Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Programmierparadigmen
Am Fasanengarten 5
76131 Karlsruhe
http://pp.ipd.kit.edu/

Praxis der Software-Entwicklung
Tipps und Tricks

Lehrstuhl Programmierparadigmen
Karlsruher Institut für Technologie (KIT)

Stand: 10. April 2014

Manche Tipps und Tricks wiederholen wir häufig. Dies ist eine Sammlung, um nichts zu vergessen und weniger Fehler beim Erklären zu machen.

1 Technisches

> Good programmers use their brains, but good guidelines save us having to think out every case.
>
> *(Francis Glassborow)*

1.1 Versionskontrollsystem

Wir empfehlen Subversion oder Git, weil wir als Betreuer damit Erfahrung haben.

Git-Tipps:

- Interactive Git Tutorial: für Anfänger.
- Git Reference: relativ kompakte Anleitung.
- Offizielle Git Projektseite
- Git for Computer Scientists: „Quick introduction to git internals for people who are not scared by words like Directed Acyclic Graph.“
- Vermeide den Parameter `-force` bzw. `-f`. Häufig schiebst du damit Probleme nur deinen Teamkollegen zu.

Genereller Tip: Keine generierten Dateien ins einchecken. Zum Beispiel erzeugt TEX üblicherweise einige Dateien (Endungen wie log, aux, out oder blg) beim Erstellen einer pdf. Diese sollten nicht im Versionskontrollsystem landen. Ebensowenig die erzeugte pdf.

1.2 Dokumentenformat

Wir empfehlen LATEX, weil es sich gut mit Versionskontrollsystemen vereinbaren lässt und man hier schon für die Bachelorarbeit üben kann. Office o.ä. ist aber auch möglich.

1.2.1 LATEX

Statt den normalen Dokumentklassen, empfiehlen wir vor allem für deutsche Dokumente die KOMA-Scriptklassen zu verwenden. Diese sind flexibler und besser für deutsche Typografie ausgelegt. Das bedeutet deutsche Dokumente starten mit der folgenden Deklaration.

```
\documentclass[parskip=full]{scrartcl}
```

Das `parskip=full` sorgt für eine Absatzabstand von einer Zeile und die erste Zeile eines Absatzes wird nicht eingerückt. Das ist im Deutsch das übliche Format, aber nicht im Englischen.

Weitere für (deutsche) LATEX-Dokument fast immer gute Einstellungen:

```
\usepackage[utf8]{inputenc} % use utf8 file encoding for TeX sources
\usepackage[T1]{fontenc}    % avoid garbled Unicode text in pdf
\usepackage[german]{babel}  % german hyphenation, quotes, etc
\usepackage{hyperref}       % detailed hyperlink/pdf configuration
\hypersetup{                % ‘texdoc hyperref‘ for options
  pdftitle={PSE: Tipps},
  bookmarks=true,
}
\usepackage{csquotes}       % provides \enquote{} macro for "quotes"
```

Die Dokumentation der Pakete ist häufig lesenswert. Insbesondere bei den Paketen hyperref und scrguide (KOMA-Script). Wer TeXLive per Kommandozeile benutzt kann einfach `texdoc scrguide` aufrufen.

Zum Erzeugen eines PDFs aus den LATEX-Sourcen empfehlen wir einen Wrapper wie `latexmk` zu verwenden. Dieser übernimmt beispielsweise das mehrfache Ausführen von `pdflatex`, wo es notwendig ist.

1.3 GUI Entwürfe

- Inkscape ist ein freies Vektorzeichenprogramm

- Pencil ist ein Open-Source GUI Prototyping Tool (enthält Formen für Android, iOS, Web, GTK, Windows XP, Sketch, ...)
- Die Android Developer Tools enthalten einen grafischen UI Builder
- Papier und Tusche geht natürlich auch

1.4 UML Diagramme

- Umbrella um einfach nur Diagramme zu zeichnen
- ArgoUML um auch Code zu generieren und beim Entwurf zu helfen
- IBM Rational Software Architect ist im ATIS Pool installiert.
- BOUML noch ein UML Tool
- UMLet noch ein UML Tool
- UMLGraph für Versions-Control-freundliche UML Diagramme.

