HSR Engineering-Projekt

Daniel Keller

Anleitung, v1.2, Mai 2018

Published electronically, first publication at HSR, February 2018

Contents

1. Motivation und Ziele

Viele Universitäten und Hochschulen haben das Fach Informatik im Angebot. Aber nur wenige unterrichten Software Engineering in nennenswertem Umfang – und auch dann beschränken sie sich meist auf einige Aspekte wie Scrum oder UML, ohne deren praktische Umsetzung in realistischem Massstab mit den Studenten zu üben.

Im Studiengang Informatik an der HSR gehört das Fach Software Engineering seit über 20 Jahren zu den Kernthemen. Das Modul 'Engineering-Projekt' (EPJ) zählt daher auch zu den Pflichtfächern. Das Engineering-Projekt wird üblicherweise im vierten Semester des Vollzeitstudiums absolviert, nach dem einführenden Unterricht in Software Engineering im dritten Semester (SE1, zwei Stunden Vorlesung plus zwei Stunden Übung pro Woche) und parallel zu Software Engineering 2 im vierten Semester (SE2, ebenfalls 2+2 Stunden/Woche). Für das Engineering-Projekt gibt es 4 ECTS, und es wird erwartet, dass jede/r Student/in während der 14 Semesterwochen ein Pensum von 120 Stunden Arbeit in das Projekt investiert.

1.1 Motivation

> Engineering-Projekt: Studenten lernen, ein kleines Software-Projekt in einem Team aus drei bis fünf Personen während eines Semesters von A-Z durchzuziehen: Requirements, objektorientierte Analyse und Design, Architektur, Programmierung, Metriken, Test, Abnahme. Dabei werden die Techniken und Methoden aus den vorangegangenen Kursen in einem (fast) realen Umfeld angewendet und somit besser verstanden. ■

Dieses praktische Projekt ist für viele Studierende ein Schlüsselerlebnis: sie arbeiten oft zum ersten mal in einem Team, sie programmieren vielleicht zum ersten Mal ein grösseres Stück Software, sie strukturieren und planen zum ersten Mal ein Software-Projekt unter verschiedenen Engineering-Aspekten, und sie lernen, sich besser einzuschätzen, indem sie während des ganzen Projektverlaufs über die aufgewendeten Zeiten Buch führen.

In diesem Unterrichts-Modul versuchen wir, den Student/innen die Erfahrung zu vermitteln, welche der theoretisch vermittelten Methoden und Werkzeuge im Praxiseinsatz unter welchen Voraussetzungen gut funktionieren und welche wann nicht.

1.2 Ziele

Das Ziel für das Engineering-Projekt ist, einen iterativen Software-Entwicklungs-Prozess an einem realem Beispiel von A-Z durchzuziehen, und dabei die folgenden Techniken einzusetzen lernen:

- Anforderungsspezifikation erstellen, um den Funktionsumfang (Scope) zu definieren
- OO-Analyse, OO-Design einsetzen
- Architektur definieren und dokumentieren
- Programmieren im Team (mit Versionskontrolle und Build Server)
- Tests & Reviews (Unit Test, Systemtests) planen und durchführen
- Projekt-/Zeitmanagement lernen
- Aufwandschätzung lernen (vorab schätzen, Zeit aufschreiben, Soll/Ist-Vergleich machen)
- Zusammenarbeit im Team üben
- gute, adäquate Dokumentation erstellen (für grösseres Team und grösseres Projekt)

1.3 Vorgaben für das Engineering-Projekt

Die folgenden Vorgaben sind für alle Teams gegeben und müssen eingehalten werden:

- Dauer: ganzes Semester, d.h. 14 Wochen, bzw. im Frühjahrssemester 15 Wochen wegen Feiertagen/Ferien.
- Teamgrösse: 3 bis 5, alle programmieren mit
- Zeitrahmen: 4 ECTS = 120 Stunden pro Teammitglied, d.h. 8.6h pro Person und Woche (oder 8h x 15 Wochen)
- Mindestens 4 Lektionen gemeinsame Zeit (ganzes Team!) pro Woche, sonst funktionieren die Zusammenarbeit und die Reviews mit dem Betreuer nicht.
- Selbstgewählte Aufgabe, welche vom Betreuer genehmigt werden muss, *kein* externer Kunde.
- Prozess: iterativer, inkrementeller Prozess (Phasen des RUP, Iterationen in Construction-Phase agil nach Scrum)
- Einsatz der gelernten Engineering-Techniken (OOA, Testen, etc. siehe unten)
- Environment: Code-Repository (git, SVN) plus automatischer Build auf CI Server; Redmine (oder JIRA, TFS) für Arbeitspakete und Zeitaufschreibung; die HSR stellt einen git-Server plus pro Team einen virtuellen Server mit vorinstalliertem Redmine und Jenkins zur Verfügung
- Als Programmiersprache sollen Java, C#, eventuell C++, TypeScript oder Python eingesetzt werden
- Jedes Team kriegt einen fest zugewiesenen Betreuer
- Regelmässige Reviews mit Betreuer (mindestens die fünf vorgegebenen Reviews), bei dem das ganze Team anwesend ist; es gibt eine Teilnote pro Review
- Als Abschluss in der letzten Semesterwoche eine Präsentation vor Publikum mit Folien und Demo, mit dem ganzen Team

Es wird vorausgesetzt, dass alle ein eigenes Notebook als Entwicklungsmaschine haben. Die Entwicklungsumgebung soll (Empfehlung) für alle im Team gleich sein (Eclipse, IntelliJ, Visual Studio o.ä.). Ein git-Server steht zur Verfügung (inkl. Backup), das git-Repoitory können die Teams selbst einrichten: https://git.hsr.ch (über https von intern und extern über VPN zugänglich). Ein Code-Versionierungssystem (git, SVN oder etwas ähnliches) muss verwendet werden.

Pro Team kann ein virtueller Server für die Projekt-Automatisierung beantragt werden, sobald das Projekt genehmigt ist: http://fs-i.hsr.ch/vServerKiosk/
Empfehlung: die Teams können Redmine und Jenkins nutzen (es gibt fertig aufgesetzte Vserver im Kiosk).

Warum die Empfehlung, Dreier- bis Fünferteams zu bilden? Es ist Tatsache, dass erst ab drei Personen eine Gruppendynamik ins Spiel kommt und ein gewisser Koordinationsaufwand getrieben werden muss: zwei Personen = ein Kommunikationskanal, drei Personen = drei Kommunikationskanäle. Somit ist die Mindestgrösse von drei Personen gesetzt. Bei grösseren Teams als Fünferteams besteht die Gefahr, dass Trittbrettfahrer unerkannt mitmachen, da es immer schwieriger wird, die Leistungen den einzelnen Teammitglieder zuzuordnen. Ebenso wächst bei ganz grossen Teams der Koordinationsaufwand unverhältnismässig. Deswegen die Vorgabe, Teams von drei bis fünf Personen zu bilden.

2. Beispiel-Projekte

In der Folge sehen Sie einige Engineering-Projekte aus vergangenen Jahren. Siehe auch Beispielprojekte auf dem Skripteserver: . . . /Informatik/Fachbereich/Engineering-Projekt/

2.1 NetWords

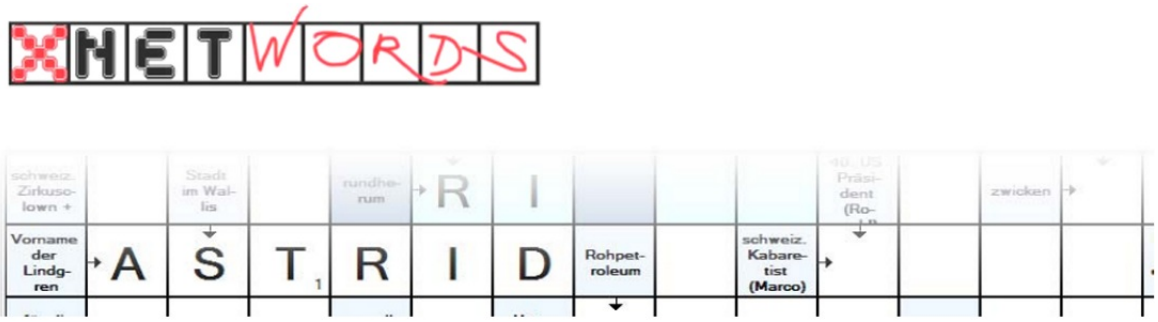


Figure 2.1: Multiplayer-Kreuzworträtsel-Programm

Hier wurde eine Multiplayer Kreuzworträtsel-Anwendung programmiert. Dabei gab es verschiedene Modi: Einzelspieler mit/ohne Zeitbegrenzung, Mehrspieler-Modus mit tickender Uhr plus die Möglichkeit, gegenseitig sich Lösungs-Wörter 'abzukaufen'. Originell und spannend zu spielen.

Wie bei Spielen üblich ergaben sich keine wirklichen Use Cases. Dafür wurden die Spielregeln umso ausführlicher dokumentiert und es wurde auch viel Wert auf eine schöne grafische Darstellung gelegt.

