SOFTWARE ENGINEERING

Bericht

22.02.2022

Team 4 – Zeiterfassungssoftware
Kevin K[redacted]
Sven M[redacted]
Paul S[redacted]
Moritz U[redacted]
Leander W[redacted]

INHALT

1 EINLEITUNG

Im Zuge des Kurses Software Engineering wurde anhand eines Projekts eine Fallstudie zur Analyse, Planung und Umsetzung von Software durchgeführt. Die Ergebnisse werden im folgenden Bericht zusammengefasst dargestellt.

Dieser Bericht bezieht sich auf den Prozess des Software Engineerings zum Entwickeln einer Desktop-Anwendung zur projektbasierten Arbeitszeiterfassung durch ein Team aus fünf Studierenden.

GitHub-Repository
Der Quellcode zur Software in folgendem Repository einsehbar:
https://github.com/SirMightyMo/QT_App

Aktueller Build v1.0.0:
https://github.com/SirMightyMo/QT_App/releases/tag/v1.0.0

Bei Ausführung der Software ist ein Login mit einem Standarddatensatz möglich:

Benutzername:	Bob
Passwort:	abc

Für die vollständige Anzeige der Anwendung ist aktuelle eine Bildschirmauflösung von 1920 × 1080 Pixeln notwendig.

Die Software wurde komplett in Java geschrieben. Zur Realisierung der Datenbank wurde die *H2 Database Engine* [1] verwendet. Zudem wurden zur Gestaltung der Benutzeroberfläche die Frameworks *Swing* und *Flatlaf*[2] eingesetzt zum Testen *AssertJ-Swing*[3] und *JUnit*[4] verwendet.

Als Vorgehensmodell im Projektmanagement wurde sich auf Scrumban geeinigt.

[1] Vgl. http://www.h2database.com/html/main.html
[2] Vgl. https://www.formdev.com/flatlaf/
[3] Vgl.https://joel-costigliola.github.io/assertj/assertj-swing.html
[4] Vgl.https://junit.org/junit5/

2 ANFORDERUNGEN

Wie bei allen Projekten steht am Anfang eine Idee bzw. ein Auftrag. In unserem Fall sind wir unsere eigenen Auftraggeber und haben uns zum Ziel gesetzt, wie eingangs bereits erwähnt, eine Desktop-Anwendung zur projektbasierten Arbeitszeiterfassung zu entwickeln.

Vorgaben seitens des Kurses gab es im Grunde nur zwei:
Die Entwicklung einer objektbasierten Software, um den Prozess des Software Engineerings gut nachvollziehen zu können, sowie die Termine, zu dem die Software oder die zugehörigen Projektberichte fertiggestellt sein sollten.

Um zu definieren, was wir überhaupt für Anforderungen mit dieser Anwendung erfüllen wollen, muss sich zunächst mit der Anforderungsanalyse beschäftigt werden. Denn erst die definierten Anforderungen können die Idee in ein planbares Vorhaben übersetzen, das von den Entwicklern (uns) dann unter Beachtung zur Erfüllung dieser Kriterien umgesetzt werden kann. Oder mit Dijkstras Worten von 1982:

„*It is the sole purpose of the specifications to act as the interface between the system's users and the system's builders.*“ [5]

2.1 Anforderungsanalyse

Um zu analysieren, was die Software am Ende überhaupt können soll und welche Bedürfnisse sie befriedigen soll, haben wir zunächst ein Brainstorming veranstaltet und ganz frei gesammelt, was sich jedes einzelne Teammitglied überhaupt unter einer Zeiterfassungssoftware vorstellt. In diesem Sinne können wir von *offenen Anforderungen* sprechen, die wir an uns selbst gestellt haben. Da jeder von uns aber eben seine eigene Vorstellung der Anwendung im Kopf hatte, mussten wir durch Diskussionen auch die *latenten Anforderungen* erarbeiten, die entweder gar nicht richtig bewusst waren und sich erst entwickelt haben oder die evtl. als gegeben betrachtet wurden.

Wie unter Punkt *3.1 Vorgehensmodell: Scrumban* beschrieben, wussten wir bereits vor der Anforderungsanalyse, dass wir das agile Vorgehensmodell *Scrumban* anwenden wollen. Dadurch war bereits klar, dass wir relativ flexibel auf sich im Laufe der Projektarbeit neu entwickelnde Anforderungen reagieren können würden.
Da wir unsere eigenen „Auftraggeber“ sind, verschwimmt die Grenze der unterschiedlichen Anforderungsarten ohnehin etwas.

2.1.1 Benutzeranforderungen

Auf Grundlage unseres Brainstormings, dem Strukturieren unserer niedergeschriebenen Gedanken und der anschließenden Diskussionen und UML Use-Case-Diagrammen haben

[5] Dijkstra, Edsger W.: Selected Writings on Computing: A Personal Perspective, Softcover reprint of the hardcover 1st edition 1982, 1982, S. 63

wir dann *User Stories* (bestehend aus dem Nutzer, der Funktion und dem Wert, der diese Funktion für den Nutzer hat) entwickelt, die für Scrum/Scrumban üblich sind und die Anforderungen widerspiegeln. Um diese User Stories möglichst genau aus Sicht des Endnutzers definieren zu können, haben wir uns auch gleich überlegt, für wen das Produkt überhaupt gedacht sein soll. Definiert haben wir so, dass die Software für Freelancer (also Einzelpersonen) geeignet sein soll, die projektbasiert arbeiten. Trotzdem wollten wir den Gedanken auf eine mögliche, perspektivische Erweiterung für kleine Unternehmen im Kopf behalten, was sich später auch in der Wahl der Datenbank-Engine niedergeschlagen hat.

Diese User Stories wurden dann in ein *Backlog* überführt, um zusätzlich *Akzeptanzkriterien* (wann ist für den Auftraggeber eine User Story erfüllt) sowie eine *Definition of Done* (welche Punkte müssen erledigt sein, damit eine User Story aus Auftragnehmer- und Entwicklersicht als „fertig“ gilt) zu definieren. Um diesen Prozess zu unterstützen, haben wir bereits erste Mockups/Wireframes umgesetzt, um auf Grundlage dieser visuellen Stütze weiter über die Software zu diskutieren. Mithilfe der Akzeptanzkriterien haben wir außerdem versucht, noch sehr vage gehaltene Anforderungen objektivierbar zu machen, ohne uns vorab im Entwicklungsprozess durch eine zu strenge Definition zu sehr einzuschränken. So haben wir beispielsweise „Benutzerfreundlichkeit“ übersetzt in „alle Funktionen sollen mit maximal drei Schritten/Klicks erreichbar sein“ oder die Definition einer maximalen „Schulungsdauer“, die nötig ist, um die Anwendung sicher verwenden zu können.