2 Kolloquium

> The success of your presentation will be judged not by the knowledge you send but by what the listener receives.
>
> *(Lilly Walters)*

Ein paar Hinweise für die Kollquien bzw. für Präsentation allgemein.

- Alleine Üben! Daheim vor dem Spiegel üben. Einen Vortrag nur alleine, aber im Stehen und laut sprechend, vorzutragen ist meist schon lehrreich im Vergleich zu stillem Folienbasteln.
- Vor dem eigenen Team üben und konstruktiv gegenseitig kritisieren. Oft sehen die Kollegen Ticks, die man selbst nicht wahrnimmt. Beispielsweise häufige Ähs, häufiges Wegschauen vom Publikum, verschränkte Arme, Nuscheln, nervöses herumlaufen, etc.
- Lauf und deutlich sprechen.
- 10 Minuten sind nicht lange, aber man kann viel Information darin unterbringen.
- Prof. Snelting achtet sehr auf die Zeit. 10 Minuten sind nicht 12 Minuten und auch nicht 7 Minuten.
- Daumenregel: 2 Minuten pro Folie, also 5 Folien. Ein Folie umfasst dabei möglicherweise Animationen / Overlays.

- Die Folien dem Betreuer zur Durchsicht geben.
- Auf die interessanten Punkte konzentrieren und sich nicht in den Details verlieren. Trotzdem sollte natürlich der Inhalt möglichst vollständig sein.
- Keine Agenda-/Inhaltsverzeichnis-/Gliederungsfolie. Ist nicht mehr zeitgemäß und langweilt nur.

3 Organisatorisches

> Design and programming are human activities; forget that and all is lost.
>
> *(Bjarne Stroustrup)*

- Meldet euch so früh wie möglich im Studierendenportal für PSE (Pr.Nr. 529) und TSE (Pr.Nr. 455) an.
- Mails an den Betreuer sollten vom Phasenverantwortlichen kommen.

4 Pflichtenheft

> I have always found that plans are useless, but planning is indispensable.
>
> *(Dwight Eisenhower)*

Artefakt der Phase Pflichtenheft: PDF Dokument

Das fertige (Software-)Produkt wird am Ende der gesamten Entwicklung am Pflichtenheft gemessen (Stichwort Endabnahme). Stellt der Auftraggeber zu große Unterschiede fest, wird er nicht bezahlen. D.h. schon das Pflichtenheft muss es dem Leser ermöglichen, eine exakte Vorstellung des fertigen Produkts zu bekommen. Insbesondere müssen *alle* Produktfunktionen und -daten genannt und hinreichend genau beschrieben werden, (G)UI-Entwürfe sind Pflicht, Bedienelemente müssen erklärt sein (z.B. Menüführung), und der Leser muss durch das Pflichtenheft wissen, wie er das fertige Produkt verwenden kann (Stichwort Testfallszenarien).

Musskriterien Mindestanforderungen, gehen aus Aufgabenstellung hervor. Möglichst klein halten.

Wunschkriterien Von den Gruppen selbst definierte, zusätzliche Funktionalität.

Abgrenzungskriterien Was unterstützt unser Produkt explizit *nicht*.

Und noch einige generelle Anforderungen an das Pflichtenheft.

- Wird jedes Kriterium durch mindestens einen Testfall geprüft? Zumindest intern (möglicherweise auch direkt im Pflichtenheft selbst) sollte für jedes Kriterium eine Liste an Testreferenzen und für jeden Test eine Liste an Kriteriumsreferenzen feststehen.

- Man stelle sich vor, das Pflichtenheft wird für Entwurf und Implementierung an ein anderes Team übergeben und das Ergebnis kommt zur Qualitätssicherung zurück. Wie zuversichtlich seid ihr, dass das Ergebnis euren Vorstellungen entspricht? Abweichungen von euren Vorstellung dürfen nur kritisiert werden, wenn dadurch Testfälle fehlschlagen.