2.2 PopPins

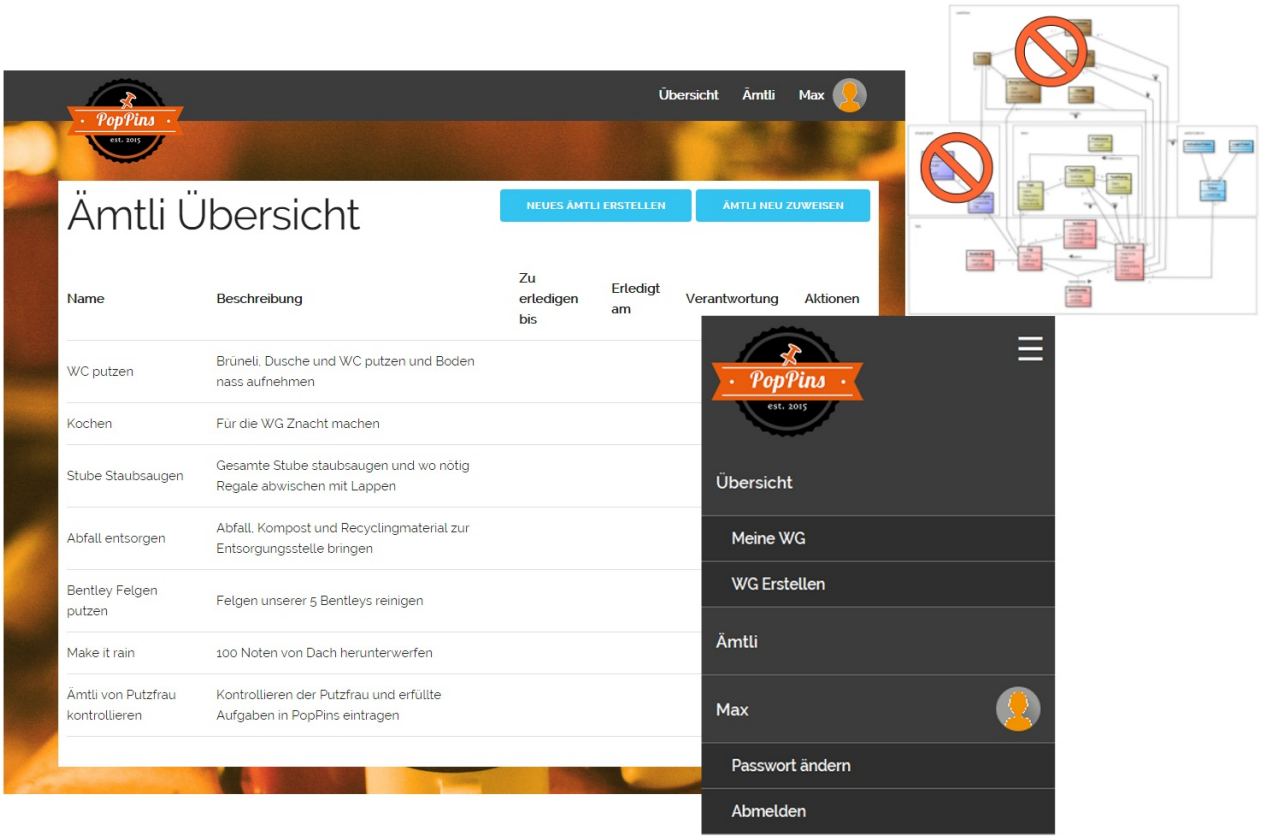


Figure 2.2: PopPins: WG-Verwaltung mit Aufgaben, Bewerungen, Einkaufsliste

Die Arbeit profiterte davon, dass ein Student aus der Gruppe in einer WG wohnte und somit als Endanwender und Erst-Tester fungieren konnte. Dort gab es immer wieder Unstimmigkeiten darüber, wer kochen, putzen oder einkaufen sollte, und wie gut diese Aufgaben dann auch tatsächlich erledigt wurden. Die Software unterstützt die Bildung von WGs, das Definieren und Zuteilen von Aufgaben, die Bewertung der ausgeführten Arbeiten und das Führen einer Einkaufsliste. Die Punkte 'Abrechnung' und Kostenverteilung wurden nicht erreicht.

Diese Aufgabenstellung ergab eine Reihe aussagekräftiger Use Cases, dazu ein schönes Domainmodell, zwar etwas zu gross für ein EPJ. Das Domainmodell wurde geordnet und eingefärbt, das ergab eine gute Orientierung, was in welcher Reichenfolge implementiert werden sollte. Wie schon aus der Grösse des Domainmodells abzusehen war, konnte nicht alles implementiert werden, so aber mit einer funktionierenden Basis, die noch ausgebaut werden kann.

Java, Play Framework mit MVC und O-R Mapper, HTML5 responsive design, Angular JS, less, Ebean ORM

2800 Java LOC, plus HTML, CSS und etwas JavaScript

2.3 TowerDefense

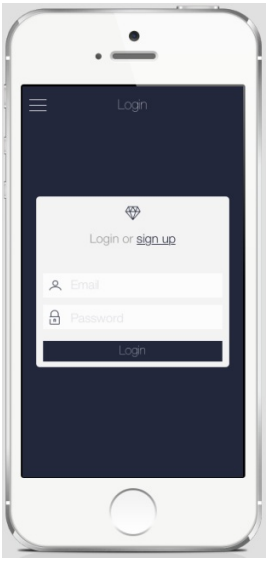
Ein klassisches Spiel: bis zu acht Spieler versuchen jeweils, auf dem vorgegebenen Weg ans Ziel zu kommen. Dabei wird man unterwegs von Computer-generierten Kreaturen attackiert und kann sich mit gesetzten Türmen verteidigen.

Figure 2.3: Mehrspieler-Modus von TowerDefense

Das Spiel hat eine Client-Server-Architektur mit einem selbstgeschriebenen Protokoll von Messages auf Socket-Basis; Verarbeitungs-Queues im Server.

TowerDefense wurde implementiert in Java, mit Slick2d als Grafik-Library.

2.4 "Login App"



In diesem Projekt lief einiges schief: das Team schaffte es nur, den Login-Screen und einen Screen für das Editieren von User-Daten zu implementieren, nur zwei von sieben Punkten konnte abgehakt werden, der ganze interessante Rest wurde nicht fertig.

Diese Gruppe hat:
- falsch priorisiert (Unwichtiges zuerst)
- zuwenig zusammen gearbeitet
- sich in Technologien verzettelt.

Trotz aller Schwierigkeiten bekam das Team eine genügende Note, weil sich alle für eine gute Lösung eingesetzt haben, viele interessante Teilaspekte gut gelöst wurden und alle Zwischenresultate befriedigend bis gut waren. Das Teile passten aber aus verschiedenen Gründen nicht zusammen. Der Lerneffekt für das Team war trotzdem gross, denn manchmal lernt man aus Fehlern mehr, als wenn alles perfekt gelaufen ist.

2.5 Haoki

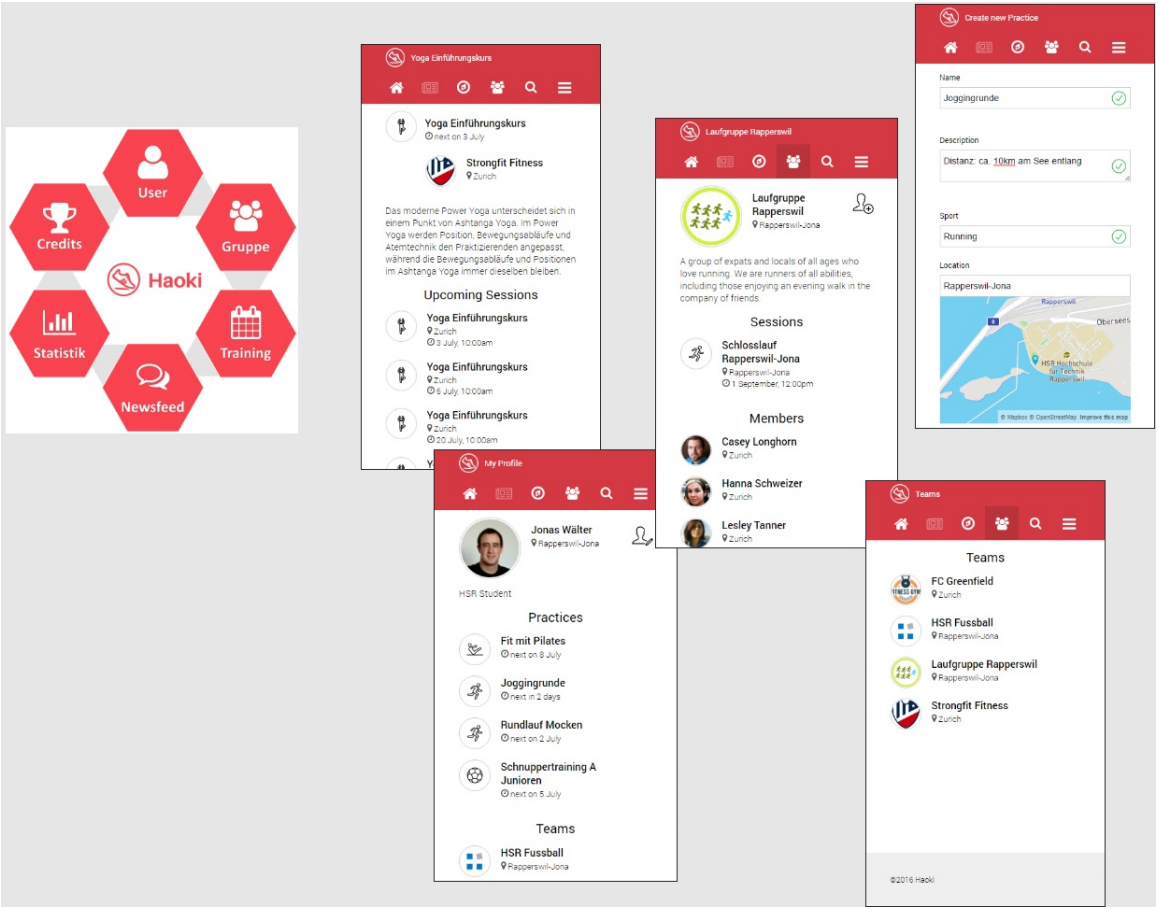


Figure 2.4: Sport/Trainings-App

Haoki: die Spitzenarbeit eines Fünferteams, das ausgezeichnet harmoniert hat. Man kann Trainings ansagen, daran teilnehmen, kurzfristig Mit-Trainierende suchen, Punkte sammeln, einen Gruppen-Trainingskalender pflegen, sehen wer was gemacht hat. Das alles in JavaScript in einem Mobile-freundlichen Design, auch optisch hervorragend umgesetzt. Viele neuartige Tools eingesetzt, dabei alle gesetzten Ziele erreicht, und das exakt mit den geplanten 120 Stunden pro Person, ein kleines Wunder.

