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: 9. November 2021

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

1 Tools für alle Phasen

> 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:

- Learn Git Branching: Interaktive Einführung 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.

- Benutze den Parameter `--rebase` bei einem `git pull`, damit vermeidest du, dass alles voller merge-commits ist. Alternativ kann man das (ab git 1.7.9) auch als globalen Standard einstellen mit `git config -global pull.rebase true`.

Genereller Tip: Keine generierten Dateien 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. Bei Git kann man hierzu z.B. die Datei `.gitignore` erstellen/verwenden.

Man kann sich optional auch Issue Tracker, Code Review, und ähnliche Tools gleich dazuholen, beispielweise bei Github, Bitbucket, Gitlab oder git.scc.kit.edu.

1.2 Dokumentenformat: LATEX

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.

Statt den normalen Dokumentklassen empfehlen 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 einen Absatzabstand von einer Zeile und die erste Zeile eines Absatzes wird nicht eingerückt. Das ist im Deutschen das übliche Format, aber nicht im Englischen.

Weitere für (deutsche) LATEX-Dokumente 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.

2 Technisches Schreiben

> Anything that helps communication is good. Anything that hurts is bad. And that's all I have to say.
>
> *(Donald Knuth)*

Technisches Schreiben ist wichtig für alle Arten von technischen und wissenschaftlichen Dokumenten, also auch API-Dokumentation, Bachelorarbeit, Pflichtenheft und Testbericht. 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.
- Nebensatzkonstruktionen vermeiden; Hauptsätze verwenden!
- 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.
- Benutze nicht die automatische Umbruch-Funktion des Editors. Setze stattdessen im Sourcecode einen Zeilenumbruch nach jedem Satz. Damit werden die Diffs des Versionskontrollsystems lesbarer.

3 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.
- Laut 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.
- Ihr seid nicht gezwungen den KIT Folienstil zu verwenden. Es ist sogar eher falsch, schließlich repräsentiert ihr nicht das KIT, sondern nur euch als Team von Studenten.

4 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 und TSE an:
  - Prüfungsnummern 7500076 und 7500075
- Mails an den Betreuer sollten vom Phasenverantwortlichen kommen.

5 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. Diese *müssen* im fertigen Produkt implementiert sein, und reichen i.d.R. zum Bestehen von PSE. Möglichst klein halten.

Wunschkriterien Von den Gruppen selbst definierte, zusätzliche Funktionalität. Kein einzelnes der Wunschkriterien muss am Ende implementiert sein. Wunschkriterien können im Zweifelsfall sehr optimistisch ausgelegt werden, da vorausschauende Projektplanung schwierig ist und es nicht schlimm ist, wenn am Ende nicht alle Wunschkriterien implementiert sind.

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 fachfremdem zum Lesen geben und danach Verständnisfragen stellen.
- Das Pflichtenheft ist das entscheidendste Dokument zwischen Kunde und Entwickler am Ende eines Projekts. Es darf **keinen Spielraum für Interpretationen** offen lassen.
- Erfahrungsgemäßer Umfang: ca. 40 Seiten

5.1 Struktur

Beschreibung und Beispiele von Pflichtenheften finden sich in der Vorlesung Softwaretechnik 1.

Als „erweitertes Lastenheft“ enthält es zusätzlich: Produktumgebung, Testfälle sowie Entwürfe der Benutzerschnittstelle. Objektmodell und dynamische Modelle sind i.d.R. nicht verpflichtender Teil des PSE-Pflichtenhefts (detaillierte Klassen- und Sequenzdiagramme sind Teil des *Entwurfs*).

Die Struktur aus SWT1 muss nicht exakt übernommen werden. Sinnvolle Änderungen könnten sein:

- Reihenfolge der Kapitel verändern, so dass es flüssiger von vorne nach hinten gelesen werden kann.
- Kapitel, die im Kontext des eigenen Projektes keinen Sinn machen oder übermäßig redundant werden würden, können potenziell weggelassen werden. Eventuell kann es auch sinnvoll sein, Kapitel zusammenzufassen.
- Funktionale Anforderungen von Muss- und Wunschkriterien mischen oder nach Programmbereich aufteilen und stattdessen 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. Auch Bilder und Grafiken sind erlaubt.
- 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 erwartete Reaktion des Programms sein. Beispiel:

  T1337 Einloggen

  1. **Stand** Offenes Browserfenster.

     Aktion Benutzer gibt „facebook.com“ in die Kopfzeile ein und drückt Enter.