- Das Pflichtenheft sollte auch von nicht technisch-versierten Menschen zu großen Teilen verstanden werden können. Am besten jemand fach-fremden zum Lesen geben und danach Verständnisfragen stellen.

- Erfahrungsgemäßer Umfang: ca. 40 Seiten

4.1 Struktur

Beschreibung und Beispiele von Pflichtenheften finden sich in der Vorlesung Softwaretechnik 1. Die Struktur muss aber nicht exakt übernommen werden. Sinnvolle Änderungen könnten sein:

- Funktionale Anforderungen von Muss- und Wunschkriterien mischen und durch Layout, Stil, Marker, Icons entsprechend kennzeichnen.

- Ein Pflichtenheft darf mehr enthalten als nur einen Katalog an Kriterien. Das Endprodukt soll komplett erklärt werden, also könnte Hintergrundwissen aus anderen Quellen hilfreich sein.

- Reihenfolge der Kapitel verändern, so dass es flüssiger von vorne nach hinten gelesen werden kann.

- Testfallszenarien als Liste formatieren, so dass es als Checkliste für den Tester fungiert. Ein Punkte sollte dabei immer genau eine Aktion des Testers und die erwartet Reaktion des Programms sein.

4.2 Technisches Schreiben

Technisches Schreiben ist wichtig für alle Arten von technischen und wissenschaftlichen Dokumenten, also auch API-Dokumentation, Bachelorarbeit und Pflichtenheft. Es bedeutet vor allem eine präzise Ausdrucksweise und widerspricht dabei einigen Regeln, die man im Deutschunterricht gelernt hat. Ein paar praktische Tipps:

- Vermeide Adjektive. Oft (nicht immer) sind sie unnötig oder ein schlechter Ersatz für einen ungenauen Begriff.

- Definiere Begriffe klar und verwende keine Synonyme. Synonyme lassen offen, ob genau das gleiche gemeint ist oder nur etwas ähnliches.

- Abkürzungen sollten bei der ersten Verwendung (EV) ausgeschrieben werden. Nach der EV reicht dann die Kurzform.

- Versuche konkrete Zahlen und Namen anzugeben. Vermeide ungenaue Ausflüchte wie: meistens, viele, oft, möglichst, üblich, jemand, manche.

- Viele kurze Sätze sind einfacher zu verstehen als wenige lange Sätze.
- Beispiele machen das Endprodukt greifbarer.
- Illustrationen minimalistisch halten (z.B. IKEA Bauanleitung). Eine Information, ein Bild. Lieber mehrere ähnliche Bilder als ein komplexes Bild.
- Vermeide Wiederholung, stattdessen Referenzen benutzen. Wiederholungen haben oft subtile Unterschiede, was zu Unklarheit und Verwirrung führt. Bei Änderungen wird oft vergessen, dass Wiederholungen auch angepasst werden müssen.
- Versionskontrolle ergibt auch für technische Texte Sinn und nicht nur für Code.

Das Pflichtenheft ist das entscheidenste Dokument zwischen Kunde und Entwickler am Ende eines Projekts. Es darf **keinen Spielraum für Interpretationen** offen lassen.

4.3 Inhalt der Präsentation

- Kurze Einführung zur Aufgabenstellung
- Überblick über die wichtigsten Features
- Grundsätzliche selbstgesetzte Rahmenbedingungen
- Ein Testfallszenario als anschauliches durchgehenden Beispiel

5 Entwurf

> Design is the art of separation, grouping, abstraction, and hiding. The fulcrum of design decisions is change. Separate those things that change for different reasons. Group together those things that change for the same reason.
>
> *(Bob Martin)*

Artefakt der Phase Entwurf: PDF Dokument mit UML-Diagrammen, Klassenbeschreibungen, Erläuterungen, Designentscheidungen

5.1 Inhalt

- Einleitung mit grobem Überblick. Dieser Abschnitt soll an das Pflichtenheft anschließen und die Aufteilung in die Pakete erklären.

- Detaillierte Beschreibung *aller* Klassen. Das beinhaltet (JavaDoc) Beschreibungen zu allen Methoden, Konstruktoren, Packages und Klassen.