Frontend: Mobile First, responsive design, TypeScript, Angular JS, WebPack, Karma, Jasmine, MapBox

Server-side: node.js, TypeScript, PostgreSQL, Jasmine, swagger, github Workflow, Travis CI, AWS mit Load Balancer

2.6 ScrumBoard

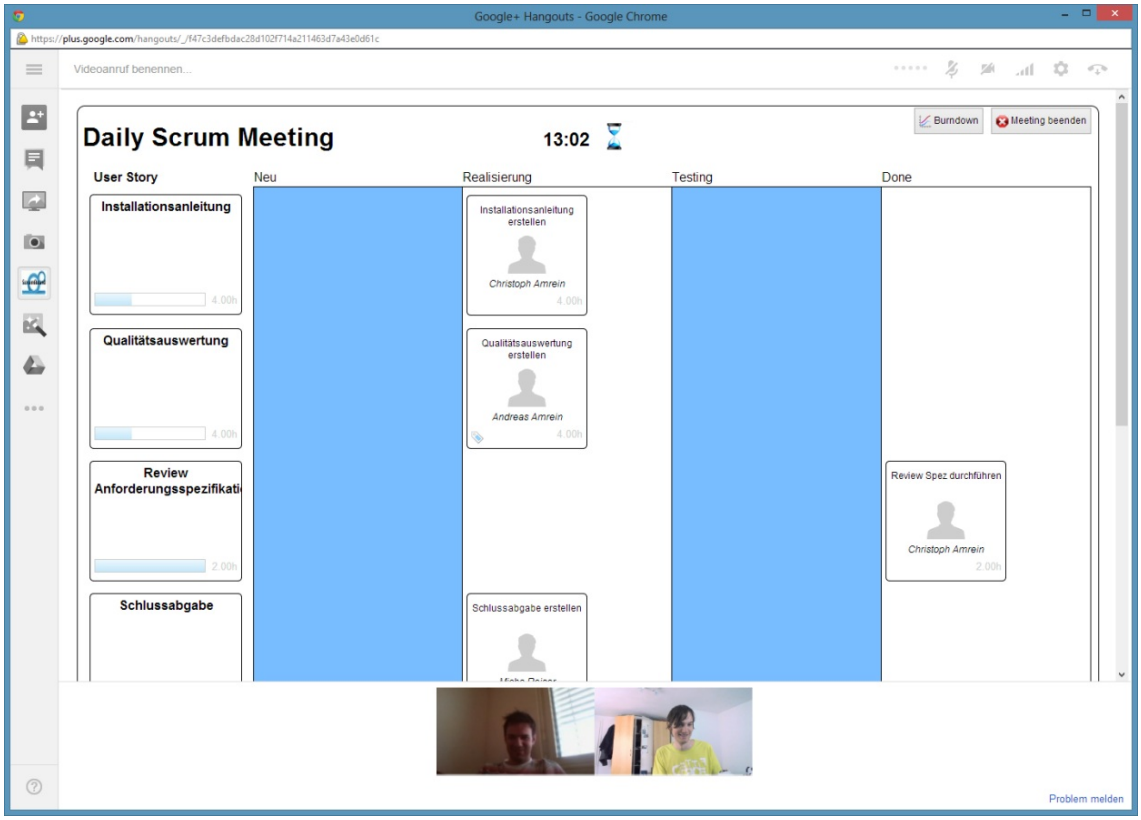


Figure 2.5: ScrumBoard mit User Stories und Google Hangout Chat

Noch eine Spitzenarbeit, diesmal in einem Vierer-Team: mit diesem browserbasierten ScrumBoard kann man User Stories priorisieren, vom Product Backlog in den Sprint Backlog verschieben (drag-n-drop), User Stories editieren und kommentieren und schliesslich als 'fertig' markieren. Zudem gibt es ein Burndown Chart und das Team kann mit dem integrierten Google Hangout chatten. Eine sehr ambitionierte Arbeit mit hochqualitativem Code und perfekt auf die Aufgabe zugeschnittener Architektur. Auch dieses Team hat es geschafft, in der vorgesehenen Zeit alle Ziele zu erreichen – was in den Engineering-Projekten normalerweise nicht gelingt, Hut ab!

6800 Lines of Java Code, Spring Framework, ZK EventBus, Hibernate, slf4j, guava-libraries, RIA Framework, Google OAuth, SonarQube

2.7 StudoGotchi

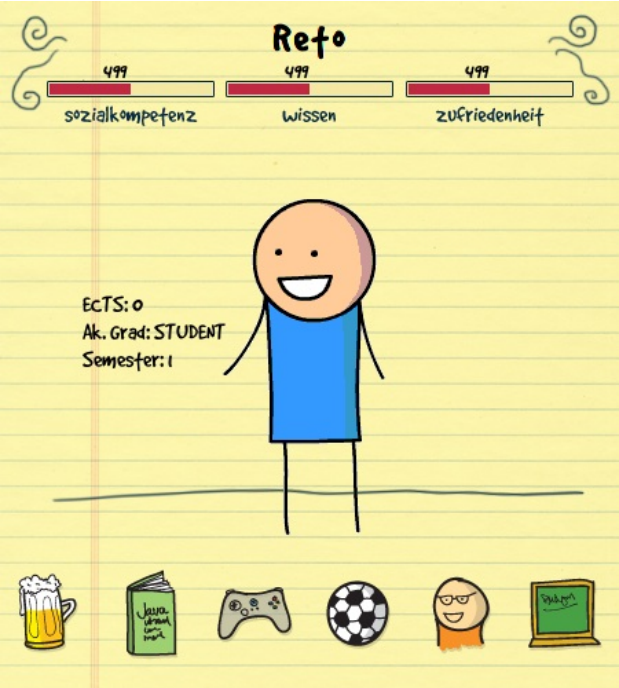


Figure 2.6: StudoGotchi mobile App

Angelehnt an die Tamagotchi-Eier der Neunziger Jahre muss in diesem Programm ein Student herangezogen, gepflegt und glücklich gemacht werden. Ziel: ECTS-Punkte sammeln um akademischen Grad zu verbessern. Challenge: Finde die Balance zwischen Sozialkompetenz, Wissen und Zufriedenheit.

Sehr kreative und optisch überzeugende Umsetzung einer witzigen Idee.
Java, Swing, Hibernate, 3700 Lines of Code

3. Durchführung

3.1 Themenwahl

In der Regel wählt das Team das Thema, welches sie in Software umsetzen werden, selbst. Die richtige Wahl der Projektidee ist enorm wichtig. Zum einen soll das Thema die Studierenden ansprechen, damit es Spass macht (Motivation). Zum anderen darf das Projekt weder zu klein noch zu gross sein. Aus der Lernpsychologie ist bekannt, dass der Schwierigkeitsgrad einer Aufgabe sehr sorgfältig gewählt werden muss, damit eine hohe Motivation entstehen kann. Wählt man die Aufgabe zu leicht, macht sie keinen Spass, weil es so einfach ist. Wählt man die Aufgabe zu schwer, ist der Frust vorprogrammiert, die Teams verrennen sich, die Arbeit wird nicht fertig, kurzum, die Motivation ist weg.

Es empfiehlt sich, die folgenden Punkte bei der Wahl der Aufgabe zu berücksichtigen:

- Umfang klein genug, aber ausbaufähig, d.h. kleiner wichtiger Kern mit mehreren Erweiterungsmöglichkeiten
- Gut mit Use Cases definierbar (Ausnahme: Games)
- Mittel-komplexes Domainmodell (4-12 Klassen)
- Mobile App + Server ist ideal für Architektur-Herausforderungen, ebenso Mehrspieler-Games
- Ohne externen Kunden, weil das erheblichen Mehraufwand bedeutet
- Keine neue, schwierige Technik (z.B. Machine Learning) die viel Einarbeitung erfordert
- objektorientiert – Problemdomain aus einigen (5-10) Klassen
- interessant genug (nicht nur CRUD)

Hinweis: Achtung, das Team soll sich nicht auf potentiell zeitfressende Experimente einlassen. Die Einarbeitung in unbekannte Gebiete (z.B. Spieltheorie, KI) und Technologien (z.B. Angular oder React Native) kann viel Zeit kosten und wird nicht entsprechend honoriert, da es kein vorrangiges Ziel des Engineering-Projektes ist, neuste Technologien oder SE-ferne Themen zu lernen.

3.2 Teamzusammensetzung

Merkmale erfolgreicher Teams:

- Das Team hat viel gemeinsame Zeit pro Woche, mehr als die vorgeschriebenen 4 Lektionen
- Alle sind durch die Aufgabenstellung/Projektidee motiviert
- Die Aufgabenstellung ist den Fähigkeiten des Teams angepasst
- Alle kennen sich schon länger (jeder weiss von des anderen Stärken, und es gibt die blöden Alphatier-Kämpfe nicht)
- Die Teammitglieder wohnen an einem Ort, pendeln dieselbe Strecke
- Die Fähigkeiten der einzelnen Teammitglieder ergänzen sich
- Ein Mix mit viel/wenig Programmiererfahrung funktioniert gut, wenn das Team gut zusammenarbeitet und ein bisschen sozial eingestellt ist
- Alle im Team sind kommunikativ (keine Einzelkämpfer, auch wenn sie genial sind)

3.3 Projektantrag

Template auf Skripteserver, Inhalt (eine A4-Seite):

- Kurzname des Projektes
- Liste der Team-Mitglieder mit eMail-Adressen
- Die für das ganze Team möglichen Zeiten für Reviews und Präsentationen (alle anwesend!)
- Zielsetzung, Motivation
- Aufgabenstellung, kurze Beschreibung (1/2 Seite) der zu entwickelnden Software
- Verwendete Programmiersprache(n), Bibliotheken, Werkzeuge (soweit schon bekannt)

Es ist vorteilhaft, wenn der Projektantrag (u.a. auch in den SE2-Übungsstunden) mit einem Betreuer vorbesprochen wird. Die Abgabe muss an d1keller@hsr.ch bis Freitagabend, 1. Semesterwoche erfolgen.

Zum Anfang der 2. Semesterwoche erhalten die Teams die Genehmigung durch ein eMail; jetzt kann ein VServer für das Team beantragt werden. Die Zuweisung des Betreuers erfolgt durch ein eMail spätestens Mitte der 2. Semesterwoche. Ab dann können über die Einschreibelisten die Termine mit dem Betreuer vereinbart werden.

3.4 Empfohlenes Prozessmodell

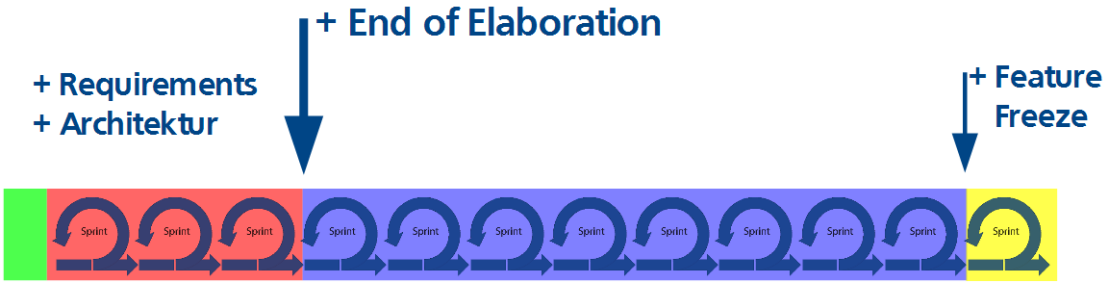


Figure 3.1: Empfohlene Vorgehenseweise: Scrum plus End of Elaboration

Für eine genauere Beschreibung dieses gemischten Vorgehensmodells konsultieren Sie bitte das Skript zu SE1 oder die Vorlesungsfolien SE1.

