach wie vor für viele Programmierer als lästige Zusatzaufgabe.

1.14.4. Klassen-Dokumentation

Ein typischer Klassen-Doc-Kommentar steht direkt über der Zeile mit dem Schlüsselwort CLASS (bzw. INTERFACE). Damit werden alle "richtigen" Klassen sowie alle inneren Klassen dokumentiert, anonyme Klassen hingegen nicht.

Ein Klassenkommentar beschreibt die Klasse im Allgemeinen und geht kurz darauf ein, wofür diese Klasse benutzt wird, skizziert Besonderheiten für den Einsatz der Klasse als Beispiele und enthält Hinweise auf den Autor der Klasse sowie auf die Version. Sowohl die Angabe von Author wie auch Version ist optional und eigentlich erst bei der Arbeit in einem Programmierteam interessant: Dann kann es auch mehrere Autoren geben (wenn später noch wesentliche Funktionalitäten hinzugekommen sind), und die Version kann man sehr elegant

mit einem Revision-Control-System wie z.B. **CVS** kombinieren, so dass bei jedem Einchecken die Versions-Nummer im Klassenfile angeglichen wird.

```
/**
 * Diese Bibliothek enthält statische Methoden zum Umformen von
 * Datumsangaben, so dass keine Instanziierung erforderlich ist.
 *
 * System.out.println("23. März: " + DateLib.getZodiacSign("23.03."));
 *
 * @author trueTigger (trueTigger@chello.at)
 * @version 2.0
 */
public class DateLib {
    ...
}
```

Wichtig: Der Klassenkommentar muß DIREKT über der Klasse stehen - etwaige import- und/oder package-Anweisungen gehören noch ÜBER den Klassenkommentar! Klafft zwischen Kommentar und Klassenbeginn eine Lücke, verliert der JavaDoc-Parser den Zusammenhang (siehe auch das Gesamt-Beispiel).

Klassenkommentare sollten auf die folgende Klasse einstimmen, sollten die Motivation erklären, warum die Klasse entstanden ist, warum sie von einer bestimmten Klasse abhängig ist, warum welche Interfaces implementiert sind und so weiter.

1.14.5. Methoden-Dokumentation

Methoden-Doc-Kommentare sind fast die wichtigsten - sie beschreiben neben der Idee hinter der Methode die Parameter, den Rückgabewert und die möglichen Fehler.

So wie Klassenkommentare immer direkt über den Klassendefinitionen stehen, so stehen Methoden-Kommentare direkt über den entsprechenden Methoden.

```
/**
 * Berechnet das Tierkreiszeichen zu einem Geburtstags-Datum. Dabei
 * wird nur der Tag und der Monat betrachtet, das Jahr spielt bei dieser
 * Berechnung keine Rolle.
 *
 * @param dateStr Das Datum als String im Format "dd.mm."
 * @return den (deutschen) Namen des Tierkreiszeichens
 * @throws NullPointerException wenn dateStr null ist
 * @since 1.3
 * @deprecated as of 1.4, replaced by {@link #getZodiacSign(Date)}
 */
public String getZodiacSign(dateStr) {
    ...
}
```

Jeder Parameter sollte mit Namen und kurzer Erklärung aufgeführt werden, mit erwartetem Wertebereich, dazu dient *@param*. Der Rückgabewert sollte bei jeder nicht-void-Methode mit *@return* beschrieben werden.

Mit *@throws* ist die Angabe von Exceptions möglich, die auftreten können - dabei sollten alle Nicht-Runtime-Exceptions auftreten, die sieht man ja auch in der Signatur der Methode (und die API-Dokumentation beschreibt diese Fehler auch). Ebenso ist es möglich, bekannte Runtime-Exceptions in der Dokumentation zu vermerken und damit schon in der

Dokumentation auf mögliche Fehlerfälle hinweisen.

Mit *@since* kann man Methoden deklarieren, die in früheren Versionen des Frameworks noch nicht vorhanden waren. Pflegt man diese Tags sorgfältig, bietet man Third-Party-Entwicklern ein sehr brauchbares Hilfsmittel.

Schließlich kann man mit *@deprecated* eine Methode als "veraltet" kennzeichnen. Normalerweise läßt man solch veralteten Code bestehen, damit darauf aufsetzende Programme nicht plötzlich streiken. Aber für neu zu entwicklende Programme sollte auf diese Methoden dann verzichtet werden. Das JDK von Sun ist voll von deprecated-Methoden aus der langjährigen Geschichte von Java.