- Beschreibung von charakteristischen Abläufen anhand von Sequenzdiagrammen. Beispielsweise bieten sich Testszenarien aus dem Pflichtenheft hier an. Wir empfehlen Sequenzdiagramme möglichst früh zu erstellen, dann dabei werden die Schnittstellen zwischen Packages und Klassen klar.

- Aufteilung in Klassen/Pakete die unabhängig voneinander implementiert und getestet werden können. Mit Blick auf den Implementierungsplan.

- Änderungen zum Pflichtenheft, bspw. gekürzte Wunschkriterien

- Vollständiges großformatiges Klassendiagramm im Anhang. Ausschnitte/Teile können bereits vorher verwendet werden, um Teilkomponenten zu beschreiben.

- Identifikation von Entwurfsmustern um Struktur gröber zu beschreiben (möglicherweise).

- Erfahrungsgemäßer Umfang: über 100 Seiten, primär Klassenbeschreibungen

5.2 Bewertung eines Entwurfs

Hier ein paar Tipps, wie man einen Entwurf einschätzen kann. Paradoxe Anforderungen zeigen, dass gutes Design ein Kunst ist.

- Geheimnisprinzip (information hiding) beachtet? Jede Entwurfsentscheidung sollte in genau einer Klasse gekapselt sein, so dass eine Änderung dieser Entscheidung auch nur diese eine Klasse betrifft. Allgemein sollte ein Klasse/Paket möglichst wenig interne Details nach außen preisgeben.

- Lose Koppelung zwischen Klassen/Paketen? Abhängigkeiten zu fremden Schnittstellen machen spätere Änderungen aufwendiger. Im UML Diagramm sollten möglichst wenig Verbindungen zwischen Klassen zu sehen sein. Siehe auch Law of Demeter.

- Starke Kohäsion innerhalb von Klasse/Paket? Wenn Methoden einer Klasse eigentlich unabhängig voneinander sind, ist es ein Zeichen, dass eine Auftrennung in zwei Klassen sinnvoll sein könnte. Kohäsion führt zu besserer Wiederverwendbarkeit der einzelnen Klassen.

- Klassen/Pakete sind gleichzeitig erweiterbar und stabil? Erweiterbarkeit bei stabilem Interface ist der große Vorteil von objekt-orientertem gegenüber proceduralem Entwurf, der durch Vererbung und Polymorphie erreicht wird. Siehe auch Open-Closed Principle.

- Liskovsches Substitutionsprinzip bei Vererbung erfüllt? Unterklassen sollte alle Nachbedingungen und alle Invarianten der Oberklasse erfüllen. Andernfalls könnte es zu Fehlern kommen, wenn eine Unterklasse als Oberklasse verwendet wird.

- Verhalten von Implementierung getrennt? Das Verhalten (Was soll getan werden) ändert sich sehr viel häufiger als die Implementierung (konkrete Algorithmen). Beispielsweise sind Sortieralgorithmen recht statisch, während die Frage wonach sortiert werden soll, sehr flexibel sein sollte.

- Keine zyklischen Abhängigkeiten? Beispielsweise ist eine zyklische Abhängigkeit von Konstruktoren schlicht nicht möglich. Eine zyklische Abhängigkeit von größeren Modulen bedeutet, dass man alles auf einmal implementieren muss, bevor irgendwas funktioniert. Entwurfsmuster um Abhängigkeiten zu entfernen: Observer, Visitor, Strategy

- Lokalitätsprinzip beachtet? Eine Klasse/Paket/Methode sollte für sich verständlich sein, ohne das Kontext notwendig ist.

5.3 Mehr Links

- TeXdoclet um mit JavaDoc LaTeX zu erzeugen. Das ermöglicht den automatischen Flow: ArgoUML → JavaDoc+TeXdoclet → LaTeX → Section „Klassenbeschreibung“ im Entwurf.

- Prinzipien für den objektorientierten Entwurf, Guter Überblicksartikel.

- Prinzipien Der Softwaretechnik, Blog zum Thema Prinzipien im Software Engineering.