3.5 Projektplan

Der Projektplan ist ein Text-Dokument von ca. 4-10 Seiten (s. Template), er enthält zur Hauptsache:

- Planung der Meilensteine (Phasen und Iterationen, was wird wann in etwa abgeliefert, wann sind die Reviews eingeplant)
- Festlegung der Werkzeuge und Methoden (IDE, Sprachen, Libraries)
- Angaben zu geplanten Q-Massnahmen und Tests (Coding Guidelines? Unit Tests? Abdeckung? Usability Tests? Code Reviews? ...)
- Abschätzung der Risiken
- Zuständigkeiten und Kommunikationswege im Team
- Wann und wo sich das Team jeweils treffen wird (regelmässig, mind. 1 x pro Woche)

Bereits zum Zeitpunkt der Erstellung des Projektplans muss man sich Gedanken machen, was der Funktionsumfang der Software sein sollte (Scope, definiert durch UCs und Domainmodell), wie die Architektur strukturiert sein müsste (Schichten, Deployment), und wie das User Interface aussehen könnte (UI Entwürfe). Ebenso kann/sollte man mit der Detail-Planung beginnen: man macht sich Gedanken "was muss alles gemacht werden?" (User Stories, Arbeitspakete), am besten mit einem Projektmanagement-Tool wie Redmine oder JIRA, denn Arbeitspakete gehören besser nicht in ein Word- oder Excel-Dokument.

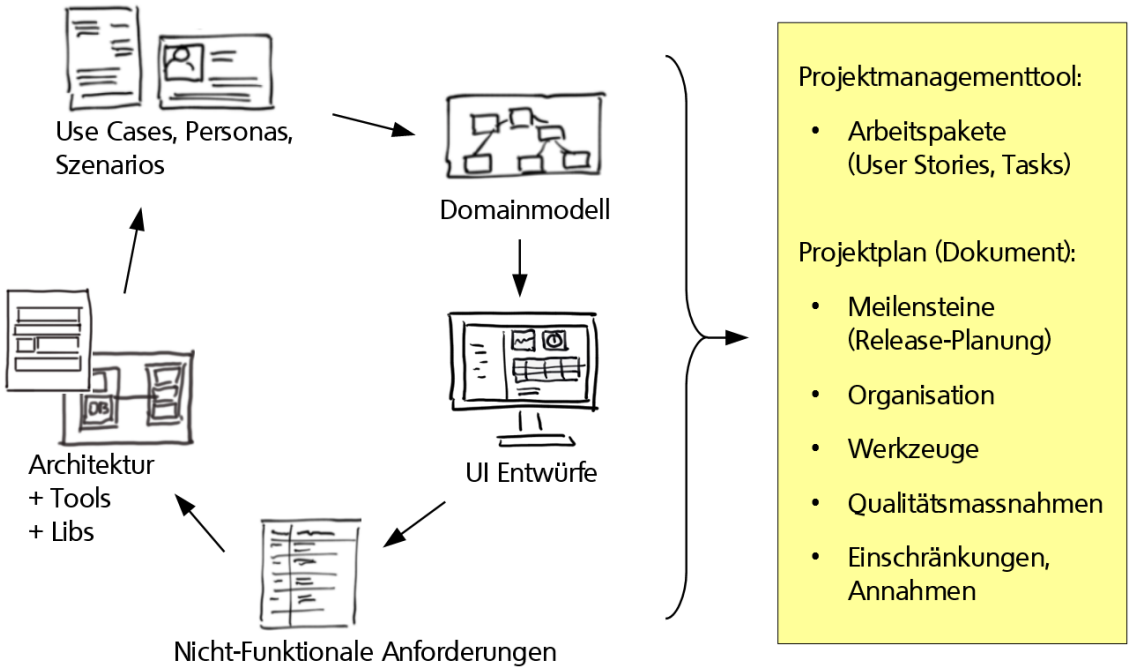


Figure 3.2: Schon früh sollte man die Anforderungen – wenigstens grob – festlegen

Jedes Projekt braucht einige Zielvorgaben, damit alle im Team in dieselbe Richtung steuern. Man muss sich also schon sehr früh um die Anforderungen und um die Architektur kümmern. Darum empfiehlt es sich, schon ganz zu Beginn des Projektes 'die Runde zu machen' (siehe Bild), d.h. die Use Cases zu skizzieren, das Domainmodell zu entwerfen, Ideen zum UI festzuhalten, und das gleich mehrfach im Kreis, damit Lücken und Inkonsistenzen entdeckt und korrigiert werden können. Diese Arbeit dauert nicht lange (wenige Prozente des Gesamtumfangs), aber das Resultat ist sehr wertvoll – und man wird die Richtung im agilen Vorgehen ohnehin noch mehrfach ändern. Es wäre nur ein Fehler, so ganz ohne grobe Ausrichtung einfach loszulaufen.

Im EPJ geht es nicht darum, den Kunden zu verstehen – denn man hat ja keinen – sondern dass das Team sich auf einen Satz von Funktionen einigen kann, im Sinne von "Das machen wir, und das machen wir nicht". Nur so kann man das Projekt vernünftig planen.

Die Abgabe des Projektplans muss bis Ende SW02, Freitag 17h, an den zugewiesenen Betreuer erfolgen. Achtung: dieser Abgabetermin ist knapp, das Team sollte schon in der SW01 mit dem Projektplan beginnen. Der Review des (revidierten) Projektplans mit Notengebung findet normalerweise in der dritten Semesterwoche statt.

3.6 Meilensteine und Reviews

Die Teams setzen die Meilensteine selbstständig – im Rahmen von gewissen zeitlichen Vorgaben (s. weiter unten). Hier sind als Beispiel sieben Meilensteine eingezeichnet, wovon zwei bereits gesetzt sind (M0, M1) und einer noch zwingend gesetzt werden muss (M3):
M0: Abgabe Projekantrag
M1: Abgabe Projektplan
M2: Requirements: UCs, Domain Model, nicht-funktionale Anforderungen, GUI Design
M3: End of Elaboration (s. Checkliste)
M4: Architektur (Schichten, Server Logik, Persistenz, Deployment)
M5: Usability Test (Alpha-Release)
M6: Feature Freeze (Beta-Release)

Die Meilensteine M2, M4, M5, M6 in diesem Beispiel müssen nicht zwingend gesetzt werden, sie sind aber sinnvoll durch die Reviews, die danach angesetzt sind.

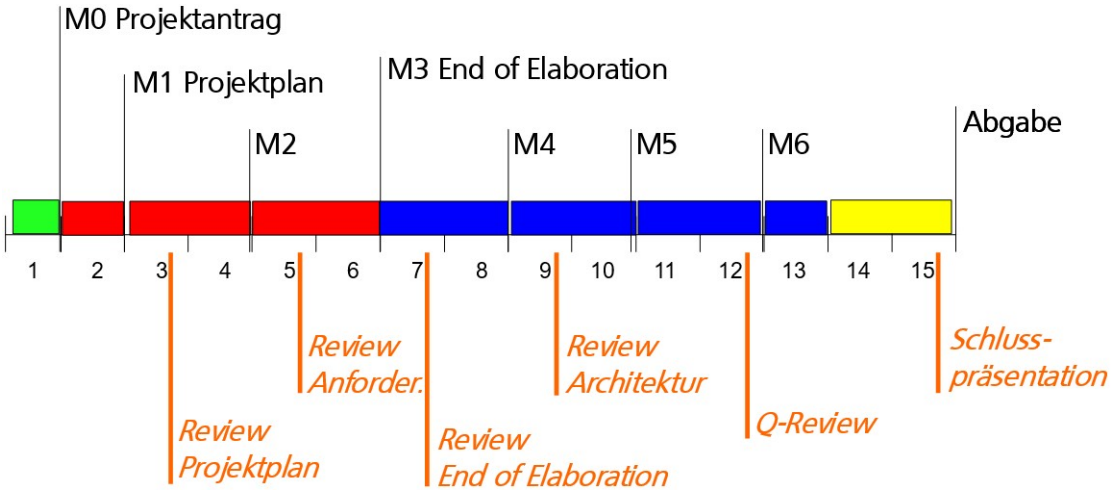


Figure 3.3: Typischer Engineering-Projektverlauf mit Meilensteinen und Reviews

Der Projektplan muss die folgenden sechs Reviews/Präsentationen enthalten (im obigen Bild orange eingezeichnet):

1. Review Projektplan mit Zeitplan und aktuellen Iterationsplänen – Termin: SW03
2. Review der Anforderungsspezifikation und der Domainanalyse – Termin: frei wählbar (empfohlen: vor End of Elaboration)
3. Ende Elaboration (s. Checkliste) mit Architekturprototyp – Termin: wählbar SW05-08
4. Review von Architektur/Design und Architekturdoku – Termin: wählbar bis SW14*)
5. Q-Review: Code-Qualität (u.a. Metriken), Tests und weitere Q-Massnahmen – Termin: wählbar nach End of Elaboration bis SW14*)
6. Schlusspräsentation und Demo der Software: SW15 (genaues Datum wird bekanntgegeben)

Spätester Abgabetermin für alle Quelldaten und Dokumentation: Freitag der letzten Semesterwoche um 17h.

Weitere Meilensteine können nach Bedarf festgelegt werden, ebenso können auch weitere Reviews/Besprechungen mit dem Betreuer vereinbart werden.

*) Es sind nominell 14 Semesterwochen in 15 Kalenderwochen – minus Feiertage (Ostern, Pfingsten, Auffahrt) und Frei-Tage, die Angaben beziehen sich auf die 15 effektiven Wochen.

3.7 Vorgehen an Meilenstein-Review

Bei jedem Review muss sich das Team Zeit beim Betreuer reservieren, und das ganze Team muss beim Review anwesend sein. Die Art der Zeit-Reservation kann von Betreuer zu Betreuer unterschiedlich sein: per e-Mail, immer zu fixen Zeiten, oder via Einschreibeliste auf wiki.hsr.ch. Für einen Review sollten 45 Minuten reserviert werden.

Normalerweise setzt man den Review unmittelbar nach Erreichen des entsprechenden Meilensteins an (siehe Bild 3.3). Üblicherweise wünscht der Betreuer, dass die Dokumentation per E-Mail 48 Stunden vor dem Review zugesandt wird, damit der Review vorbereitet werden kann. Die Dokumentation sollte gemäss Projektplan nachgeführt sein, Teile der Doku können auch Links zu einem (zugänglichen) Server sein.