     Reaktion Der Browser wechselt zur Facebook-Frontseite.

  2. **Stand** Facebook-Frontseite ist geladen. Loginmöglichkeit oben rechts.

     Aktion Benutzer gibt „zuckerberg@facebook.com“ als Email und „admin“ als Passwort ein. Dann klickt der Benutzer auf den „Log In“ Knopf.

     Reaktion Erfolgreich eingeloggt. Der Browser wird zur Newsfeedseite des Benutzers umgeleitet.

  Diese Testfallszenarien eignen sich als Startpunkt für den Entwurf, und werden bei der internen Abnahme benutzt um das Produkt zu validieren.

5.2 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

5.3 Inhalt der Präsentation

Die Präsentation muss nicht das komplette Pflichtenheft abbilden oder gar ersetzen. Konzentriert euch auf die Aspekte, die technisch interessant sind oder euer Produkt von den anderen abheben.

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

6 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; evtl. großformatiges Klassendiagramm

6.1 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. Was hier *nicht* reingehört sind private Felder und Methoden. Das sind Implementierungsdetails.
- Beschreibung von charakteristischen Abläufen anhand von Sequenzdiagrammen. Beispielsweise bieten sich Testszenarien aus dem Pflichtenheft hier an. Wir empfehlen Sequenzdiagramme möglichst früh zu erstellen, denn dabei werden die Schnittstellen zwischen Packages und Klassen klar.
- Mit Blick auf den Implementierungsplan: Aufteilung in Klassen/Pakete, die unabhängig voneinander implementiert und getestet werden können.
- Änderungen zum Pflichtenheft, z.B. gekürzte Wunschkriterien.
- Vollständiges großformatiges Klassendiagramm im Anhang. Ausschnitte/Teile können bereits vorher verwendet werden, um Teilkomponenten zu beschreiben. Assoziationen zwischen Klassen dabei bitte mit entsprechenden Pfeilen darstellen, statt nur durch Feldtypen. Ein kleines Beispiel ist in Abbildung 1 gezeigt.
- Identifikation von Entwurfsmustern um Struktur gröber zu beschreiben.
- Erfahrungsgemäßer Umfang:
  - 100 Seiten, primär Klassenbeschreibungen
  - 40–80 Klassen ohne Interfaces
- Möglicherweise weitere UML-Diagrammarten?
- Formale Spezifikation von Kernkomponenten?