5.4 Inhalt der Präsentation

- Kurze Einführung und Verbindung zum Pflichtenheft

- Ein Sequenzdiagram als anschauliches Beispiel mit Ausschnitten aus dem Klassendiagramm.

- Überblick über das Gesamtklassendiagramm, Pakete, Module geben.

- Gestrichene Wunschkriterien

- Verwendete externe Resourcen (Bilder,Frameworks,Bibliotheken,Sounds,Musik,etc)

- Nur kurz erwähnen weil eher langweilig: Eigene Formate, Datenbankschemata, Einstellungen und Dialoge.

6 Implementierung

> Talk is cheap. Show me the code.
>
> *(Linus Torvalds)*

Artefakt der Phase Implementierung: Implementierungsplan zu Beginn, Implementierungsbericht und vollständiger source code

6.1 Implementierungsplan

- Klarer zeitlicher Ablauf der Implementierungsphase, um frühzeitig Verzögerungen zu bemerken.
- Klare Aufgabenverteilung im Team.
- Abhängigkeiten aus dem Entwurf müssen beachtet werden.
- Als Form bietet sich ein GANTT Chart an. Verpflichtend ist es aber nicht.
- Zu *Beginn* der Implementierungsphase beim Betreuer abzugeben.

6.2 Implementierungsbericht

- Erfahrungsgemäßer Umfang: ca. 20 Seiten
- Einleitung mit Anschluss auf Pflichtenheft und Entwurf
- Dokumentation über Änderungen am Entwurf, beispielsweise entfernte oder neu hinzugefügte Klassen und Methoden
- Welche Muss- und Wunschkriterien sind implementiert?
- Welche Verzögerungen gab es im Implementierungsplan? Kann beispielsweise als zweites GANTT Diagramm am Ende dargestellt werden.
- Übersicht zu unit tests

6.3 Unittests

- Von Anfang an Unittests benutzen. Diese Tests gehören *nicht* in die Phase Qualitätssicherung.
- Vor allem nicht-graphische Klassen können so frühzeitig getestet und Regressionen vermieden werden.
- Nur öffentliche Schnittstellen testen. Private Methoden werden nicht getestet.
- Unittests testen nur kleine „units“ (üblicherweise einzelne Klassen) für sich. Es sollen nicht ganze Testszenarien getestet werden.

6.4 Sonstiges

- Warnungen reparieren. Es ist empfehlenswert, dass ein Projekt beim Bauen keine Warnungen ausgibt. Eine Warnung bedeutet, dass der Code üblicherweise aber nicht garantiert fehlerhaft

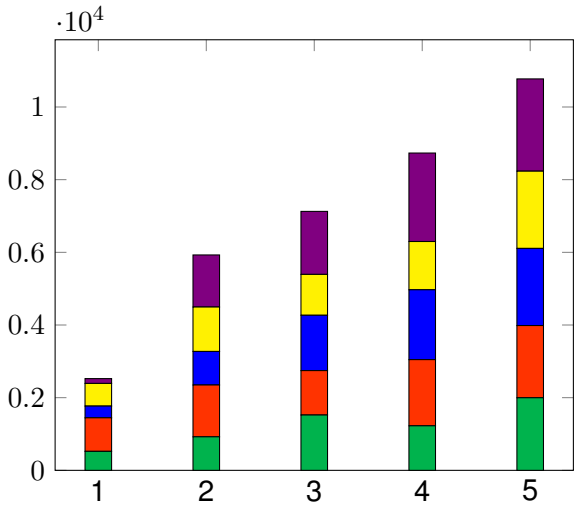

Abbildung 1: Anzahl Codezeilen je Person über 5 Wochen hinweg. Diese Art von Graph zeigt dass die Verteilung im Team fair aussieht. Auch sieht man den kontinuierlichen Wachstum.

ist. In Ausnahmen kann der Programmierer in Java dann Annotationen einfügen um Warnungen zu unterdrücken.

- Warnungen anschalten. In Eclipse kann man zusätzliche Warnungen anschalten: Project properties → Java Compiler → Errors/Warnings. Standardmäßig wird das meiste ignoriert. Beispielsweise kann man aktivieren, dass eine Zuweisung innerhalb einer Bedingung „`if(a=b)`“ eine Warnung (oder sogar ein Fehler) ist.