Während der Reviews bestimmt der Betreuer die Agenda, fragt, hakt nach, kommentiert (siehe auch die entsprechenden Checklisten, Abschnitt 4.3). Die Note für die Review-Resultate erhält das Team spätestens 48 Stunden danach per e-Mail mit Begründung und Erläuterungen.

3.8 Meilensteine in Redmine

Die Meilensteine werden im Projektplanungs-Dokument beschrieben. Es ist nur logisch, wenn die Meilensteine auch im Projektmanagement-Tool (Redmine, JIRA o.ä.) auftauchen. Hier ein Vorschlag für die Umsetzung der Meilensteine in Redmine-Zielversionen; die Arbeitspakete werden dann einer dieser Versionen zugeordnet.

Redmine-Versionen:

MS1: Projektplan-Abgabe
MS2: Anforderungen und Analyse
MS3: Ende Elaboration
Alpha
Beta
End of Construction
Optional Features

Nochmals: Die Reviews finden in der Regel kurz nach den entsprechenden, im Projektplan vom Team gesetzten Meilensteinen statt (siehe Bild 3.3).

3.9 Einschreibelisten für Reviews

Einschreibelisten für Betreuung – falls der Betreuer überhaupt Einschreibungen wünscht – enthalten Slots von 15 Minuten. Einschreibelisten sind online auf Wiki des Moduls: http://wiki.hsr.ch/SE2Projekt einsehbar und editierbar.

Für Beratung bei den Betreuern:
für eine einfache Beratung 1 – 2 Slots buchen

Für Reviews/Präsentationen bei zugewiesenem Betreuer:
für einen gesetzten Review 3 Slots buchen

Slot-Buchungen für ein Team müssen mindestens 24 Stunden im Voraus erfolgen. Sind Slots nicht belegt, können Betreuer spontan, auch ohne Einschreibung beansprucht werden. Nach Absprache sind Termine mit dem Betreuer auch ausserhalb der Slots möglich.

3.10 Hinweise zur Dokumentation

Die Teams können als Dokumenten-Vorlage die Templates von SE1 und auch die Vorlagen und Beispiele auf dem Skripteserver benutzen.

> Achtung: Templates und Beispiele sollten nicht blind verwendet werden, sie sind nicht perfekt und nicht für jeden Fall geeignet, also sind sie gegebenenfalls anzupassen. ■

In den Dokumenten und im Code ist immer die Urheberschaft auszuweisen, d.h. wenn ein Teammitglied z.B. Code von stackoverflow.com kopiert und ein Fragment davon im eigenen Code einbaut, dann soll das entsprechend kommentiert sein. Dasselbe gilt für Texte und Bilder in Dokumentation, z.B. von Wikipedia. Die Benutzung fremder Quellen ohne Angaben führt zu empfindlichen Abzügen.

3.11 Zeiterfassung

Alle Teammitglieder sind verpflichtet, die für das Projekt aufgewendete Zeit zu erfassen. Am einfachsten ist es, wenn die tatsächlich aufgewendeten Stunden direkt auf den Arbeitspaketen im Projektmanagement-Tool (Redmine, JIRA) erfasst werden. Das ergibt dann auch einen unmittelbaren Vergleich zum geschätzten Aufwand.

Die Zeiterfassung sollte auf ca. eine Viertelstunde genau erfasst werden. Ebenso gehört ein kurzer Text dazu, welcher zeigt, *was* gearbeitet wurde. Fakultativ, aber sehr hilfreich für die Auswertung und für spätere Projekte zum Vergleich: das Einteilen in Tätigkeitsgruppen (s. unten).

An jedem Review muss die Zeiterfassung einsehbar sein – das ist kein Problem und Null Aufwand, wenn die im Projektmanagement-Tool erfassten Zeiten aktuell sind.

Am Schluss müssen die Zeiterfassungen zusammen mit den Zeitauswertungen abgegeben werden. Am besten machen Sie einen kompletten Export von Redmine als CSV zu jedem Review und am Schluss und bewahren Sie das als Log-Dateien auf.

P.S.: Die Zeiterfassung kostet nur ca. 1% der Gesamtarbeitszeit, dafür kriegt man so einen Erfahrungsschatz der unbezahlbar ist. Leider ist es so, dass man während des ganzen Projektes den kleinen, vorerst mal lästigen Aufwand für die Zeitaufschreibung treiben muss und man den Nutzen erst ganz zum Schluss oder gar erst beim nächsten Projekt entdeckt ("Wie war das noch im letzten Projekt: wieviele Stunden haben wir für das Programmieren der vier wichtigen Android-Screens gebraucht (ohne die CRUD)? Wieviele Stunden für das Usability Testing aufgewendet? und wieviele Probanden hatten wir da bei den Usability Tests?")

3.12 Zeitauswertung

Am Schluss des Projektes ist es interessant, Bilanz zu ziehen: wo haben die Aufwandschätzungen gestimmt, wo hat man sich verschätzt? Am besten macht man (midestens) die zwei folgenden Auswertungen:

Stunden-Aufwand pro Woche im Gesamt-Verlauf des Projektes, als Balken oder Kurve, pro Woche summiert für das Team:

Figure 3.4: Verlauf des Aufwandes über die Zeit

Soll/Ist-Vergleich mit Kommentaren zu den grössten Abweichungen, gezeigt am Stundenaufwand pro Tätigkeitsgruppe mittel eines Kuchen- oder Balkendiagramms:

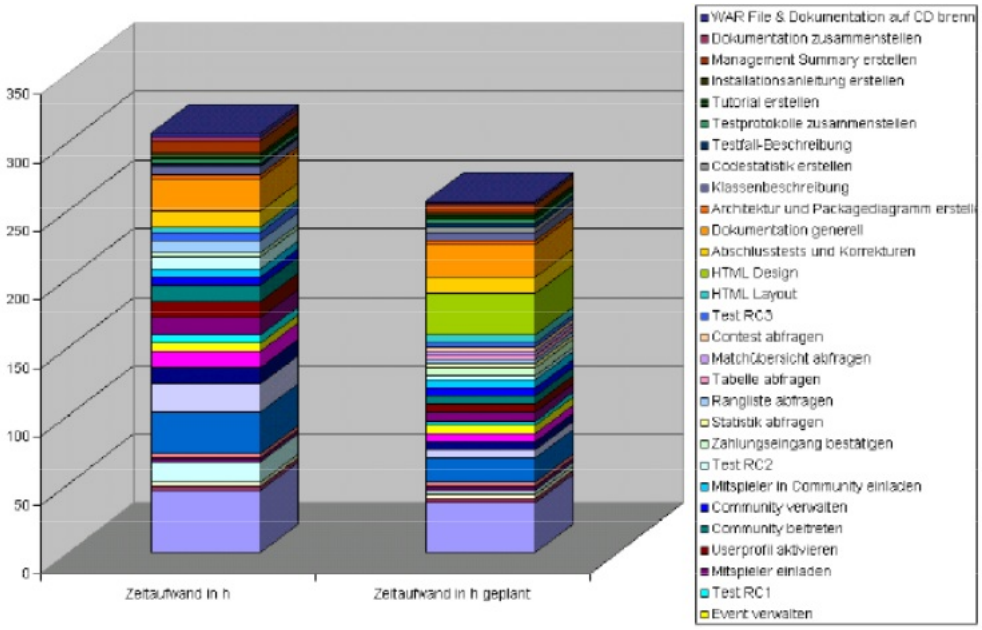


Figure 3.5: Soll/Ist-Vergleich Stundenaufwand pro Tätigkeitsgruppe

Man sieht im Bild einige auffallende Abweichungen: der geplante Aufwand für HTML Design (grün) war viel grösser als der tatsächliche. Umgekehrt wurde der Aufwand für 'Test RC 2' (hellblau) viel zu tief eingeschätzt. Wenn man solche Soll/Ist Vergleiche machen kann, lernt man, wo man sich verschätzt hat und macht es das nächste Mal besser.

Das obige Bild zeigt m.E. eine *zu* detaillierte Aufteilung, die neun untenstehend vorgeschlagenen Kategorien genügen.

Vorschlag für Tätigkeitsgruppen (kommt auch auf Ihr Projekt an):

Requirements
Analyse und Design
Programmierung
Testen und QS
Projekt-Management
Infrastruktur-Arbeiten, Deployment
Einarbeitung neue Technologien
Schlusspräsentation, Aufarbeitung Schlussdokumentation
Reserve

Achtung: keine Kategorie "Dokumentation" führen, sondern zum Beispiel die Zeit für das Zeichnen und Erklären des Domainmodells dem Entwurf ('Analyse und Design') hinzuzählen.

3.13 Schlusspräsentation

In der letzten Semesterwoche finden die Projekt-Schlusspräsentationen statt. Alle Teams (es sind meist 10-15 Teams) präsentieren ihre Arbeit vor Publikum. Für die Präsentationen wird ein eigener Zeitplan aufgestellt und es werden speziell Räume gebucht: welches Team präsentiert wann und wo. Alle, die am EPJ teilnehmen, müssen neben der eigenen Präsentation noch mindestens drei weitere Präsentationen besuchen. Das ist besonders spannend, wenn ein anderes Team eine ähnliche Projektidee gewählt hat, oder wenn ein anderes Team mit der gleichen Technologie gearbeitet hat, dann kann man gut vergleichen.

Vorgaben:

Alle Teammitglieder übernehmen einen Teil der Präsentation
Dauer der Präsentation (inkl. Demo und kurzer Fragenbeantwortung) 10 Minuten für die Gruppe + 5 Minuten pro Gruppenmitglied.

Inhalt:

- Ziele der Arbeit, und was wurde erreicht, was nicht?
- Demonstration des Programmes (ca. 5 Minuten)
- Übersichten: Analyse Modell, Architektur, Design Modell
- Auflistung der eingesetzten Technologien
- Umfang: Anzahl eigene/verwendete Klassen, eigene/fremde/generierte Codezeilen
- Angaben über den geleisteten Stunden-Aufwand und ein Soll-Ist Vergleich (s. oben)
- Projektverlauf: Was ist gut gelaufen? Wo steckten die Probleme? Was hat das Team gelernt?

Der Aufbau muss nicht zwingend der angegebenen Reihenfolge entsprechen. Empfehlung: nach einer kurzen Einführung soll schnell die Demo folgen, damit allen klar wird, um was es geht. Erst danach sollen die Konzepte und Hintergründe präsentiert werden, denn erst nach einem Gesamtüberblick (Demo) kann ein uneingeweihter Zuschauer die Hintergrundinformationen einordnen.