Wie der Begriff „User Story“ bereits verrät, handelt es sich hierbei vor allem um Benutzeranforderungen, die in diesem Prozess definiert werden. Gleichzeitig sind diese auch zum Großteil rein funktional, wobei nichtfunktionale Anforderungen (wie bspw. Datenschutz und Rechtliches oder Portierbarkeit) aus unserer Sicht ohnehin eher irrelevant sind (abgesehen von so etwas wie dem Abgabetermin für unser Projekt), da wir nicht mit einem „echten Kunden“ konfrontiert sind. Die *nicht funktionalen Anforderungen* sind für uns also vor allem durch den Rahmen unseres Hochschulprojekts definiert. Hierbei war für uns besonders relevant sich an die erlernten Vorgehensweisen zum Projektmanagement und zur Programmierung korrekt anzuwenden.

2.1.2 Systemanforderungen

Innerhalb des Anforderungsanalyseprozesses haben wir nur den groben Rahmen für die Systemanforderungen festgelegt. So war schnell für uns klar, dass wir die Anwendung in Java programmieren wollen, was sich mit dem Wunsch eine möglichst plattformunabhängige Anwendung (MacOS, Linux, Windows) zu entwickeln, gut gedeckt hat. Auch das wir eine geeignetes Datenbankverwaltungssystem benötigen, war schnell klar, festgelegt haben wir uns aber zunächst nicht.

Ohnehin kam hier wieder der Vorteil des agilen Vorgehensmodells zum Tragen, dass wir uns auch zum Zeitpunkt einer Sprintplanung noch über Anforderungen austauschen können, sollten diese erst zu diesem Zeitpunkt auftauchen.

Teilweise haben wir Systemanforderungen aber auch bewusst zunächst offengelassen, da wir uns erst noch mit neuen Thematiken beschäftigen mussten, um diese besser einordnen zu können. So war beispielsweise anfangs nicht klar, wie wir die Benutzeroberfläche realisieren würden, nur dass wir diese in irgendeiner Form umsetzen werden müssen. Teil der Systemanforderungsanalyse war also auch die Analyse von neuen Frameworks und Tools, die zur Umsetzung des Projekts beitragen könnten. So haben wir uns bspw. aufgrund der sehr ausführlichen Dokumentation von *Java Swing* für eben dieses Framework entschieden.

Auch bzgl. der Wahl eines Datenbankmanagementsystems gab es mehrere Möglichkeiten. Fest stand für uns, dass es sich um eine relationale Datenbank handeln soll, die sich möglichst problemlos mit Java steuern können lassen muss und auch der Umgang damit möglichst einfach sein soll. Vor dem Hintergrund, dass sich die Anwendung perspektivisch auch irgendwann von der reinen Nutzung einer Einzelperson hin zur Nutzung durch ein kleineres Unternehmen entwickeln können soll, haben wir uns für die *H2 Database Engine* entschieden. Sie bringt nicht nur eine grafische Benutzeroberfläche mit sich, um die Datenbank unkompliziert einsehen und überprüfen zu können. Sie bietet außerdem verschiedene Möglichkeiten zur Verwendung der Datenbank. In unserem Fall wird die Datenbank nun lokal von der Anwendung erstellt, sie könnte aber zu einem späteren Zeitpunkt auch problemlos auf einem Server laufen und wäre damit überall erreichbar.

3 PROJEKTMANAGEMENT & VORGEHENSMODELL

3.1 Vorgehensmodell: Scrumban

Bereits vor der Anforderungsanalyse wurde sich im Team darauf geeinigt, dass wir als Vorgehensmodell *Scrumban* verwenden möchten – also eine agile Vorgehensweise, die sich an den Werkzeugen von *Scrum* und *Kanban* bedient. Denn einerseits möchten wir das agile Arbeiten, wozu Scrum laut Ian Sommerville als eine der meistgenutzten Methoden gehöre[6], besser kennenlernen, den daran hängenden Prozess anhand einer Fallstudie verinnerlichen und gleichzeitig auf nützliche Tools wie das Kanban-Board zur Organisation nicht verzichten müssen. Andererseits bringt die agile Arbeitsweise einige Vorteile mit sich:

Da wir bislang wenig Erfahrung im Softwareengineering und Software Development haben, ist es aus unserer Sicht nicht sinnvoll ein streng plangesteuertes Vorgehensmodell zu verwenden, das sich möglichst starr an ein (Lasten- und) Pflichtenheft hält. Da wir uns im Laufe des Projekt- bzw. Entwicklungsprozesses neue Fähigkeiten aneignen müssen, wie bspw. das Einrichten und Verwenden einer geeigneten Datenbank oder das

[6] Sommerville, Ian: Software Engineering, 10. Aktualisierte Auflage, S. 100 Hallbergmoos, Deutschland: Pearson Deutschland, 2018.

Umsetzen eines User Interfaces, können wir nicht im Voraus die gesamte Software durchplanen, ohne damit rechnen zu müssen, dass viele Änderungen auf uns zukommen werden. Es kann also im Laufe des Prozesses passieren, dass sich die Anforderungen ändern oder ergänzt werden müssen. Mit einer agilen Vorgehensweise ist dies aber weniger ein Problem als mit einem plangesteuerten Prozess.
Kurzgesagt sind wir mit einer agilen Methode deutlich flexibler, können schneller auf auftretende Probleme oder Änderungen reagieren, sehen schneller Zwischenergebnisse, die direkt getestet werden können (bzw. müssen), und sind somit in der Gesamtheit wohl effektiver in der Projektarbeit.

Die frühe Entscheidung für *Scrumban* war aber auch wichtig, um die Form der Anforderungen zu definieren: Da diese je nach Vorgehensmodell in unterschiedlicher Art und Weise formuliert werden (bei agilen Methoden i. d. R. als *User Stories* zusammen mit *Akzeptanzkriterien* und einer *Definition of Done*), ist es sinnvoll sich bereits vorher auf eine geeignete Methode zu einigen.