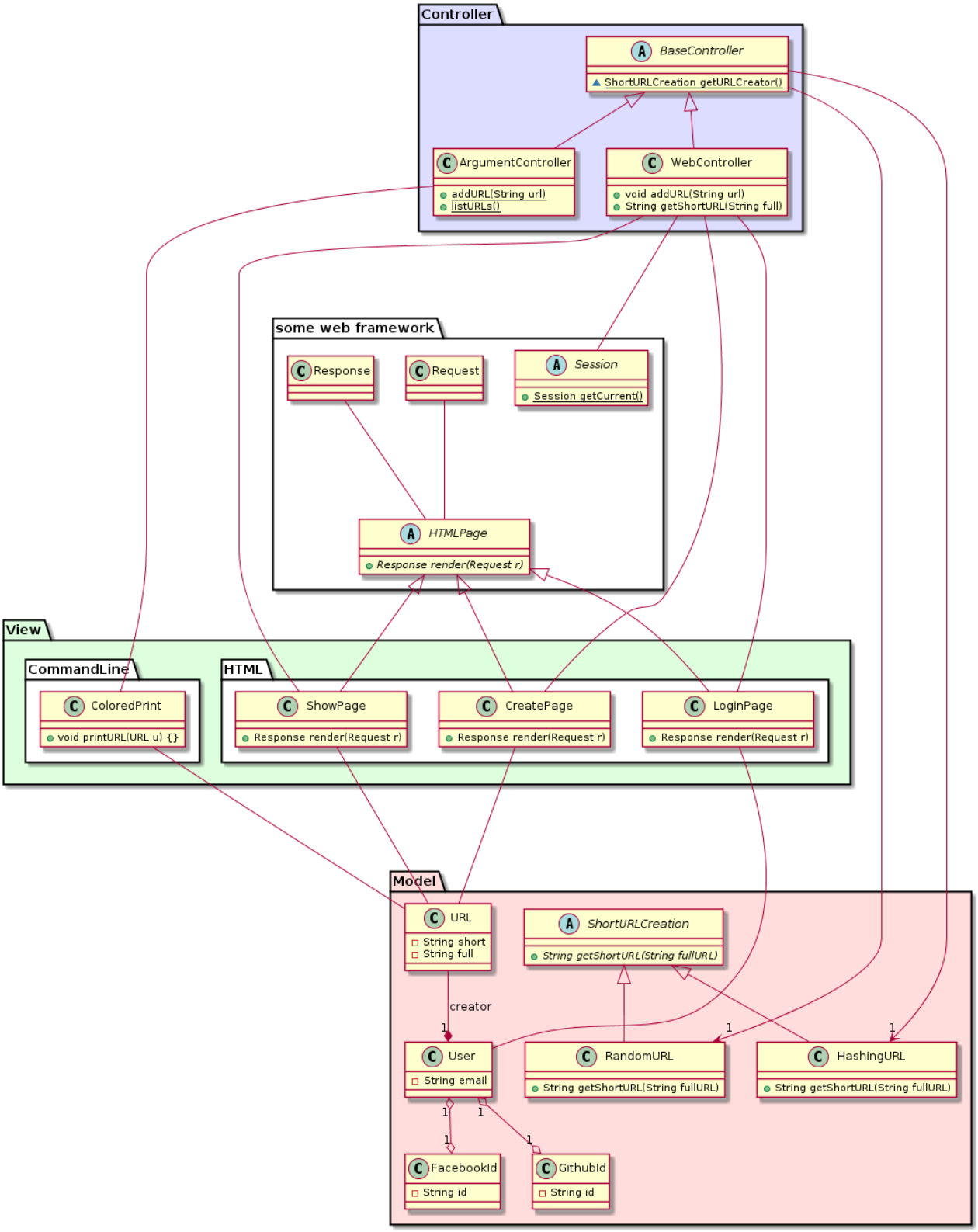

Abbildung 1: Klassendiagrammbeispiel: Grobe Paketstruktur farblich hervorgehoben. Ausgewählte Verwendungen als Assoziation eingetragen, um auch Beziehungen durch Parameter oder lokale Variablen zu verdeutlichen.

Unserer Erfahrung nach ist die Entwurfsphase die schwierigste, da Studenten hier so gut wie keine Erfahrung haben (im Gegensatz zur Implementierungsphase). Deswegen hier ein paar konkrete Schritte, die helfen sollen einen Anfang zu finden.

1. Sucht alle relevanten Substantive im Pflichtenheft heraus. Viele davon lassen sich direkt als Klassen abbilden. Zum Beispiel: Button, Level, Spielfigur, Bild, etc.

2. Was kann man mit diesen Objekten tun? So erhält man die Methoden. Beispielsweise kann man ein Bild drehen, anzeigen und umfärben.

3. In welcher Beziehung stehen die Objekte miteinander? So erhält man die Pfeile im UML Diagramm. Zum Beispiel weiß eine Spielfigur vielleicht in welchem Level sie ist.

4. Spielt die Testfallszenarien durch. So finden sich noch fehlende Methoden und Assoziationen. Außerdem könnt ihr dabei schon die ersten Sequenzdiagramme sammeln.

5. Organisiert die Klassen sinnvoll in Gruppen. So erhält man die Paketstruktur. Beispielsweise Model-View-Controller könnte hier sichtbar werden.

6. Diese ersten Schritte kann man erstmal zügig in einer Sitzung mit dem ganzen Team machen. Anschließend kann man die Pakete oder Klassen einzelnen Personen zuordnen, die dann eigenverantwortlich die Details ausarbeiten.

7. Nun ist die erste Woche rum. Man hat einiges an Material produziert und kann es dem Betreuer zeigen.

8. Jetzt ist es auch an der Zeit mit dem eigentlichen Dokument zu beginnen. Inbesondere sollten technische Fragen geklärt sein. Womit erstellen wir unsere UML Diagramme? Automatisches Erzeugen von Java-Code mit JavaDoc-Kommentaren und daraus LaTeX und PDF erzeugen? Das alles sollte man in der zweiten Woche zum Laufen bringen.

9. Da man nun bereits ein Dokument hat, kann man grundlegende Entwurfsentscheidung sofort niederschreiben. Sammelt erstmal alles in dem Dokument. Ordnen und polieren kann man das später in der Phase.

10. Ein weiterer großer Brocken ist die Anbindung an die Platform, die man benutzt. Beispielsweise Qt, Android oder ein anderes Framework. Hier ist es oft ratsam ein paar minimale Beispielanwendungen zu bauen, um ein Gefühl für die API zu bekommen.