1.14.6. Variablen-Dokumentation

Auch hier steht der Kommentar unmittelbar über der Variable. Es lassen sich Klassen- und Instanz-Variablen dokumentieren, lokale Variablen innerhalb von Methoden hingegen nicht. Die Kommentare für Variablen bestehen im allgemeinen aus nur einer Zeile.

```
/** Farb-Konstante für Fehlermeldungen (hellrot) */
public static final Color ERROR_COLOR = new Color(255, 127, 127);

/** speichert die letzte Fehlermeldung */
private String errorMsg;

/**
 * Wenn man einen langen Kommentar schreiben will, kann man
 * natürlich auch bei einem Variablenkommentar mehrere Zeilen
 * verwenden.
 */
protected Vector aVector;
```

1.14.7. zusammenfassendes Beispiel

=> JavaDocExample.java

```
/* JavaDocExample.java, created on 2004-07-28
 *
 * Ein normaler Kommentar am Anfang jeder Datei eignet sich hervorragend
 * für Copyright-Hinweise und Lizenzbestimmungen.
 */
package at.tigger.tutorial;

// darauf achten, dass imports VOR dem Klassenkommentar stehen!
import java.net.InetAddress;
import java.net.NetworkInterface;
import java.net.SocketException;
import java.util.Enumeration;

/**
 * <p>Eine Beispiel-Klasse zur Demonstration von JavaDoc</p>
 *
 * <p>In der Klassenbeschreibung sollte allgemein Informatives über die Klasse
 *  selbst stehen, Hinweise auf die Benutzung der Klasse und Aufzählung von
 *  weiterführender Literatur (z.B. wenn bekannte Algorithmen nachprogrammiert
```

```
 *  werden).</p>
 *
 * <p>Wie man sieht kann man in längeren Java-Doc-Kommentaren durchaus
 *  sinnvoll HTML verwenden. Man sollte sich jedoch auf ein allgemein
 *  gültiges Subset beschränken:</p>
 *
 * <ul>
 *  <li>Paragraphen</li>
 *  <li>Aufzählungen (numerierte und unnumerierte)</li>
 *  <li><b>fette</b> und <i>kursive</i> Schrift</li>
 * </ul>
 *
 * @author TrueTigger (<a href="mailto:trueTigger@chello.at">mail</a>)
 * @version 1.0
 */
public class JavaDocExample {

    /** die angezeigte Bezeichnung für "Network interface" */
    public static final String IFACE_NAME = "Netzwerk-Interface";
    /** die angezeigte Bezeichnung für "Internet address" */
    public static final String IADDR_NAME = "IP-Adresse";

    /**
     * Gibt alle Netzwerk-Interfaces des aktuellen Systems zurück.
     *
     * @return alle Netzwerk-Karten verpackt in eine Enumeration
     * @throws SocketException falls ein Netzwerk-Gerät Probleme verursacht
     */
    public static Enumeration getNetworkInterfaces() throws SocketException {
        return NetworkInterface.getNetworkInterfaces();
    }

    /**
     * Ein Netzwerk-Interface kann durchaus mehrere IP-Adressen besitzen.
     * Diese Methode sucht alle IP-Adressen zusammen, die einem bestimmten
     * Netzwerk-Interface zugeordnet sind.
     *
     * @param iFace das Netzwerk-Interface, deren IPs gefragt sind
     * @return alle IP-Adressen verpackt in eine Enumeration
     * @throws NullPointerException wenn das Netzwerk-Interface null ist
     */
    public static Enumeration getAllIps(NetworkInterface iFace) {
        return iFace.getInetAddresses();
    }

    /**
     * <p>Die main()-Methode wird beim Programmstart ausgeführt.</p>
     *
     * <p>Es werden alle IP-Adressen des Computers ausgegeben, und falls es
     *  sich um private IP-Adressen
     *  (siehe <a href="http://www.ietf.org/rfc/rfc1918.txt">RFC 1918</a>)
     *  oder um Loopback-Adressen handelt, wird dies ebenfalls ausgegeben.</p>
     *
     * @param args die Kommandozeilen-Parameter (hier nicht benutzt)
     * @throws SocketException bei Problemen mit Netzwerk-Interfaces
     */
    public static void main(String[] args) throws SocketException {

        for(Enumeration ni = getNetworkInterfaces(); ni.hasMoreElements(); ) {
```

```
            NetworkInterface iFace = (NetworkInterface)ni.nextElement();
            System.out.println(IFACE_NAME + ": " + iFace.getDisplayName());

            for(Enumeration ia = getAllIps(iFace); ia.hasMoreElements(); ) {

                InetAddress address = (InetAddress)ia.nextElement();

                String addrStr = address.getHostAddress();
                System.out.print("\t" + IADDR_NAME + ": " + addrStr);

                if(address.isLoopbackAddress()) {
                    System.out.print(" (Loopback)");
                }
                if(address.isSiteLocalAddress()) {
                    System.out.println(" (private IP)");
                }
                System.out.println();
            }
        }

    }
}
```

1.14.8. Rendern als HTML-Seiten

JavaDoc startet man in dem Wurzelverzeichnis seines Projekts und gibt den Namen des Packages an, für welches die Doku erzeugt werden soll. Zum Beispiel liegen die Java-Dateien unter C:\java\beispiel (gemäß des angegebenen Packages natürlich im Unterordner at\tigger\tutorial). Dann wechselt man in das Verzeichnis C:\java\beispiel und startet dort JavaDoc.