An dieser Stelle sei auch noch einmal betont, dass wir uns frei an den Werkzeugen von Scrum und Kanban bedienen, um eine für uns und das Projekt geeignete Arbeitsweise zu entwickeln, und uns somit nicht streng an eine Workflow-Vorgabe halten, wie sie z. B. im Internet zu finden ist, die aber je nach Quelle auch variiert.
Unser Vorgehen orientiert sich daher hauptsächlich an Scrum unter Zuhilfenahme eines Kanban-Boards zur besseren Organisation und Einsicht des aktuellen Stands.

3.2 Arbeitsweise im Team

Durch die gewählte Methode Scrumban ergibt sich bereits eine bestimmte (von dem Modell vorgegebene) Arbeitsweise im Team, von der wir aber aufgrund dessen, dass wir es nicht mit einem Realprojekt zu tun haben, an dem wir in Vollzeit arbeiten können, etwas abgewichen sind und abweichen werden.

Die Anforderungen wurden gemeinsam mit allen Teammitgliedern im Rahmen eines Brainstormings definiert, um diese im Anschluss durch stetige Diskussionen anzupassen und zu ergänzen. Das gesamte Team füllt also nicht nur die Rolle als Planungs- und Entwicklungsteam aus, sondern auch die des Kunden.

Um die in Scrumban vorgesehenen Sprints realisieren zu können, werden wir versuchen die Sprintdauer auf eine Woche begrenzen, entsprechend müssen die User Stories bzw. deren zugehörige Tasks bei der Sprintplanung so eingeteilt und definiert werden, dass dies möglich wird. Andernfalls (mit längerer Sprintdauer) bleiben bis zum Abgabetermin voraussichtlich zu wenige Iterationszyklen, um das Projekt sinnvoll steuern zu können.
Auch werden wir aufgrund der neben dem allgemeinen Studium bleibenden knappen

Zeit kein tägliches Scrum (Meeting) abhalten, wie es in *The Scrum Guide* definiert wird[7], sondern ein wöchentliches. Damit werden dann auch das *Sprint Review* (produktbezogen), *Sprint Retrospective* (teambezogen) sowie das *Sprint Planning* einhergehen müssen. Kurze, zwischenzeitliche Absprachen sind dennoch über die üblichen Kanäle wie bspw. *MS Teams* möglich, um unkompliziert, aber zeitunabhängig Themen zu besprechen.

Durch das eingesetzte Kanban-Board, in dem alle für den geplanten Sprint definierten Tasks aufgeführt sein werden, ergibt sich eine gute Übersicht über den aktuellen Projektstand und welches der Teammitglieder gerade an welcher Aufgabe arbeitet. Gleichzeitig wird auch das *Sprint* sowie *Product Backlog* stetig (als eigenständiges Dokument) aktuell gehalten.
Während in *Scrum* die Tasks mit dem *Sprint Planning* auf die Mitglieder des Entwicklerteams aufgeteilt werden und Kanban eine *Pull-Herangehensweise*[8] (Entwickler nehmen sich noch unbearbeitete Tasks selbst aus der Liste, sobald der vorherige Task abgeschlossen wurde) verfolgt, werden wir voraussichtlich eine Mischung davon realisieren: Im Team wird grob darüber gesprochen, wer gerne welchen Bereich bearbeiten würde (bspw. *Datenbanken* oder *UI*), die genauen Tasks aber nach dem Pull-Verfahren ausgewählt. So sollen bereits vorhandene Kompetenzen gut ausgespielt werden können, ohne die Aufgaben zu starr zu verteilen und die verfügbare Zeit eines Sprints möglichst effizient nutzen zu können.
Die User Stories werden entsprechend ihrer Priorität absteigend (von hoher bis zu niedriger Priorität) eingeplant und abgearbeitet. Bei der Planung berücksichtigt werden müssen auch die Abhängigkeiten der User Stories untereinander, wobei die Priorisierung dies im Groben schon abbildet. Nach einem Sprint können Prioritäten u. U. angepasst werden.

Zur Organisation mit einem Kanban-Board verwenden wir die Software *Trello*[9]. In das Trello-Board werden die User Stories als einzelne Kanban-Karten überführt, innerhalb dieser Karten befinden sich dann Checklisten mit den zu den User Stories gehörigen Tasks sowie den Akzeptanzkriterien. Da eine User Story durchaus von mehreren Mitgliedern bearbeitet werden kann, werden die daran Arbeitenden in der Karte markiert. Unter diesen markierten Mitgliedern findet dann eine individuelle Kommunikation bzgl. der zu bearbeitenden Tasks statt. So soll vermieden werden, dass die Tasks zu kleinteilig im Kanban-Board dargestellt werden und dadurch zu schnell Überschneidungen von Arbeitsbereichen entstehen. Es soll also eine engere Abstimmung untereinander stattfinden, um parallele Arbeiten an gleichen Funktionen von unterschiedlichen Personen möglichst zu minimieren.

---

[7] Schwaber, Ken/Jeff Sutherland: The Scrum Guide – The Definitive Guide to Scrum: The Rules of the Game, S. 9, in: scrumguides, 10.2020, https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf (abgerufen am 14.12.2021).
[8] Reddy, Ajay/Jack Speranza: Scrumban [R]Evolution, The: Getting the Most Out of Agile, Scrum, and Lean Kanban, PartIII, Kap. 5, 1. Aufl., London, England: Pearson Education, 2016.
[9] Trello 2021, https://trello.com/