11. Nach zwei Wochen sollten die meisten Klassen entworfen sein. Allerdings hat üblicherweise jedes Teammitglied seinen Bereich für sich bearbeitet. Nun wird es Zeit sich mit der Integration zu beschäftigen. Nehmt euch noch einmal die Testfallszenarien aus dem Pflichtenheft vor und geht diese anhand des Klassendiagramms detailliert durch. Wer ruft wen mit welchen Argumenten auf? Üblicherweise fallen dabei viele Details auf, wo man mehr Informationen weitergeben muss.

12. Auch nicht zu vergessen sind Dinge die nicht im UML auftauchen. Bilder, SQL-Schema, JSON-Schema, Tools wie ein Leveleditor, etc.

13. Nun sind drei von vier Wochen rum und eigentlich sollte der Entwurf so ziemlich vollständig sein. Jetzt ist genügen Zeit für Feinschliff, Präsentation und Konsistenzprüfung.

6.2 Bewertung eines Entwurfs

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

- Dazu gehört: Behalten Klassen ihre internen Datenstrukturen für sich? Eine Klasse, die eine Liste von Objekten verwaltet, sollte selbst Methoden zum Hinzufügen, Löschen, u. s. w. bereitstellen. Sie sollte *keine* veränderbare Referenz auf die Liste nach außen geben und ihre Aufrufer die Liste verändern lassen. Für Java siehe z. B. Unmodifiable View Collections.

- 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.

- Keine indirekten Abhängigkeiten? Wenn eine Abhängigkeit zwischen Objekten sein muss, soll sie direkt und explizit sein. Sich stattdessen an Referenzen entlangzuhangeln sieht auf dem Papier aus wie lose Kopplung, koppelt aber tatsächlich mehr Klassen enger aneinander. 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-orientiertem gegenüber prozeduralem Entwurf, der durch Vererbung und Polymorphie erreicht wird. Siehe auch Open-Closed Principle.

- Liskovsches Substitutionsprinzip bei Vererbung erfüllt? Unterklassen sollten 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 Änderung der Spezifikation sollte nur lokale Änderungen benötigen. Das impliziert: Eine Klasse/Paket/Methode sollte für sich verständlich sein, ohne dass Kontext notwendig ist.

- Sind Methoden frei von unerwarteten Nebeneffekten? Eine Methode, die Informationen aus

einem Objekt zurückgibt, sollte nichts Wesentliches am Objektzustand verändern. Auch eine Methode, die dazu dient, den Objektzustand zu verändern, sollte nur eine überschaubare Änderung durchführen. Siehe auch Command-query separation

Ähnlich zu den üblichen Entwurfsmustern gibt es auch Anti-Entwurfsmuster. Also Muster, die man im Entwurf erkennen kann, die praktisch immer zu Problemen führen. Ein paar Beispiele, die in vergangenen PSE-Projekten auftraten:

- God Object. Wenn zuviel Funktionalität in eine Komponente (Klasse) gesteckt wird. Es verletzt Lokalitäts- und Geheimnisprinzip.
- Anemic domain model. Wenn das Model praktisch nur noch Datenspeicher ist. Objekt-Orientiertes Design zeichnet sich gerade dadurch aus, dass Daten *und* Verarbeitung in Objekten zusammengebaut werden. Objekte, die nur zur Datenhaltung da sind, erzwingen prozedurale Programmierung.
- `switch` und `instanceof`. Sieht man im Entwurf zwar noch nicht direkt, ist aber manchmal absehbar. Dynamische Bindung ist meistens die bessere Wahl.

6.3 UML Diagramme

- UMLet noch ein UML Tool
- Umbrella um einfach nur Diagramme zu zeichnen
- BOUML noch ein UML Tool
- UMLGraph für Versions-Control-freundliche UML Diagramme. (Tipp: Alles in eine Datei und erst in der Implementierungsphase aufspalten)
- PlantUML noch ein Tool für Versions-Control-freundliche UML Diagramme. Allerdings eigene Syntax, so dass kein Java-Code daraus erzeugt werden kann.
- ObjectAid UML Explorer for Eclipse ein Source-to-Diagram plugin.
- Eclipse Papyrus
- Visual Paradigm
- ArgoUML um auch Code zu generieren und beim Entwurf zu helfen (Vorsicht: keine „Undo“-Funktionalität)

6.4 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.
- Game Programming Patterns Buch zu Design Patterns in der Spieleprogrammierung
- Example Toolchain zum Wiederverwenden
- How to Write Doc Comments for the Javadoc Tool mit einigen Tipps und Beispielen.