Die obige Aufzählung der wichtigsten Inhalts-Punkte ist auch nicht vollständig, es können auch andere Punkte, die vom Team als wesentlich eingeschätzt werden, eingebaut werden.

3.14 Schlussabgabe

Ganz zum Schluss des Semesters – letzte Semesterwoche Freitag 17h – muss das gesamte Material des Projektes abgegeben werden. Die Form der Abgabe sollte mit dem Betreuer abgesprochen sein, üblicherweise eine DVD, ein Memorystick oder ein Link auf eine Dropbox-Zip-Datei. Es muss nichts ausgedruckt sein, die Dokumente können als Word, OpenOffice, PDF oder HTML Dateien vorhanden sein, am besten alles Wichtige als PDF.

Es sollte auch die ganze Entwicklungsumgebung in der Abgabe enthalten sein, sodass der Projekt-Code gegebenenfalls übersetzt und zum Laufen gebracht werden kann. Aber bitte die IDEs (Eclipse, IntelliJ...) in der Abgabe *nicht* einschliessen, die sind zu gross, die können vom Internet heruntergeladen werden.

Abzugeben sind sämtliche Arbeitsresultate:

- Projektplan (ursprünglicher plus Nachführungen)
- Anforderungsspezifikation
- Domainanalyse
- Entwurfsbeschreibung (Software Architecture Document)
- Testspezifikation(en) und Testprotokoll(e)
- Installationsanleitung
- Bildschirmabzüge von allen relevanten Teilen der Software (mind. 4!)
- Sourcecode
- Dokumentation der REST/GraphQL-Schnittstelle, falls vorhanden
- Export von wichtigen Dateien in der Cloud: Wiki, Config Files CI-Server, ...
- Folien der Schlusspräsentation
- Erklärung zur eigenständigen Durchführung der Arbeit
- Persönliche Berichte (was gelernt, Erfahrungen, ... 1/2 - 1 Seite pro Teammitglied)
- Weitere Dokumente nach Bedarf

Projektmonitoring:

- Zeiterfassung und Zeitauswertung
- CSV-Export der Arbeitspakete (inkl. Stundenschätzung und -aufschreibung)
- Codestatistik (Anzahl Zeilen Code netto, Anzahl Klassen)
- Soll/Ist Vergleiche ("geschätzt/tatsächlich gemacht")

4. Bewertung

In den Engineering-Projekten können die Teams recht eng geführt werden: es gibt während der Projektlaufzeit fünf Reviews zu 45 Minuten, so ca. alle zwei Wochen einen, plus am Schluss eine Projektpräsentation durch das gesamte Team (für einen beispielhaften Plan siehe Fig. 3.3). In den Reviews wird immer auf einen Teilaspekt fokussiert:

1. Projektplan
2. Anforderungen und Analyse
3. Ende Elaboration (Projektdemo mit Architektur-Prototyp)
4. Architektur/Design (Software Architecture Document)
5. Q-Massnahmen (Code-Qualität, Metriken, Testing, Code-Reviews)
6. Schlusspräsentation

Falls das Team mit einem Aspekt grössere Schwierigkeiten hat, dann kann man diese diskutieren und von einer Bewertung absehen. Dann wird der Review noch einmal angesetzt – üblicherweise eine Woche später – und bis dahin kann das Team die offenen Punkte erledigen.

Wenn in einem Team erfahrene Leute mitarbeiten und das Team bereits sehr selbständig arbeitet, dann gleichen die Reviews mehr dem zügigen Abhaken einer Checkliste. Man kann dann die verbleibende Zeit mit allgemeineren Fragen oder mit der detaillierten Betrachtung einer Lösung und Diskussion von Alternativen verbringen.

4.1 Benotung

Unmittelbar nach dem Review oder bis spätestens 48h später per mail erhält das Team eine Note für diesen Review. In dieser Note sind (mit einem Gewicht von 1/4) auch der Projektverlauf und der Projektstand seit dem letzten Meilenstein berücksichtigt.

Das ergibt während dem Semester fünf Teilnoten plus am Ende eine Note für die Schlusspräsentation, d.h. 6 Reviews/Präsentationen –> 6 Einzelnoten für betrachtete Resultate gemäss Meilensteinplanung.

Nach der Schlussabgabe vergibt der Betreuer noch eine Gesamteindruck-Bewertung, das ergibt eine Note mit 3-fachem Gewicht. Dabei werden folgende Punkte berücksichtigt:
– Dokumentation (Darstellung, Nachführung, Stil, Inhalt)
– erstellte Software (Qualität, Komplexität, Umfang)
– Besondere Punkte des Projektes

Die Schlussnote wird berechnet aus den sechs Einzelnoten mit einfacher Gewichtung und der Note Gesamteindruck mit 3-fachem Gewicht. Diese berechnete Note kann durch den Betreuer noch kräftig gerundet werden.

4.2 Team-Noten

In der Regel wird allen Teammitgliedern die gleiche Note gegeben. Bei stark abweichenden Leistungen sind aber auch Individualnoten möglich.

Drei Beispiele aus den letzten Jahren:

- Einer macht kaum mit. Das Team beschloss, den Trittbrettfahrer mitzutragen. Sachverhalt wird dem Betreuer nicht transparent: gleiche Note für alle.
- Ein anderes Team verkracht sich heftig "...ich steige aus, mir ist es egal, wenn ich keine Punkte kriege, macht‘s doch alleine". Hitzige Diskussionen, das Team rauft sich zusammen, gutes Resultat, gleiche (gute) Note für alle.
- Einer im Team arbeitet nicht recht mit (zuwenig Einsatz, dürftige Resultate). Nach sorgfältiger Erwägung kriegt er eine ungenügende Note, und das restliche Team eine genügende.

Anmerkung: der letzte Fall, wo einer Person eine ungenügende Note gegeben wurde, passierte genau zweimal in den vergangenen sechs Jahren, bei einem Total von ca. 400 teilnehmenden Studenten.

Fall bei Ihrem Projekt Probleme auftauchen, melden Sie sich bitte frühzeitig und diskutieren Sie offen mit Ihrem Betreuer. In den allermeisten Fällen lässt sich eine gute Lösung für alle finden.

4.3 Checklisten für Reviews

Damit der Bewertungsprozess einfacher und ein wenig standardisierter wird, gibt es zu jedem Review eine Checkliste, insgesamt also sechs Checklisten, die in der Folge beschrieben und kommentiert werden. Die Checklisten sind auch für die Teams zugänglich, dann können sie sich besser auf die Reviews vorbereiten.

4.3.1 Checkliste 'Review Projektplan'

Software Engineering Projekt: _____ **Datum:** _____

Checkliste: Review der Projektplanung

Projektplanung						
Textueller Projektplan	++	+	o	-	--
Formale Punkte (Zweck, Gültigkeit, Referenzen, Änderungsgeschichte)					
Stil, Sprache, Darstellung					
Keine Allgemeinplätze					
Projektorganisation sinnvoll definiert					
Zusammenarbeit im Team geregelt (Besprechungen, Protokolle, …)					
Arbeitspakete, Tätigkeiten definiert und Zeitabschätzung/Zeitbudget zugeordnet					
Arbeitsresultate definiert					
Phasen, Iterationen, Meilensteine und Fertigstellungsgrad von Arbeitsresultaten def.					
Projektspezifische Risiken identifiziert und Massnahmen, Reserven eingeplant					
Qualitätssichernde Massnahmen zweckmässig geplant (Reviews, Tests)					
Richtlinien für Code und Dokumentation					
Konfigurationsverwaltung und Änderungsverfahren für Code und Doku					
Build Server, Continuous Integration					
Infrastruktur/Werkzeuge und entsprechende Verfahren zweckmässig geplant					
Vorgehen für Überarbeitung der Planung definiert					
Zeitplan					
Zeigt Phasen, Iterationen, Meilensteine					
Zeigt Tätigkeiten auf Zeitachse (mindestens für nächste Iteration)					
Übersichtliche Darstellung					
Zeiterfassung und Zeitauswertung aufgesetzt					
Stand der Arbeiten **Note:**					
	++	+	o	-	--
Zeiterfassung nachgeführt und ausgewertet auf einzelne Teammitglieder					
Stand der Arbeiten gemäss Projektplan					
Stand der Arbeiten gemäss Semesterwoche					
Beiträge der einzelnen Teammitglieder ausgewogen					
Infrastruktur gemäss Projektplanung aufgesetzt					
Alle Dokumente an dieser Review aktuell und einsehrbar					
Gesamtbeurteilung am Meilenstein **Note:** **Fazit:**					

Legende: ++ sehr gut / + gut / o genügend / - schlecht / -- sehr schlecht oder fehlt

Die aufgeführten Meilensteine sollten grob zeigen, zu welchem Zeitpunkt welche Features in etwa zu erwarten sind. Das soll keine Wasserfall-Planung sein, sondern nur zeigen, dass das Team sich über den Umfang der Features und die Reihenfolge der Implementierung Gedanken gemacht hat. Ähnliches gilt für die Arbeitspakete in Redmine: wenn sie zu allgemein gehalten sind ("Umgebung aufsetzen", "UI entwerfen", "Architektur-Proto machen"), dann sagt das nichts über das Projekt aus. Einige Arbeitspakete müssen nicht-generisch sein, so z.B. "Entwurf Level-Editor", oder "Einbindung von OAuth", sodass man eine Ahnung hat, was für Arbeit auf das Team zukommt.

Hier wird zum ersten Mal (typischerweise in Woche 3) eine Zeitauswertung verlangt. Das ist der Punkt, wo kontrolliert wird, a) ob alle Teammitglieder Zeitaufschreibung machen, b) ob die Zeit auf Arbeitspakete gebucht wird (und nicht in einer Excel-Tabelle), c) die Stundeneinträge auch vernünftige kurze Texte zur Art der Arbeit enthalten, sodass man gut sieht, *was* gemacht wurde, und d) ob alle Teammitglieder ungefähr gleich viele Stunden investiert haben.