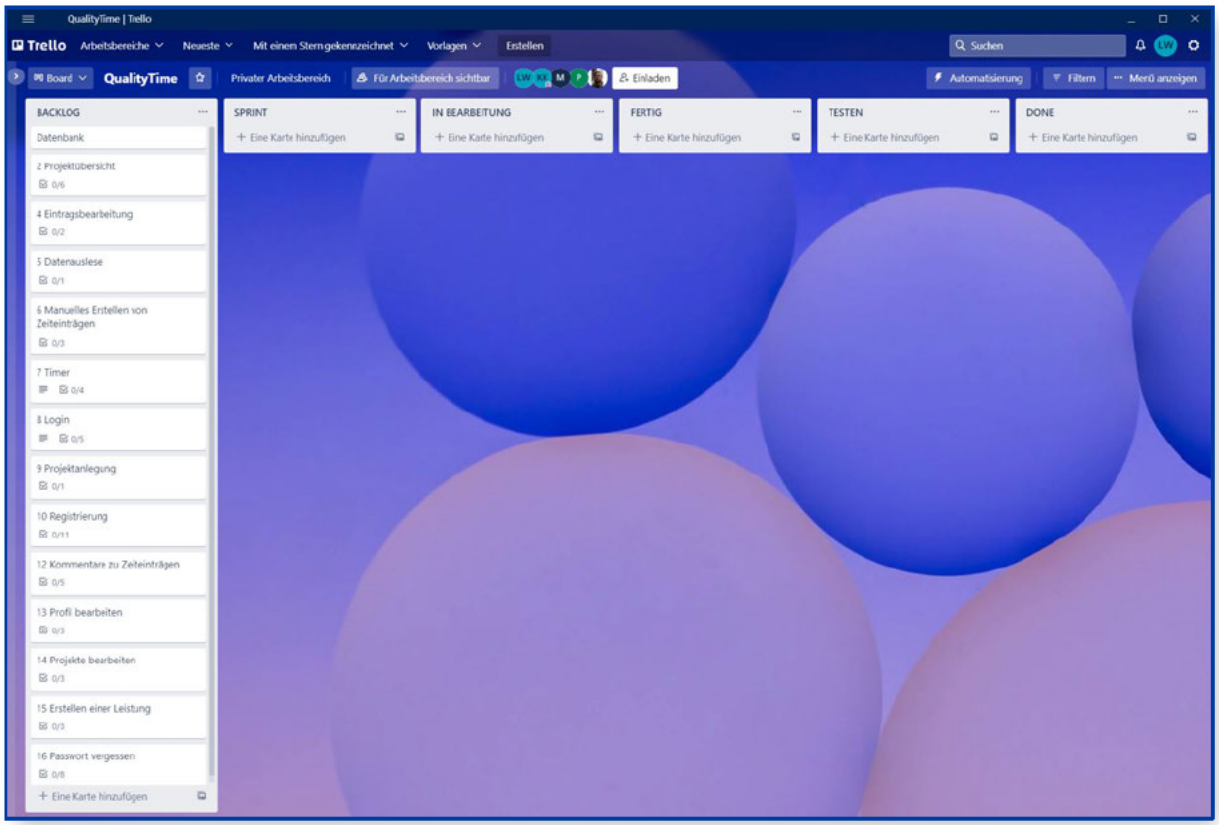


Abbildung 3.1: Screenshot von unserem Trello-Board (in Arbeit); https://trello.com/ (14.12.2021)

3.3 Teamrollen

Wie eingangs erwähnt haben wir als Team insgesamt bereits eine Mehrfachrolle: Wir sind selbst unsere Auftraggeber bezogen auf das Produkt (die Idee zur Realisierung einer Zeiterfassungssoftware stammt von uns), wir planen die Umsetzung und sind letztlich auch für das Development verantwortlich.

Das agile Vorgehen mit Scrumban erfordert keine spezifischen Rollen, wobei wir uns dennoch an den von Scrum vorgegebenen Rollen orientieren möchten, die sich auch bei uns im Team wiederfinden:
Die Rolle der *Product Owner* übernehmen wir gemeinsam. Im Rahmen der Anforderungsanalyse haben wir die Produktmerkmale und Anforderungen identifiziert, diese gemeinsam priorisiert und werden auch das Product Backlog stetig gemeinsam überprüfen, um den Projektfortschritt laufend zu kontrollieren.
Den *Scrum Master*, der vor allem darauf achten soll, dass der agile Prozess korrekt eingehalten wird, setzen wir mit der Rolle eines *Projektmanagers* gleich. Wir wollen so sicherstellen, dass nicht nur die Anforderungen der Software berücksichtigt werden, sondern auch die Anforderungen und Deadlines des Kurses „Software Engineering“. Die Rolle des Scrum Masters übernimmt Moritz [redacted].

Durch die Vorgabe, dass alle im Team auch programmieren sollen, ergibt sich automatisch, dass alle Teammitglieder zusammen das *Development Team* darstellen.

3.4 Aufwandsschätzung

Um den Aufwand der User Stories abzuschätzen haben wir gemeinsam den Prozess des *Planning Poker* durchlaufen. Wir haben dabei mit modifizierten Fibonacci-Zahlen als Story Points gearbeitet (0, 0.5, 1, 2, 3, 5, 8, 13, 20, 40, 100, ?) und als Basis eine für uns möglichst gut abschätzbare Aufgabe – das Erstellen einer Login-Maske – gewählt und mit acht Story Points beziffert (sowie mit einem tatsächlichen Zeitwert verknüpft: zwei Arbeitstage inkl. Testing, wobei ein Arbeitstag acht Arbeitsstunden entspricht), um davon ausgehend alle weiteren User Stories relativ einzuordnen.
Bis auf ein paar User Stories mit sehr niedriger Priorität bzw. Einordnung als optional, die wir aufgrund der knappen Zeit voraussichtlich gar nicht bearbeiten werden, wurden sämtliche Aufwände geschätzt.

Abbildung 3.2: Screenshot vom Planning-Poker-Prozess von "Planning Poker Online"; https://planningpokeronline.com/ (25.11.2021)

Wir gehen zunächst von einer *Velocity* (Schätzung pro Sprint, wie viel Arbeitszeit bzw. User Stories wir vom Product Backlog abdecken können) von zusammen (fünf Personen) 60 Stunden je Sprint (eine Woche) aus, also 1.5 Arbeitstage (à acht Arbeitsstunden) pro Person. Dieser Wert ist voraussichtlich recht optimistisch geschätzt, wird sich aber nach Abschluss des ersten „offiziellen" Sprints (siehe Erklärung unter Punkt *2) Arbeitsweise im Team*) genauer definieren lassen. Dann lassen sich unsere Erfahrungswerte einbringen, um ggf. die Leistung steigern oder das Sprint-Pensum verringern zu können.