- Kommentare im Code (also nicht API Doku) sollten das „Warum“ klären, nicht das „Wie“. Es ist naturgemäß schwierig vorherzusehen, welche Warum-Fragen sich jemand stellt, der ein Stück Code liest. Die Fragen sind üblicherweise „Warum ist X hier notwendig?“, „Warum ist kein X hier?“ oder „Warum X, wobei Y doch besser wäre?“.

- Exceptions (nicht RuntimeException) für Fehler beim Aufrufer. Assertions für interne Konsistenzfehler bzw. um Annahmen explizit zu machen.

6.5 Inhalt der Präsentation

- Was wurde implementiert? Welche Wunschkriterien gestrichen?

- Zeitplan eingehalten? Wo nicht?

- Unerwartete Probleme bei der Implementierung?

- Größere Änderungen am Entwurf?

- Grobe Statistiken. Lines of Code, Wieviele Commits, Arbeitsaufteilung im Team.

7 Qualitätssicherung

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> *(Edsger Dijkstra)*

Artefakt der Phase Qualitätssicherung: Testberichte

- Überdeckung der Unittests maximieren. (Siehe bspw.: EclEmma) Wenn GUIs im Spiel sind, ist 100% Überdeckung üblicherweise nicht möglich.
- Komplette Testszenarien aus dem Pflichtenheft durchgehen. Ein Testdurchlauf sollte möglichst automatisch ablaufen.
- Monkey Testing für einfaches GUI testen. Der Ertrag ist vermutlich nicht sehr groß, also nicht zuviel Aufwand hineinstecken.

  Automatisierung mit Robotium und UIAutomator?
- Lint und andere statische Werkzeuge recherchieren. Codequalität kann gar nicht gut genug sein.
- Hallway Usability Testing. Systematisches Vorgehen ist wichtig, also vorher Fragen, Aufgaben und Umfang festlegen. Qualitative (Interviewzitate) und quantitative (Durchschnitt über alle Tester) Ergebnisse dokumentieren.
- Alle gefunden Bugs und deren Reparatur dokumentieren. So entsteht der Testbericht am Ende.

8 Abschlusspräsentation

> We love to hear stories. We don't need another lecture. Just ask kids.
>
> *(George Torok)*

Artefakt der Phase Abschlusspräsentation: Präsentationsfolien

Showtime!

8.1 Inhalt

Wichtigster Inhalt ist „**Was rauskam**“. Präsentiert eure Applikation. Das darf so kreativ sein, wie ihr euch traut. Von der Livedemo bis zum Theaterspielen ist alles erlaubt.

Nebenbei dürfen folgende Informationen auch in den Vortrag:

- Kurze Statistik: Wieviel Zeilen Code, wieviele Commits und Tests, wieviel Überdeckung?
- Lernerfahrung: Was hat erstaunlich gut funktioniert? Was war aufwendiger als gedacht? Was würden wir beim nächsten Mal anders machen? Insbesondere Soft-Skill Erfahrung in Sachen Teamarbeit und Organisation sind hier interessant.

Zu diesem Zeitpunkt habt ihr erfolgreich ein ordentliches Softwareprojekt von Anfang bis Ende durchgeführt. Selbst wenn nicht alles glatt lief, so könnt ihr trotzdem Stolz auf euch sein. Egal was ihr gemacht habt, völlig falsch kann es nicht gewesen sein.

8.2 Mehr Links

- Presentation Zen
- 9 Tips How to Give a Technical Presentation
- The Science of Making Your Story Memorable
- Backstage at the First iPhone Presentation

9 Feedback

Wir wissen gerne was wir besser machen könnten. Falls ihr also den Jahrgängen nach euch etwas Gutes tun wollt, dann gebt eurem Betreuer ein Feedback. Zum Beispiel könnt ihr die folgenden beiden Fragen beantworten:

1. Auf einer Skala von 0 (schlecht) bis 10 (perfekt), wie fandet ihr PSE/TSE? ☐

2. Warum in Frage 1 genau diese Zahl?