6.5 Inhalt der Präsentation

- Gerade für UML-Diagramme wäre eine unkonventionelle Präsentation mit Prezi einen Versuch wert. Man kann ein großes allumfassendes Klassendiagram zeigen und dann in Packages und Klassen reinzoomen.
- Kurze Einführung und Verbindung zum Pflichtenheft
- Ein Sequenzdiagram als anschauliches Beispiel mit Ausschnitten aus dem Klassendiagramm. Hier kann man ein Testfallszenario aus dem Pflichtenheft auswählen.
- Überblick über das Gesamtklassendiagramm, Pakete, Module geben. Hier kann man über die Verwendung von Entwurfsmustern erzählen. Das **Klassendiagramm** sollte den Kern der Präsentation bilden, also sollte man hierfür auch die meiste Zeit aufwenden.
- Ein großformatig ausgedrucktes Klassendiagramm vermittelt einen Eindruck von der Gesamtarchitektur und ist für die folgende Diskussion nützlich.
- Einhaltung softwaretechnischer Prinzipien zeigen (z.B. Kohäsion, Lokalitätsprinzip, etc.)
- Gestrichene Wunschkriterien können, aber müssen nicht erwähnt werden. Man muss hier aufpassen, dass kein negativer Eindruck zurückbleibt. Für manches gibt es gute Gründe, aber oft ist es geschickter Streichungen nur im Dokument zu erwähnen.
- Verwendete externe Resourcen (Bilder, Frameworks, Bibliotheken, Sounds, Musik, etc.)
- Nur kurz erwähnen weil eher langweilig: eigene Formate, Datenbankschemata, Einstellungen, Menüstruktur und Dialoge.

7 Implementierung

Talk is cheap. Show me the code.

(Linus Torvalds)

Artefakt der Phase Implementierung: Implementierungsplan zu Beginn, Implementierungsbericht als PDF Dokument und vollständiger Sourcecode

7.1 Plan

- Klarer zeitlicher Ablauf der Implementierungsphase, um frühzeitig Verzögerungen zu bemerken.
- Abhängigkeiten aus dem Entwurf müssen beachtet werden. Wo ist der Critical Path?
- Wie können Teile der Anwendung möglichst früh getestet werden? Braucht es dafür Stub-/Mock-Klassen?
- Klare Aufgabenverteilung im Team. Dabei muss eine faire Verteilung und die Abhängigkeiten beachtet werden.
- Als Form bietet sich ein GANTT Chart an, wie das Beispiel in Abbildung 2. Verpflichtend ist es aber nicht. Zum Beispiel lässt sich auch eine Tabelle oder `dot` benutzen.
- Einzelne Jobs kann man Klassen oder Packages zuordnen, aber häufig ist das nicht möglich. Zwischen vielen Klassen gibt es zirkuläre Abhängigkeiten. Manche Aufgaben erfordern nur Teile von Klassen. Da bietet es sich an, gröbere Aufgaben festzulegen. Auch könnte man die Testszenarien als Aufgabenbeschreibung nutzen.
- Zu *Beginn* der Implementierungsphase (d.h. nach 1-2 Tagen) beim Betreuer abzugeben.

7.2 Bericht

- 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. Gruppiert (und zusammengefasst) werden sollte nach dem Grund für die Änderung und nicht nach der geänderten Klasse.
- Welche Muss- und Wunschkriterien sind implementiert?
- Welche Verzögerungen gab es im Implementierungsplan? Kann beispielsweise als zweites GANTT Diagramm am Ende dargestellt werden.
- Übersicht zu Unittests