Zum aktuellen Zeitpunkt kommen wir auf insgesamt 147 Story Points mit den User Stories, die mit der Priorität „hoch" oder „mittel" klassifiziert wurden (niedrig priorisierte oder optionale User Stories nicht inbegriffen). Gehen wir von unserer Basis aus, dem Login mit acht Story Points mit einem gegengehaltenen Zeitwert von zwei Arbeitstagen, kommen wir damit auf 147 Points ÷ 8 Points × 2 Arbeitstage × 8 Std. = 294 Stunden.
Teilen wir die 294 Stunden durch unsere derzeit geschätzte Velocity von 60 Stunden ergeben sich 4.9 Wochen bzw. knapp 5 Sprints bis zur Fertigstellung der wichtigsten Funktionen des Softwareprojekts. Erfolgt die Präsentation und Abgabe des Projekts Anfang Februar, bleibt somit noch ein kleiner Puffer übrig, der entweder genutzt werden könnte, falls sich die Entwicklung deutlich verzögern sollte (bspw. durch eine deutlich niedrigere Velocity oder viele unvorhergesehene Probleme), oder aber um sich doch noch um niedriger priorisierte Funktionen zu kümmern.

3.5 Versionierung

Um eine saubere Projektentwicklung zu gewährleisten, arbeiten wir mit *Git* [10] als Versionierungs-Tool.
Hierbei werden wir einen Main-Branch innerhalb des Repositories pflegen, in dem immer nur der finale Stand des aktuellen (getesteten, funktionsfähigen und theoretisch auslieferbaren) Inkrements liegen soll.
Neben diesem Hauptzweig wird ein Entwicklungszweig liegen, in dem die in separaten Branches entwickelten Komponenten zusammengeführt werden. Dieser Entwicklungszweig entsteht dann aus der Zusammenführung von weiteren Zweigen, die je Softwarebaustein angelegt werden. So wird einerseits sichergestellt, dass ein Zweig vorhanden ist, der nur der Zusammenführung von Komponenten dient (der Hauptzweig bleibt also unberührt, falls dieses Zusammenführen nicht reibungslos abläuft). Andererseits werden so nicht zu viele (und ggf. sogar voneinander unabhängige) Komponenten miteinander vermischt.

Allgemein gilt für die Arbeitsweise mit Git, dass nur Arbeiten in die entsprechenden Zweige gepusht werden, die alle Kriterien der Definition of Done erfüllen und somit vollständig überprüft, getestet und mit den Akzeptanzkriterien abgeglichen worden sind. Bei Änderungen sollen Pull-Requests erstellt werden, damit alle Teammitglieder einen möglichst aktuellen Stand vorliegen haben und um Merge-Konflikte möglichst zu

[10] Git, 2021. https://git-scm.com/

verringern. Durch das automatische zusammenführen von Git werden bereits viele Konflikte, die bei der alleinigen lokalen Arbeit am Code auftreten können, vermieden.

Da mit einer Datenbank gearbeitet wird, die (zumindest für die Entwicklung) nur lokal als Datei und nicht auf einem externen und für alle zugänglichen Server betrieben wird, wird diese in die *.gitignore*-Datei eingeschlossen. Änderungen können so zu Testzwecken lokal von jedem durchgeführt werden, ohne dass durch den Push in das Remote-Repository Merge-Konflikte aufgrund von unterschiedlichen Datensätzen entstehen. Am Anfang werden händisch abgestimmte Datenbankversionen mit Testdaten in das Repository eingefügt, um dieselbe Struktur für alle zur Verfügung zu stellen. Änderungen an der Datenbank, die dann für alle bzw. die Anwendung relevant und wichtig sein sollten, werden dann auf demselben Weg für alle zur Verfügung gestellt.

Durch die Versionskontrolle wird nicht nur sichergestellt, dass das gesamte Entwicklungsteam Zugriff auf denselben Code und dieselben Komponenten hat, sondern es lassen sich durch die Änderungshistorie auch besser sämtliche Vorgänge nachvollziehen und unter Umständen auftretende Fehler und Probleme besser identifizieren und schneller überprüfen, ob bestimmte Entwicklungen mit Sicherheit bereits stattgefunden haben. Änderungen, die trotz des Testprozesses ggf. zu Nichtfunktionalität der Software oder von einzelnen Teilen geführt haben und sich nicht ohne weiteres beheben lassen, können deutlich einfacher Rückgängig gemacht werden.

4 ENTWURF

Für die Umsetzung einer Idee in eine lauffähige Software ist eine genaue Planung (der Entwurf) entscheidend. Ein gut durchdachter Entwurf entscheidet maßgeblich über den Erfolg eines Projekts und eine reibungslose Organisation innerhalb des Entwicklungsteams.
Wir gehen zur Erläuterung unseres Entwurfs auf die nachfolgenden Punkte ein:

4.1 Softwarearchitektur

Helmut Balzert definiert Softwarearchitektur als Beschreibung von „*Strukturen eines Softwaresystems durch Architekturbausteine und ihre Beziehungen und Interaktionen untereinander sowie ihre physikalische Verteilung.*“ [11]
Mit dem Festlegen auf eine Softwarearchitektur legen wir also den Rahmenplan, wie unsere Software aufgebaut sein soll, aus welchen Teilen diese besteht und wie diese einzelnen Teile miteinander interagieren, fest. Wir wechseln also von den Anforderungen,

[11] Vgl. Balzert, Helmut: Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb, 3. Aufl. 2012, S. 23, Heidelberg, Deutschland: Spektrum Akademischer Verlag, 2011.

die beschreiben, was entwickelt werden soll, in den Prozess, der definiert, wie etwas entwickelt wird, um diese Anforderungen erfüllen zu können.

Um dies zu tun, ist zunächst ein Abstraktionsschritt nötig, in dem sich „je nach Abstraktionsgrad […] verschiedene Architektur-Bausteine unterscheiden" lassen.[12] Für unsere Anwendung können wir bspw. drei *Subsysteme* definieren:

Subsystem 1: Benutzeroberfläche
Wir entwickeln eine (dialogorientierte) Anwendungssoftware, benötigen also ein System, das entsprechende Daten anzeigen und Nutzereingaben entgegennehmen kann.
Zugehörige Beispiele aus dem Backlog: Übersichten, Eingabemasken, Buttons

Subsystem 2: Datenhaltung
Informationen bzw. Daten, die eingegeben oder ausgelesen werden, müssen (längerfristig oder temporär) gehalten werden.
Zugehörige Beispiele aus dem Backlog: Speicherung von Zeiteinträgen, Projekten, Usern, Kunden