4.3.2 Checkliste 'Review Anforderungen und Analyse'

Software Engineering Projekt: _____ Datum: _____

Checkliste: Review Anforderungsspezifikation und Domainanalyse

Anforderungsspezifikation und Domainanalyse	++	+	o	-	--
Anforderungsspezifikation					
Use Cases					
• Sinnvoll strukturiert (EBP Level, CRUD zusammengefasst) und vollständig					
• Übersichtliches, korrektes UC-Diagramm und Beschreibungen in Brief-Format zur Übersicht					
• Wichtigste UCs: verständlich, vollständig, essential style, hoher Informationsgehalt (aber nicht ganz 'fully dressed' v.a. nicht bei Abläufen)					
Nichtfunktionale Anforderungen					
• Qualitätsmerkmale (Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit,Übertragbarkeit)					
• Mengen, Leistung					
• Überprüfbar formuliert (ja/nein, ‚abhakbar')					
Darstellung, Sprache, Stil					
Beschreibung der Schnittstellen zu Umsystemen					
Kontext-Diagramm (optional)					
User Interface Skizzen (optional)					
Personas (optional)					
Domainanalyse					
Domainmodell Diagramm					
• Gute Abbildung der Problemdomain					
• Formal korrekt					
• vollständig (Elemente mit tiefer Prio markiert/gefärbt)					
• Beschreibung der Klassen wo sinnvoll					
• Beschreibung der Attribute wo sinnvoll					
• Beschreibung von Beziehungen wo sinnvoll					
	++	+	o	-	--
Stand der Arbeiten					
Zeiterfassung nachgeführt und ausgewertet auf einzelne Teammitglieder					
Stand der Arbeiten gemäss Projektplan					
Stand der Arbeiten gemäss Semesterwoche					
Beiträge der einzelnen Teammitglieder ausgewogen					
Arbeitsresultate gemäss Feedback letzter Review überarbeitet					
Alle Dokumente an dieser Review aktuell und einsehbar					

Legende: ++ sehr gut / + gut / o genügend / - schlecht / -- sehr schlecht oder fehlt

Dieser Review ist zur Hauptsache den Use Cases, dem Domainmodell und den nicht-funktionalen Anforderungen gewidmet. Das Team hat ja keinen Kunden, den man verstehen und mit dem man sich einigen muss. Es geht einzig darum, zu sehen, was sich das Team vorgenommen hat, und ob die Anforderungen gut formuliert sind.

Bei diesem Review soll klar werden, was das Team machen wird: der Funktionsumfang (Scope) des Projektes soll gut definiert sein, so, wie wenn ein Kunde vom Entwicklungsteam präsentiert bekommt "Das werden wir in der Projektlaufzeit schaffen" und der Kunde sagt "Ich glaube, das Team hat mich gut verstanden und wird das liefern, was ich brauche".

4.3.3 Checkliste 'Review End of Elaboration'

Software Engineering Projekt: _____ **Datum:** _____

Checkliste: End of Elaboration

Demonstration Architekturprototyp / Review End of Elaboration	++	+	o	-	--
Architekturprototyp					
lauffähig					
über alle Layers, über alle Tiers					
technische Risiken abgedeckt					
Architektur in groben Zügen dokumentiert (Schichten und wichtigste Klassen)					
Evtl. Deploymentdiagramm(e)					
Evtl. Designklassendiagramm(e) im Entwurfsstadium vorhanden					
Evtl. Sequenzdiagramm(e) für komplexe Zusammenhänge					
Test des Architekturprototyps					
systematische Unittest, wo sinnvoll					
Systemtest durchgeführt					
Stabilität des Prototyps an Demo					
Demonstration des Architekturprototyps					
gut vorbereitet					
gut demonstriert					
interessante Aspekte gut erläutert					
Weitere Ziele der Elaboration-Phase erreicht (neben Architekturprototyp)					
Anforderungen jetzt vollständig und stabil (UCs, NFR)					
(UI Design, Paper Prototypes, Wireframes) (Datenmodell stabil)					
Test- und Reviewprozeduren definiert					
Risikoanalyse nachgeführt, hauptsächliche Risiken beseitigt					
Planung für Construction-Phase verfeinert					
Entwicklungsumgebung komplett eingerichtet (IDEs, git, CI mit Metriken und Unit Tests, Dokumentenablage, Wiki)					

Stand der Arbeiten **Note:** **Fazit:**	++	+	o	-	--
Zeiterfassung nachgeführt und ausgewertet auf einzelne Teammitglieder					
Stand der Arbeiten gemäss Projektplan /					
Stand der Arbeiten gemäss Semesterwoche					
Beiträge der einzelnen Teammitglieder ausgewogen					
Arbeitsresultate gemäss Feedback letzter Review überarbeitet					
Alle Dokumente an dieser Review aktuell und einsehbar					

Gesamtbeurteilung am Meilenstein **Note:** **Fazit:**

Legende: ++ sehr gut / + gut / o genügend / - schlecht / -- sehr schlecht oder fehlt

Das ist der wichtigste Review im ganzen Projekt. Hier wird geschaut, ob alle Punkte abgehakt werden können, ob das Ende der Elaboration-Phase tatsächlich erreicht ist. Falls alle Punkte erfüllt sind, kann das Team getrost in die Construction-Phase gehen und mit dem Programmieren vorwärts machen.

Wenn die Punkte nicht erreicht sind, dann a) soll das Team schauen, dass die nicht erfüllten Punkte so schnell wie möglich nachgeholt werden, und b) ist zu erwägen, dass bereits jetzt der Funktionsumfang reduziert bzw. neu priorisiert werden sollte, damit das Team trotzdem bis zum Projektende etwas Schönes zum Laufen bringt. Das Nicht-Erreichen des Punktes 'End of Elaboration' ist ein frühes Signal dafür, dass die ursprünglichen Projektziele in der zur Verfügung stehenden Zeit vermutlich nicht erreicht werden.

4.3.4 Checkliste 'Review Architektur'

Checkliste: Review SW-Architektur

Review Architektur & Package-/Klassendesign	++	+	o	-	--
Architektur/Design inhaltlich					
Physische Architektur (Verteilung auf Rechner, Prozesse): Schwierigkeitsgrad? Gut gelöst?					
Logische Architektur (Gliederung in Packages, Schichtenarchitektur): Schwierigkeitsgrad? Gut gelöst (hohe Kohäsion)?					
Schnittstellen zwischen Packages: Gut gelöst (geringe Kopplung)?					
Persistenz: Gut gelöst?					
Design der Packages: Gute Klassenstruktur (evtl. Designpattern)?					
Externes Design, User Interface, UX: Gut gelöst?					
Besonderheiten der Architektur: gut gelöst?					
Erweiterungs-Szenario "Mehr Funktionalität, grössere Domäne"					
Performance-Szenario "Grosser Erfolg: mehr User, mehr Last"					
Eingesetzte Technologien: Schwierigkeitsgrad, gut gemeistert?					
Architektur/Designdokumentation					
Korrekt, gut verständlich?					
Das Wesentliche beschrieben?					
Designentscheidungen begründet?					
Form, Sprache, Stil					
Entspricht die Doku der tatsächlichen Code-Struktur?					
Stand der Arbeiten **Note:** **Fazit:**					

	++	+	o	-	--
Zeiterfassung nachgeführt und ausgewertet auf einzelne Teammitglieder					
Stand der Arbeiten gemäss Projektplan /					
Stand der Arbeiten gemäss Semesterwoche					
Beiträge der einzelnen Teammitglieder ausgewogen					
Arbeitsresultate gemäss Feedback letzter Review überarbeitet					
Alle Dokumente an dieser Review aktuell und einsehbar					
Gesamtbeurteilung am Meilenstein **Note:** **Fazit:**					

Hier werden zwei Dinge angeschaut:

a) Ist die Architektur gut gemacht? Ist die Architektur der Problemstellung angemessen?

b) Ist die Architektur gut dokumentiert, sodass Aussenstehende (Entwickler) sie gut verstehen? mit dem Nebenschauplatz: Entspricht die dokumentierte Architektur der tatsächlichen?

Als Architekturbeschreibung sollten mindestens die statische Sicht (Schichten-Architektur) und die Verteilungssicht (Deployment) dokumentiert sein. Von Vorteil ist auch eine gute Beschreibung der Schnittstelle(n), z.B. eine REST Schnittstelle, die mit Swagger (swagger.io) entworfen wurde, ist gleichzeitig auch schön dokumentiert.

Neben den oben erwähnten Punkten finden sich in der Architektur-Dokumentation zur Hauptsache noch Begründungen zu Entwurfs-Entscheidungen ("Warum wurde das so gelöst"), und verworfene Alternativen, damit die Nachfolger die Gründe kennen, warum die eine Lösung bevorzugt wurde. Kurz, es werden 'warum'-Fragen beantwortet.

Je nach Projekt und Architektur wird der Fokus auch noch auf anderen Punkten liegen (z.B. Persistenz, Performance, Security u.a.m.), weil die nicht-funktionalen Anforderungen die Architektur stark beeinflussen und die Schwerpunkte anders setzen.

4.3.5 Checkliste 'Review Q-Massnahmen'

Software Engineering Projekt: _____ **Datum:** _____

Checkliste: Qualitätsicherungs-Massnahmen und Code-Qualität

Hinweis: **Fett gedruckte Punkte müssen erfüllt sein** Bewertung der Q-Massnahmen:						
	++	+	o	-	--
Planung Q-Massnahmen im Projektplan/Zeitplan					
Überprüfung der Dokumentation (Reviews)					
Reviews mit Betreuer: geplant, durchgeführt und protokolliert					
interne Reviews: geplant, durchgeführt und protokolliert					
Überprüfung des Codes durch Codereview					
zumindest informal (geplant, in Protokollen erwähnt)					
geplant, durchgeführt und dokumentiert					
Codierrichtlinien vorhanden (bzw. existierende referenziert)					
Überprüfung des Codes durch Test:					
Unittests (JUnit etc.)					
Tests der Haupt-(REST/GraphQL)-Schnittstelle					
Systemtests von Hand (mit Protokoll) oder autom. End-to-End Tests					
Einsatz an Werkzeugen					
CI-Server mit Tools und Dashboard					
Metriken (LOC, Anzahl Klassen…)					
autom. Überprüfung von Codierrichtlinien, potentiellen Bugs, Strukturanalyse					
Testabdeckung					
Weitere Q-Massnahmen					
Performance-Tests					
Usability-Tests					
Informeller Codereview					
Qualität von Code					
alle Teammitglieder haben programmiert?					
Übereinstimmung mit Design					
Codierrichtlinien eingehalten					
Codestatistik: Anzahl Klassen, Anzahl Zeilen Code (Delivered Lines of Code)					
Stand der Arbeiten					
	++	+	o	-	--
Zeiterfassung nachgeführt und ausgewertet auf einzelne Teammitglieder					
Stand der Arbeiten gemäss Projektplan /					
Stand der Arbeiten gemäss Semesterwoche					
Beiträge der einzelnen Teammitglieder ausgewogen					
Arbeitsresultate gemäss Feedback letzter Review überarbeitet					
Alle Dokumente an dieser Review aktuell und einsehbar					
Gesamtbeurteilung am Meilenstein **Note:** **Fazit:**					

Legende: ++ sehr gut / + gut / o genügend / - schlecht / -- sehr schlecht oder fehlt

Der Q-Review fokussiert auf die qualitätsverbessernden Massnahmen: Unit Tests, Usability Testing, Code Reviews oder Pair Programming (müsste man in git, auf dem CI-Server und in den Arbeitspaketen sehen) plus Einsatz von Werkzeugen, die den Code durchleuchten (metrics, ReSharper, findbugs, lint, checkstyle, etc.).