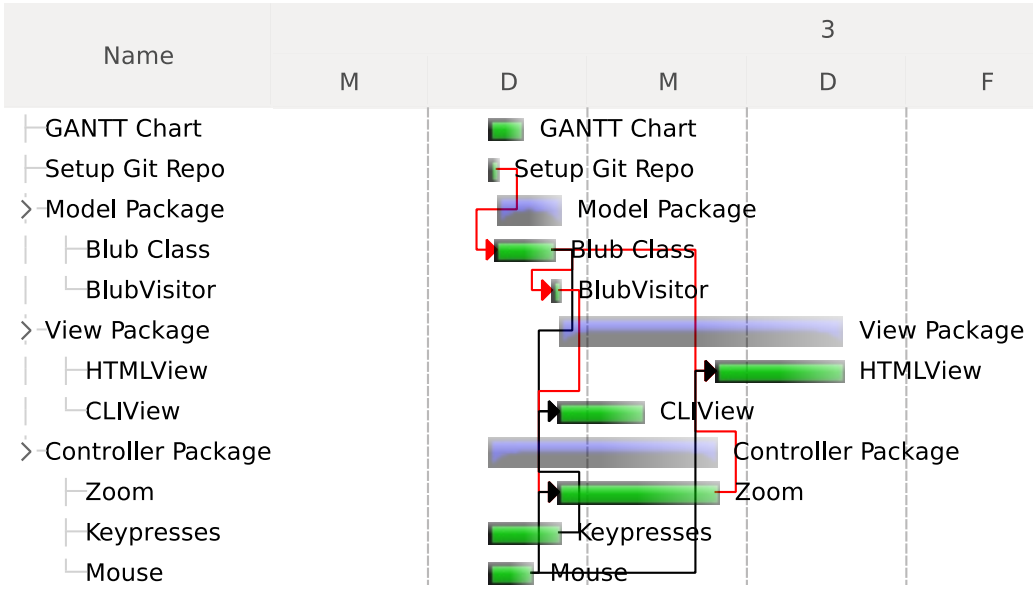

Abbildung 2: Beispiel GANTT Chart: Jede einzelne Klasse ist als Aufgabe eingetragen. Abhängigkeiten sind als Pfeile erkennbar. Der kritische Pfad ist in rot gekennzeichnet.

7.3 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.

7.4 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 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, eine Warnung anzuzeigen wenn eine Klasse `equals` überläd aber nicht auch `hashCode`.
- 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?“.

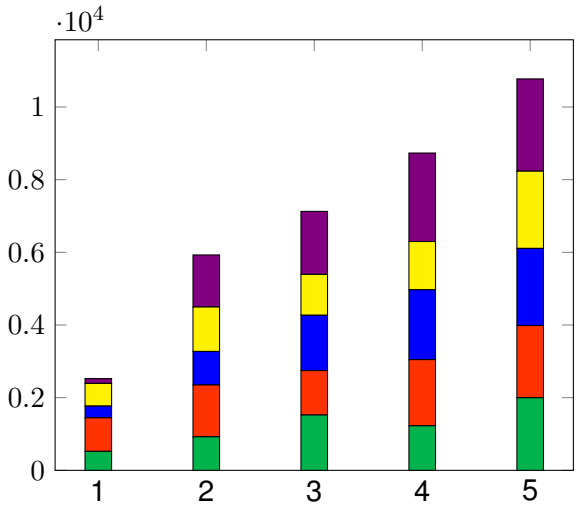

Abbildung 3: Anzahl Codezeilen je Person über 5 Wochen hinweg. Diese Art von Graph zeigt dass die Verteilung im Team fair aussieht. Man sieht auch das kontinuierliche Wachstum.

- Exceptions (nicht RuntimeException) für Fehler beim Aufrufer. Assertions für interne Konsistenzfehler bzw. um Annahmen explizit zu machen.
- Keine Magic Numbers. Explizite Zahlen im Quellcode sind entweder selbsterklärend oder durch benannte Konstanten zu ersetzen. Am besten immer letzteres.

7.5 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.
- Entwicklungsmodell. Wie wurde im Team kommuniziert? Wie wurde Git benutzt (Gab es Feature-Branches? War das Mergen in den master-Branch beschränkt? . . .)

8 Qualitätssicherung

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

Artefakt der Phase Qualitätssicherung: Testbericht

- Überdeckung der Unittests maximieren. (Siehe bspw.: JaCoCo) Wenn GUIs im Spiel sind, ist 100% Überdeckung üblicherweise nicht möglich, aber 90% sollten schon angestrebt werden.
- Komplette Testszenarien aus dem Pflichtenheft durchgehen. Ein Testdurchlauf sollte möglichst automatisch ablaufen.
- Monkey Testing für einfaches Testen der GUI. Der Ertrag ist erfahrungsgemäß nicht sehr groß, also nicht zuviel Aufwand hineinstecken.
- Lint und andere statische Werkzeuge recherchieren. Codequalität kann gar nicht gut genug sein. Ein weiteres Tool wäre SonarQube oder PMD.
- 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. Schema im Bericht: Fehlersymptom, -grund und -behebung
- Erfahrungsgemäßer Umfang: ca. 10–20 Seiten

8.1 Interne Abnahme

Von der Phaseneinteilung her gehört die interne Abnahme zur Qualitätssicherung. Terminlich ist sie entweder zusammen mit dem Kolloquium der Qualitätssicherung oder kurz danach. Hier ist der Code nun endgültig „fertig“, eure Anwendung wird danach bewertet, was jetzt läuft.