Subsystem 3: Verarbeitung und Steuerung (Logik)
Eingaben, die auf der Benutzeroberfläche gemacht werden, müssen verarbeitet und interpretiert werden, ein (oder mehrere dieser Systeme) muss alle unterschiedlichen Komponenten zusammenhalten.
Zugehörige Beispiele aus dem Backlog: Filterfunktionen, Datenmanipulation

Diese Subsysteme lassen sich so auch in einem der sehr bekannten Architekturmuster wiederfinden: dem Model-View-Controller-Architekturmuster (nachfolgend *MVC* genannt). Es zähle zu denen, die dem Autor Joachim Goll in der Praxis besonders oft begegnen würden[13]. Und ein solches Muster einzusetzen, bietet sich schon deshalb an, weil es für eine bereits häufiger aufgetretene Aufgabe (hier: Entwicklung einer interaktiven Desktopanwendung) einen sich im Allgemeinen bewährten Lösungsvorschlag liefert. Das MVC werden auch wir für die Implementierung verwenden und den Vorteil davon im Nachfolgenden begründen.
Das MVC trennt seine Komponenten in ebenfalls drei Arten: In das *Model*, das für die Datenhaltung zuständig ist, die *View*, die zur Darstellung bzw. Ausgabe der Daten verantwortlich ist und den *Controller*, der Eingaben verarbeitet und die Daten des Models verändern kann. [14]
Diese Einteilung der Komponenten passt nicht nur wunderbar zu unserem Vorhaben, sondern vereint viele Vorteile. Bspw. wird dadurch die Einhaltung eines wesentlichen Architektur- bzw. Softwareentwicklungsprinzips vereinfacht: Die Trennung von Zuständigkeiten (engl. *separation of concerns)*. Dieser Begriff tauchte 1974 bereits in

[12] Vgl. Balzert, Helmut, 2012, S. 24
[13] Vgl. Goll, Joachim: Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java, 2., aktualisierte Aufl. 2014, Wiesbaden, Deutschland: Springer Vieweg, 2014, S. 288.
[14] Vgl. Goll, Joachim, 2014, S. 289

Edsger Wybe Dijkstras Veröffentlichung „On the role of scientific thought" auf[15], wurde wieder aufgegriffen (siehe z. B. „Elements of Functional Programming" von Chris Reade[16]) und ist ein etabliertes Prinzip, das Balzert wie folgt beschreibt:
„Jede Komponente einer Architektur ist bei dem Prinzip Trennung von Zuständigkeiten (separation of concerns) für eine Aufgabe oder einen Aufgabenkomplex verantwortlich. Die Aufgabe ist nicht über mehrere Komponenten verteilt. Eine Komponente ist also verantwortlich für eine einzige Aufgabe und erledigt diese Aufgabe richtig. […]" [17]

Im MVC-Modell hängt, wie Goll beschreibt, das *Model* dabei nicht von der Ein- und Ausgabe ab und es vereinfacht durch Trennung der Oberflächenkomponenten von der Verarbeitung die View und den Controller im System auszutauschen.[18] Dies ist auch im Hinblick auf die Projektarbeit im Team hilfreich, da so von unterschiedlichen Teammitgliedern auch flexibler und unabhängig programmiert werden kann (bspw. kann das Model unabhängig von View und Controller umgesetzt werden, da es nur Daten hält und nicht abhängig davon ist, wie an anderer Stelle diese Daten angezeigt oder interpretiert werden). Bei Änderungen am Controller und der View, die wir im Team aufgrund der geringen Entwicklungserfahrung voraussichtlich häufiger vornehmen werden müssen, kann das Model voraussichtlich unberührt bleiben. Das Model selbst sollte nicht so oft verändert werden müssen, da sich die Inhalte (die zu speichernden Daten) bereits sehr klar durch die Anforderungsanalyse ergeben haben.

Da wir Daten in unserer Anwendung speichern, die wir an unterschiedlichen Stellen auslesen und entsprechend auch unterschiedlich darstellen wollen (siehe auch *3.4 Diagramme zur Veranschaulichung, Komponentendiagramm*), eignet sich das MVC aus unserer Sicht sehr gut: Der Benutzer kann mit der Benutzeroberfläche interagieren bspw. in einer Eingabemaske Daten zu einem Zeiteintrag hinzufügen oder verändern. Diese Eingaben werden an den Controller weitergeleitet, um diese zu interpretieren und im Model festzuschreiben bzw. zu verändern. Auch ist der Controller in der Lage, eine oder mehrere Views anzuweisen, sich u.U. zu verändern, falls die Eingabe des Benutzers dies erforderlich macht (bspw. Felder ein- oder ausblenden). Da die Views sich entsprechend der in den Models enthaltenen (und zur Laufzeit ggf. veränderten) Daten anpassen können müssen, braucht es eine Möglichkeit diese Views über entsprechende Änderungen auch zu informieren.

Dies führt uns zum nächsten Punkt, den *Software Pattern*, die bereits einen geeigneten Lösungsvorschlag mitbringen, um diese Änderungsinformation möglichst gut realisieren zu können.

---

[15] Dijkstra, Edsger W.: Selected Writings on Computing: A Personal Perspective, Softcover reprint of the hardcover 1st edition 1982, 1982, S. 60-61
[16] Reade, Chris: Elements of functional programming 1. Auflage 1989, S. 2-3, Wokingham, England: Addison-Wesley Publishers Company
[17] Balzert, Helmut, 2012, S. 31
[18] Vgl. Goll, Joachim, 2014, S. 289

4.2 Entwurfsmuster

4.2.1 Beobachter-Muster

Während sich ein Architekturmuster, wie der Name bereits beinhaltet, auf die gesamte Softwarearchitektur bezieht, liefern Entwurfsmuster (engl. *design pattern)* Lösungen für spezifischere Aufgabenstellungen innerhalb des Systems.

Um die zuvor beschriebene Aufgabe (Views über Änderungen des Models informieren) zu lösen, liefert das s. g. „Beobachter-Muster" (engl. *Observer* Pattern) einen passenden Vorschlag. Die Absicht, die dieses „Observer"-Muster verfolgt, wird in „Design Patterns" (Erich Gamma et al.)[19] wie folgt beschrieben:

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

Und genau das ist, was wir möchten: Ein Objekt (unser Model) soll alle jeweiligen Beobachter (die Views, die Daten aus diesem Model darstellen sollen) über seine Zustandsänderung informieren. Auf diese Weise stellt eine View, die ein Model „beobachtet", immer zu dem Zeitpunkt die (neuen) Daten dar, wenn diese im Model verändert (und die vom Model informierten „Beobachter" informiert) werden. Das hat nicht nur den Vorteil, dass der Beobachter, also in diesem Fall die View-Komponente, das Model nicht durchgehend nach Änderungen fragen muss, sondern sich diese eben nur dann abholt, wenn es darüber informiert wurde. Auch kann ein beobachtetes Element (ein Model) theoretisch unendlich viele Beobachter (Views) haben. So können viele unterschiedliche Darstellungen in der Anwendung erreicht werden, die alle aus demselben Datensatz bestehen. Umgekehrt kann eine View genauso mehrmals werden, die Informationen aus einem anderen Model verwenden.