- javadoc at.tigger.tutorial
  => erzeugt die komplette API für das Package mit Default-Werten.
- javadoc -doctitle "JavaDoc-Beispiel" at.tigger.tutorial
  => erzeugt die API mit einem eigenen Namen
- javadoc -private at.tigger.tutorial
  => nimmt auch private Methoden/Felder mit in die Doku auf
- javadoc -link http://java.sun.com/j2se/1.4/docs/api at.tigger.tutorial
  => Referenzen auf die Java2-API werden sauber verlinkt
- javadoc -d doc/api at.tigger.tutorial
  => Schreibt die Dateien in das angegebene Verzeichnis statt ins aktuelle

Das abschließende Beispiel ist einmal mit allen Optionen zusammen erstellt worden:

- javadoc -doctitle "JavaDoc-Beispiel" -private -link http://java.sun.com/j2se/1.4/docs/api -d doc/api at.tigger.tutorial
  => generierte JavaDoc

1.14.9. Zusammenfassung: Javadoc und Eclipse

Javadoc ist ein Tool zur **Generierung von API Dokumentationen** aus den Kommentarzeilen im Quellcode. Es kann als ein Teil des Java 2 SDK heruntergeladen werden und stammt wie Java von Sun Microsystems.

Javadoc parst also den Quellcode und erstellt HTML Seiten, die den Programmcode beschreiben.

Damit der Parser weiß, was er in die HTML-Dateien schreiben soll müssen ein paar Konventionen beachtet werden.

Ein Javadoc-Kommentar wird im Gegensatz zu normalen mehrzeiligen Kommentaren mit /** begonnen und mit */ abgeschlossen.

Zusätzlich gibt es sogenannte Doclet-Tags, die bestimmte Sachverhalte genauer beschreiben.

Ein Beispiel:

```
/**
 * Klasse HalloWorld
 *
 * @author Tobias Seckinger
 * @version 1.0
 */
public class HalloWorld {
	/**
	 * Statische Var <i>variable</i>, die den Hallo World String hält
	 */
	private static String variable = "Hallo World";

	/**
	 * Die Main-Methode
	 *
	 * @param args
	 */
	public static void main(String[] args) {
		System.out.println(variable);
	}
}
```

Welche Tags es gibt kann man am besten auf der Sun Webseite nachlesen:

```
* @param       (classes, interfaces, methods and constructors only)
* @return      (methods only)
* @exception   (@throws is a synonym added in Javadoc 1.2)
* @author      (classes and interfaces only, required)
* @version     (classes and interfaces only, required.)
* @see
* @since
* @serial      (or @serialField or @serialData)
* @deprecated  (see How and When To Deprecate APIs)
```

Wie aber erstellt man nun die Dokumentation?

Am Beispiel von Eclipse wird gezeigt, wie man zu seiner Quellcode-Dokumentation kommt. Natürlich kann dies auch von der Konsole aus erledigt werden.

Grundvoraussetzung ist ein vorhandener Javadoc Kompiler, der in dem Java 2 SDK bereits enthalten ist z. B. C:\Programme\java\jdk1.6.0\bin\javadoc.exe. Wer mit Eclipse Java-Programme entwickelt hat in der Regel bereits eins installiert. Wenn nicht, kann man das bei Sun natürlich downloaden. In Eclipse muss man dies dann unter Windows > Properties > Java > Installed JREs einrichten.

Alles andere ist ganz einfach: Über Project > Generate Javadoc… erscheint folgender Javadoc Generation Dialog:

Wir sehen im Feld Javadoc command den Pfad zum Javadoc Kompiler angezeigt. In der Textbox links können wir auswählen, für welches Projekt wir für unsere Dokumentation erstellen wollen. Anschließend können wir auswählen, ob wir alle private, package, protected oder public Elemente in die Generierung mit einbeziehen wollen.

In der Regel benötigen wir keine weiteren Angaben. Wir können also getrost auf Finish klicken und anschließend die index.html aufrufen.

Fertig!

1.15. +Projekt: TAF - Wetterbericht

Siehe: u-seng-taf.odt

1.16. +Projekt: Weinhandel

Siehe: u-seng-weinhandel.odt

1.17. +Projekt: IOWarrior Klassenbibliothek

Siehe: u-seng-iowarrior.odt

1.18. +Projekt: InfoBank

Siehe: u-seng-infobank.odt

1.19. Zusammenfassung

1.19.1. Opensource Entwicklung und ihre Dynamik

http://www.little-idiot.de/his/t1.htm