Vollständigkeit der Abgabe ist das Wichtigste. Der Sinn dahinter ist, dass euer Betreuer verpflichtet ist Prüfungsunterlagen 5 Jahre lang aufzubewahren. Da niemand so genau weiß, was das im Fall von PSE bedeutet, wird am einfachsten ALLES archiviert.

- Sourcecode (TeX, Java, Makefile, C++, Python, HTML, etc.): vermutlich einfach der Inhalt eures Versionskontrollsystems.
- Artefakte (compiliertes Programm, Dokumente, etc.). Zwischenstufen (TeX .aux Dateien) kann man sich sparen, andererseits sind die auch nicht so groß.

Außerdem ist das der Zeitpunkt, zu dem euer Betreuer zum letzten Mal euer Produkt testet. Falls es aufwendiger ist es lauffähig zu bekommen (bspw. auf ein Androidgerät laden) am einfachsten ein Gerät mit fertiger Software mitbringen.

9 Abschlusspräsentation

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

Artefakt der Phase Abschlusspräsentation: Präsentationsfolien

Showtime!

9.1 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.

Allerdings soll es kein reiner Werbevortrag sein, schließlich präsentiert ihr trotzdem vor akademischem Publikum. Es sollten auch folgende Informationen in den Vortrag:

- Kurze Statistik: Wieviel Zeilen Code, wieviele Commits und Tests, wieviel Überdeckung?
- Einblick in die Softwaretechnik: Interessante Fakten aus den vorherigen Phasen. Kernkomponenten skizzieren. Wie fandet ihr „Wasserfall mit Rückkopplung“?
- 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.
- Die ganze Gruppe: Mindestens sollten alle Namen auf den Folien stehen. Alle Teammitglieder sollten sich mal zeigen. Vielleicht können sogar alle sinnvoll in die Präsentation eingebunden werden?
- Welche Tools hab ihr für die verschiedenen Aufgaben benutzt und wie gut fandet ihr sie?
- Zeitrahmen sind 15 Minuten plus Fragen.

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.

Wie bereits gesagt, seid ihr nicht gezwungen den KIT Folienstil zu verwenden. Lasst euch stilistisch also woanders inspirieren.