Seit JDK1.0 gibt es in Java die Klasse „Observable" und das Interface „Observer", welche die Umsetzung des Beobachter-Patterns leicht ermöglichen. Diese gelten seit Java 9 zwar als veraltet, da der Funktionsumfang recht eingeschränkt sei, die Reihenfolge der Benachrichtigungen des *Observables* an den *Observer* unspezifiziert wäre und Statusänderungen nicht eins zu eins mit Benachrichtigungen korrespondieren würden.[20] [21] Da es aber weiterhin für unseren Einsatzzweck gut funktionieren sollte und das Observer Pattern insgesamt so umsetzt, wie es bspw. auch in „Design Patterns"[22] beschrieben wird, werden wir es für unser Projekt einsetzen.

---

[19] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John; Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, "Behavioral Patterns, Observer"
[20] Vgl. Observable (Java SE 16 & JDK 16): in: Oracle | JDK 16 Documentation, o. D., https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Observable.html (abgerufen am 17.01.2022).
[21] Vgl. Observer (Java SE 16 & JDK 16): in: Oracle | JDK 16 Documentation, o. D., https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Observer.html (abgerufen am 17.01.2022).
[22] Vgl. Gamma et al. 1995, "Behavioral Patterns, Observer"

Anmerkung vom 17.01.2022:
Da wir uns mittlerweile in der Implementierungsphase befinden, lässt sich hierzu noch anmerken, dass die Einschränkungen, die diese in Java bereits enthaltene Klassen und das Interface mit sich bringen, sich an der ein oder anderen Stelle bereits bemerkbar gemacht haben. So ist es bspw. nicht optimal, dass der *Observer* vom *Observable* zwar über eine Änderung informiert wird, aber nicht darüber, was genau sich verändert hat. Das erschwert die Fallunterscheidung, falls sich in einer View (dem *Observer)* z. B. nur ein Teil aktualisieren soll. Eine auch von Oracle selbst empfohlene Möglichkeit wäre, einen s. g. *PropertyChangeListener* aus dem Paket *java.beans* zu implementieren. Dies ermöglicht, nicht auf die Änderung eines gesamten Objects zu reagieren, sondern auf einzelne Eigenschaftsänderungen dieses Objekts. Bislang kommen wir allerdings noch gut mit den *Observern* und *Observables* zurecht, weshalb wir nicht sofort alle Klassen abändern werden, was aber auch kein allzu großer Aufwand sein sollte.

4.2.2 Kompositum-Muster

Ein weiteres Entwurfsmuster, das automatisch durch die Verwendung von *Java Swing* als GUI-Framework Anwendung findet, ist das *Kompositum-Muster*. In diesem Muster werden Objekte zu Baumstrukturen (also hierarchisch) zusammengefügt, um aus mehreren kleinen Teilen ein größeres, zusammengesetztes zu machen. Diese können dann wieder miteinander komponiert werden.[23]
Java Swing verwendet bereits dieses Muster, um den Einsatz der einzelnen Komponenten zu realisieren: Aus verschiedenen kleinen Klassen (z. B. Panels, Buttons, Labels, Textfeldern etc.) können größere Komponenten zusammengesetzt (komponiert) werden. Dieses Entwurfsmuster muss von uns also nicht eigenhändig implementiert, sondern nur sinnvoll eingesetzt werden. So lassen sich bspw. aus mehreren kleinen Views eine größere View zusammensetzen (z. B. ein „Dashboard“).

4.2.3 Strategie-Muster

Das s. g. Strategie-Muster ist ein weiteres Muster, dass klassischerweise[24] bei der Umsetzung des MVC-Architekturmodells verwendet wird. Dabei werden laut Erich Gamma (et al.) in „Design Patterns“ Familien von Algorithmen definiert, zusammen gekapselt und austauschbar gemacht. So lassen sich die auszuführenden Algorithmen jeweils nach Client, der sie aufruft, variieren. [25] Bezogen auf das MVC-Modell können wir also dieselbe View auf unterschiedliche Weis auf Benutzereingaben reagieren lassen. Der zu einer View zugewiesene Controller kann dabei als Strategie angesehen werden. So ließen sich so bspw. unterschiedliche Verhalten realisieren, je nachdem mit welcher Rolle ein Nutzer eingeloggt ist.
In welcher Form bzw. ob wir das Strategie-Muster überhaupt einsetzen, ist nicht definiert.

[23] Vgl. Gamma et al. 1995, “Structural Patterns, Composite”
[24] Vgl. Goll, Joachim, 2014, S. 385
[25] Vgl. Gamma et al. 1995, “Behavioral Patterns, Strategy”

4.2.4 Kurz zusammengefasst

Zusammenfassend kommt das Beobachter-Muster also zwischen Model und View zur Benachrichtigung zum Tragen, das Kompositum-Muster bei dem Zusammenbau von Swing-Komponenten zu den von uns gewünschten Views und das Strategie-Muster findet beim Zusammenspiel von View und Controller Anwendung, wenn entschieden werden muss, wie auf Eingaben reagiert wird.

Neben den nun drei genannten Entwurfsmustern, die oftmals Anwendung im MVC-Modell finden, ist natürlich auch vorstellbar, dass weitere Muster hinzukommen bzw. bei auftretenden Herausforderungen unterstützen können. Die Entwurfsmuster können uns nämlich nicht nur dabei helfen, bereits vor der Implementierung bekannte Probleme möglichst gut zu bewerkstelligen, sondern ebenso gut beim Reorganisieren oder Verbessern von bereits geschriebenem Code.