Das, was sich das Team vorgenommen hatte, wurde in den Projektplan geschrieben. Das, was das Team tatsächlich gemacht hat, sollte man im Code der Unit Tests, in den Review-Protokollen (für grosse Code-Reviews) und Test-Protokollen (für System- und Usability-Tests) ersehen können. Danebst wird der Betreuer den Code durchlesen und auf Auffälligkeiten hinweisen (Code Smells, schlechter Stil, nicht eingehaltene Richtlinien, ...).

4.3.6 Schlusspräsentation

EPJ Meilensteinreview „Schlusspräsentation“

Projekt : _____ **Datum:** _____

Inhalt:

- Ziele der Arbeit, Was wurde erreicht?
- Demonstration der SW (das Wesentliche gezeigt? Verständlich?)
- Übersichten:
  - Kontext
  - Analyse Modell: UC Diagramm, Domainmodell
  - UI Design, UX
  - Architektur, (statisch, Schichtendiagramm)
  - Deployment
  - Datenmodell
  - Dynamik (Szenario, Sequenzdiagramm)
  - Tools, Frameworks
  - Schnittstellen
  - Testing (Unit, Usability, Performance)
- Besonderes:
  -
- Umfang:
  - Anzahl eigene/verwendete Klassen
  - LOC (eigene/fremde/erzeugte Codezeilen; auch XML etc.)
- Angaben über geleisteten Stunden-Aufwand mit Soll-/Ist-Vergleich
  - Total-Aufwand
  - auf Personen aufgeschlüsselt
  - nach Tätigkeiten aufgeschlüsselt
- Projektverlauf:
  - Was ist gut gelaufen?
  - Probleme?
  - Was gelernt?

Aufbau

- Gliederung? Timing (das Wichtigste bekam am meisten Zeit?)
- das Wesentliche gezeigt?
- Aufteilung auf Referenten
- Verständlich? Das Publikum abgeholt, interessiert?
- Zeit eingehalten? **Note:** _____

Bei der Schlusspräsentation ist es sehr wichtig, dass das Team die Zeitvorgaben einhält (siehe Abschnitt 3.13). Nur so kann der straffe Präsentations-Fahrplan in der letzten Woche eingehalten werden.

Die anderen Punkte der Checkliste sind einfach zu verstehen, es müssen auch nicht alle Punkte erfüllt sein. Hauptsache ist, dass das Publikum gespannt zuhört und etwas Interessantes zu sehen bekommt. Die Gesamtstimmung ist sehr schwierig zu bestimmen oder zu erklären, am besten mit den Gedanken, die im Publikum am Schluss auftauchen sollen, z.B. "Wow, in diesem Team hätte ich auch gerne mitgearbeitet", oder "Das ist eine super Idee, diese Software könnte ich auch gut gebrauchen", oder "Gute Idee, toll umgesetzt".

5. Einordnung

5.1 Typische Werte für ein Engineering-Projekt

Ein typisches, d.h. durchschnittliches Engineering-Projekt weist die folgenden Kennwerte auf:

- 10 Use Cases (inkl. CRUD)
- 120 Arbeitspakete (User Stories) insgesamt im Backlog während der gesamten Projektdauer
- 7 Iterationen/Sprints
- 8 Domain-Klassen im Domainmodell
- 2 wichtige, komplexe Screens, plus mehrere weniger wichtige Screens (z.B. CRUD oder einfache Listen)
- 4000 Lines of Code (Netto Codezeilen, ohne Leerzeilen, ohne Kommentarzeilen)
- 4 Personen im Team
- 500 Arbeitsstunden (d.h. ca. 5% mehr als die vorgegebenen 120 Std./Person)

Eine Bitte an alle Teams: versuchen Sie nicht, diese Durchschnittswerte als Idealwerte zu interpretieren, das Ideal liegt bei allen Projekten woanders.

Die Zahlen sind stark beeinflusst von der Art des Projektes (z.B. gibt es bei Games meist nur einen oder zwei Use Cases), von der Komplexität des Themas (ein schwieriges Thema mit viel Einarbeitungsaufwand wird weniger Code zur Folge haben), von der Programmier-Erfahrung des Teams (erfahrene Programmierer/innen schaffen das Vielfache im Vergleich zu Programmier-Noobies), und von der Team-Kommunikation (grössere Teams mit Kommunikationsschwierigkeiten, z.B. zuwenig gemeinsame Zeit, brauchen mehr Diskussionen, bis sie sich einig sind – eine unproduktive Zeit, die sich in kleineren Resultaten niederschlägt).

5.2 Realitätsnähe

Das Engineering-Projekt an der HSR ist in ein paar Punkten nicht sehr realitätsnah:

- Die durchschnittliche Grösse der Arbeitspakete ist nach üblichen Masstäben definitiv zu klein. Normalerweise sind Arbeitspakete/User Stories um die 2-4 Tage gross (bei Sprint-Längen von 2-3 Wochen mit Vollzeit-Mitarbeitenden), im Engineering-Projekt sind es im Schitt 2-6 Stunden pro Arbeitspaket, weil die Studierenden pro Sprint nur 17 Stunden investieren (pro Woche 8-9 Stunden).

- Die Anzahl der Construction-Iterationen ist zu klein, wir kürzen hier ab, um innerhalb eines Semesters fertig zu werden (siehe Bild 5.1 unten). Deswegen ist das Projekt in den allermeisten Fällen unfertig – aber das Team weiss wenigstens, was noch fehlt und wie lange man noch programmieren müsste.
- Die Teams haben keine externen Kunden, also haben sie kaum Schwierigkeiten mit unklaren Spezifikationen, mit Scope Creep oder mit der Reduktion des Funktionsumfangs im Verspätungsfall: das Team muss nur (nur?) unter sich einig sein.
- Das Projekt ist mit rund 2-6 Personenmonaten nach industriellen Masstäben winzig. So kleine Projekte machen in der Regel überhaupt keine Probleme. Vor allem die Komplexität, die mit der Grösse üblicherweise nicht-linear wächst, ist bei einem so kleinen Projekt überschaubar. Ebenso ist die Kommunikation zwischen maximal fünf Personen, die sich auch mindestens einmal in der Woche sehen, noch sehr direkt.
- Wenn man das Maximum aus dem agilen Prozess holen will, dann sollte man sehr schnell ein 'Minimum Viable Product' (MVP) haben, das vom Kunden ausprobiert und getestet werden kann, damit echter Feedback aus der Praxis kommt und das Produkt weiterentwickelt werden kann. Die Engineering-Projekte erreichen ein MVP – wenn überhaupt – erst nach der Mitte des Projektes (auch eine direkte Folge der weggelassenen Construction-Iterationen).
- Der Architektur-Review kommt zu spät: normalerweise macht man den ersten Architektur-Review vor Ende Elaboration. Im Engineering-Projekt würden sich die Reviews vor Ende Elaboration aber zu sehr häufen, darum später.

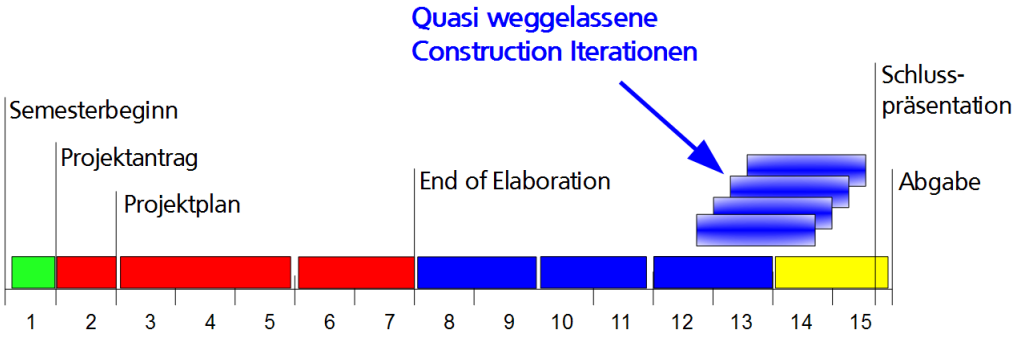


Figure 5.1: Die in einem EPJ weggelassenen Iterationen

Ungeachtet all dieser Argumente ist das Engineering-Projekt an der HSR für die Student/innen ein Highlight des ganzen Studiums, die Rückmeldungen der Studierenden ist durch die Bank positiv. Sie sind stolz auf das Erreichte und sie empfinden die intensive Gruppenarbeit als echten Zugewinn. Selbst Jahre später werde ich (D.K.) von Ehemaligen auf das Engineering-Projekt angesprochen und wie toll das war. Auch Feedback von Arbeitgebern enthält immer wieder Lob auf die praktischen Fähigkeiten der HSR-Informatik-Absolvent/innen: "die sind erstaunlich schnell einsatzfähig und wissen bereits eine Menge über Software-Projekte".

Dass die Projekte relativ klein sein müssen, ist auch eine Folge der Modularisierung des Studiengangs: Alles Wissen muss innerhalb eines Semesters in einem abgeschlossenen Modul Platz haben (Bologna-Reform mit Bachelor/Master). Trotz dieser Beschränkung ist das Engineering-Projekt gross genug um eine Menge an Lerneffekten nebst der Gruppenarbeit auszulösen. Während des Projektes setzen die Studierenden neue Techniken ein, von denen sie im Unterricht nur oberflächlich gehört haben. Hier können sie Sprachen, Werkzeuge und Techniken einmal in einem realen Projekt einsetzen und sehen, wie sie sich im echten Einsatz bewähren. Es fällt auch auf, dass die Studierenden sich nicht nur konkrete Techniken aneignen, sondern dass sie oft auch solche Dinge lernen, die mit sonstigen Übungen nur sehr schwer zu vermitteln sind, z.B. Fragen der Bewertung oder der Gewichtung, Fragen der Betrachtungsperspektive, Softwarearchitektur-Fragen, kurzum, Fragen, die bei kleineren Übungen in dieser Form nicht auftauchen.