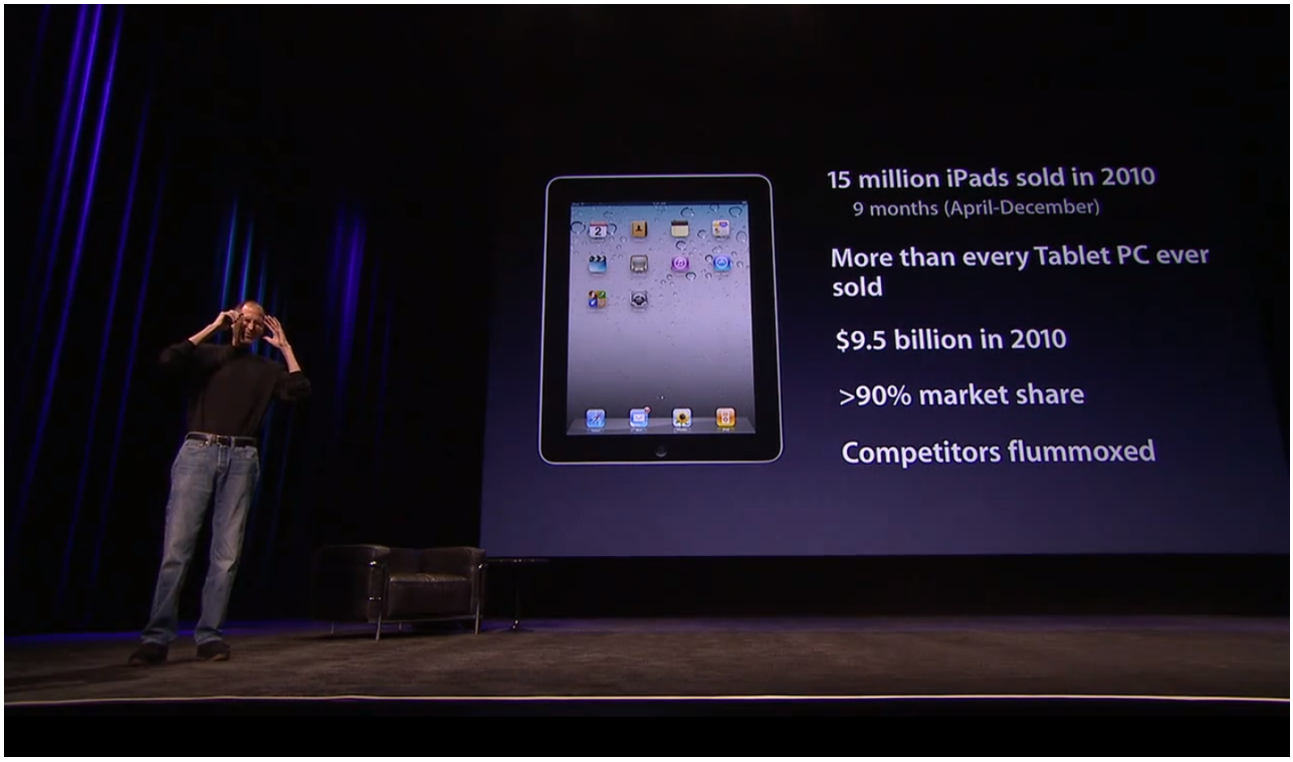

Image source: http://abovethecrowd.com/2015/07/07/in-defense-of-the-deck/

9.2 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

10 Probleme im Team

Am schönsten ist PSE, wenn alle Teammitglieder hochmotiviert sind, sodass alle ihren Beitrag leisten. Manchmal läuft es auch so. Manchmal kracht es aber auch.

An vorderster Front und bewusst in der Verantwortung steht der Phasenverantwortliche. Falls dieser ein Problem nicht lösen kann, eskaliert es zum Betreuer. Falls dieser ein Problem nicht lösen kann, eskaliert es zur PSE-Organisation. In jedem Fall hofft man aber natürlich, dass eine Eskalation nicht notwendig ist.

10.1 Wie vermeidet man Probleme?

Ein wichtiger Punkt ist klare Erwartungen auszusprechen. Besprecht in der Gruppe Fragen wie die Folgenden:

- Welche Note/Teilnote erwarte ich? Ist eine 1,0 das klare Ziel oder bin ich schon mit 1,7 zufrieden?
- Welche Aufgaben müssen zu welchen Deadlines erledigt sein? Sollte der Phasenverantwortliche wöchentliche Deadlines festlegen?
- Wie hart darf man mich kritisieren? Studenten kommen oft aus sehr unterschiedlichen Kulturen, wo Fleiß, Ehre, Respekt oder Pünktlichkeit ganz anders bewertet werden.
- Wieviel Zeit kann ich wann für PSE aufbringen? Es geht dabei nicht nur um Zeitplanung. Häufig investieren PSE-Teilnehmer mehr in ihr Projekt als gefordert, falls nur einzelne aus dem Team das tun, kann es sich negativ auf die Gruppenmoral auswirken.

Es kann hilfreich sein einige Regeln schriftlich festzuhalten. Empfehlenswert ist dabei ein Minimum an Zeremonie, indem beispielsweise jedes Teammitglied das Regelwerk unterschreibt.

10.2 Was tun bei Problemen?

Häufigstes Problem ist, dass ein oder mehrere Teammitglieder kaum mithelfen. Die Aufgabenverteilung wird unfair und diejenigen, die die Arbeit machen, sind frustriert.

Wichtig ist als erstes Fakten zu sammeln und aufzuschreiben. Wer hat wann welche Aufgabe mit welcher Deadline zugeteilt bekommen? Wer hat wann wieviel Stunden gearbeitet? Welche Deadlines wurden wie schlecht oder gar nicht eingehalten? Man braucht konkrete Fakten. Die Aufgabenverteilung kann der Phasenverantwortliche kontrollieren. Stundenzettel muss jeder für sich führen.

Nun bespricht man diese Fakten. Fokus ist primär die Suche nach einer gemeinsamen Diskussionsgrundlage. Sind wir uns einig, was wirklich geschehen ist? Hier ist Vollständigkeit wichtig. Bei stilleren Studenten ist Nachbohren mit Fingerspitzengefühl notwendig. Sind wir uns einig, dass es so nicht weitergehen soll? Falls nicht, wird eskaliert. Falls ja, hat man gemeinsames Ziel und kann gemeinsam an einer Lösung arbeiten. Falls man keine Lösung findet, sollte man nach Hilfe suchen.

11 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? Warum nicht mehr? Warum nicht weniger?

when you don't create things, you become defined by your tastes rather than ability. your tastes only narrow & exclude people. so **create**.

— Why The Lucky Stiff