4.2.5 Singleton-Pattern (nachträgliche Ergänzung)

Die Nutzung der Datenbank bzw. das Herstellen einer Verbindung zu dieser läuft über eine dafür geschriebene Controller-Klasse. Um nicht in allen Klassen, die diesen Datenbank-Controller verwenden müssen, eine neue Instanz davon erstellen zu müssen, implementieren wir das *Singleton Pattern*. Es sorgt dafür, dass sich die entsprechende Klasse selbst instanziiert und diese Instanz in einer privaten (damit nicht von außen veränderbaren) Variable speichert. Wollen andere Klassen von dieser Instanz Gebrauch machen, um bspw. die Datenbankverbindung über die entsprechenden Methoden zu verwenden, können sie über eine öffentliche Getter-Methode auf diese eine (*Singleton*) Instanz zugreifen. So entstehen einerseits nicht unnötig viele Objektinstanzen des Datenbank-Controllers, andererseits kann so die Datenbankverbindung offenbleiben, ohne dass diese von unterschiedlichen Instanzen blockiert und nicht nutzbar ist.

4.3 Datenbankstruktur

Da wir zum Speichern der Zeiterfassungsdaten eine Datenbank verwenden, müssen wir uns auch Gedanken über die dazugehörige Struktur machen.

Unser Ziel, das wir mit einer wohlüberlegten Datenbankstruktur anstreben, ist redundante Daten zu vermeiden ebenso wie Einfüge-, Lösch- und Änderungsanomalien. Dies erreichen wir mit der Beachtung der s. g. *referentiellen Integrität*, also dem Aufstellen von Regeln, wie Datensätze in die Datenbank eingefügt oder daraus gelöscht werden können.
Hierzu gehören z. B. die Normalisierung der Datenbanktabellen, die Verwendung eines Relationenmodells, die Arbeit mit eindeutigen Primär- und Fremdschlüsseln sowie der Einsatz von zusätzlichen Beziehungstabellen für n:m-Beziehungen.

Unsere Datenbank muss also so aufgebaut sein, dass konkret folgende Punkte bei den unterschiedlichen Aktionen berücksichtigt werden:

Löschen/Löschweitergabe/Aktualisierung

- Ein Projekt darf nur gelöscht werden können, wenn diesem Projekt keine User mehr zugeordnet sind. Andernfalls entstehen „Karteileichen“, da die Benutzer eine Referenz zu einem nichtexistenten Projekt enthalten.
- Auch User sollten nur dann gelöscht werden können, wenn sie keinem Projekt zugeordnet sind. Bspw. bei fehlerhafter Erstellung eines Users. Andernfalls wäre in Zukunft ggf. nicht mehr nachvollziehbar, welcher User an einem Projekt gearbeitet hat.
- Wird ein Projekt gelöscht, dürfen nicht automatisch alle zugeordneten Benutzer ebenfalls gelöscht werden (Löschweitergabe)

Eintragen von Daten

- Es darf nicht möglich sein, einen Benutzer zu einem Projekt hinzuzufügen, das nicht existiert. Umgekehrt gilt dasselbe.

Abfrage/Sortieren

- Bei festen Zuordnungen (z. B. welche Benutzer sind welchem Projekt zugeteilt) muss diese Zuordnung aus den Datenbankabfragen ersichtlich werden

4.4 Diagramme zur Veranschaulichung

4.4.1 Komponentendiagram

Das Komponentendiagramm zeigt die Struktur der Anwendung und ihre einzelnen Komponenten. Es veranschaulicht, welche Schnittstellen von welchen Komponenten benötigt und welche bereitgestellt werden.

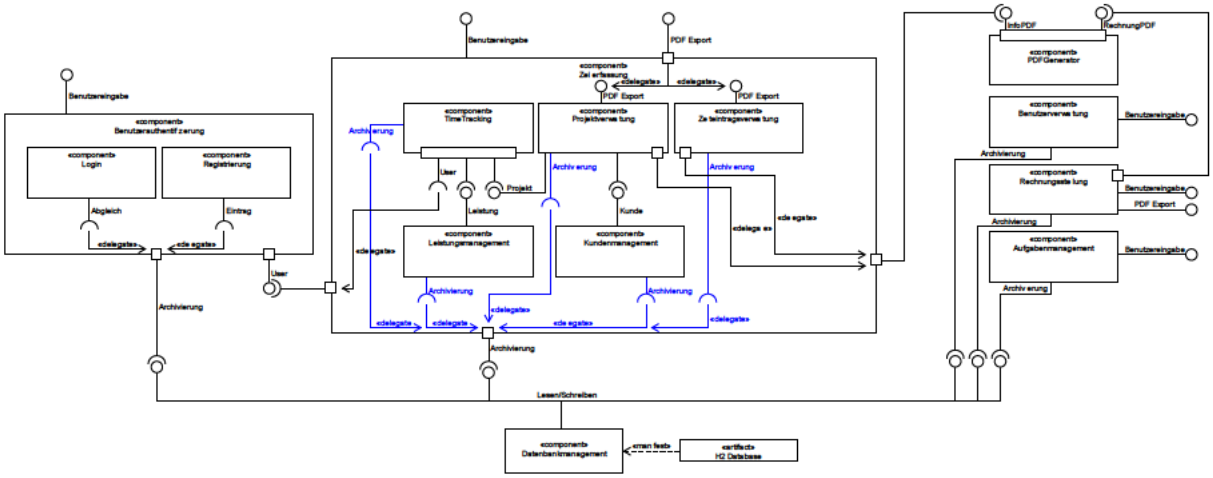


Abbildung 4.1 Komponentendiagramm

4.4.2 Klassendiagramm

Das Klassendiagramm zeigt die Beziehungen zwischen den einzelnen Klassen, hier am Beispiel des *Timers*, der in der Lage ist, die Zeit zu stoppen, um darauf basierend einen Zeiteintrag zu erstellen. Diese für alle weiteren Features stellvertretende Darstellung zeigt die Implementierung des MVC-Modells: Die datenhaltenden Models erben von einer Klasse *Observable*. Die *View*, die eine Benutzeroberfläche darstellt, implementiert das Interface *Observer*. So kann sich eine View bei einem Model anmelden und wird fortan über Updates informiert. Der Controller, der eine View instanziiert und damit als *Strategie* fungiert, hört dabei auf Aktionen, die innerhalb der View stattfinden, um diese dann zu interpretieren und zu verarbeiten. Der Controller ist außerdem in der Lage, die Daten in den Models zu verändern.

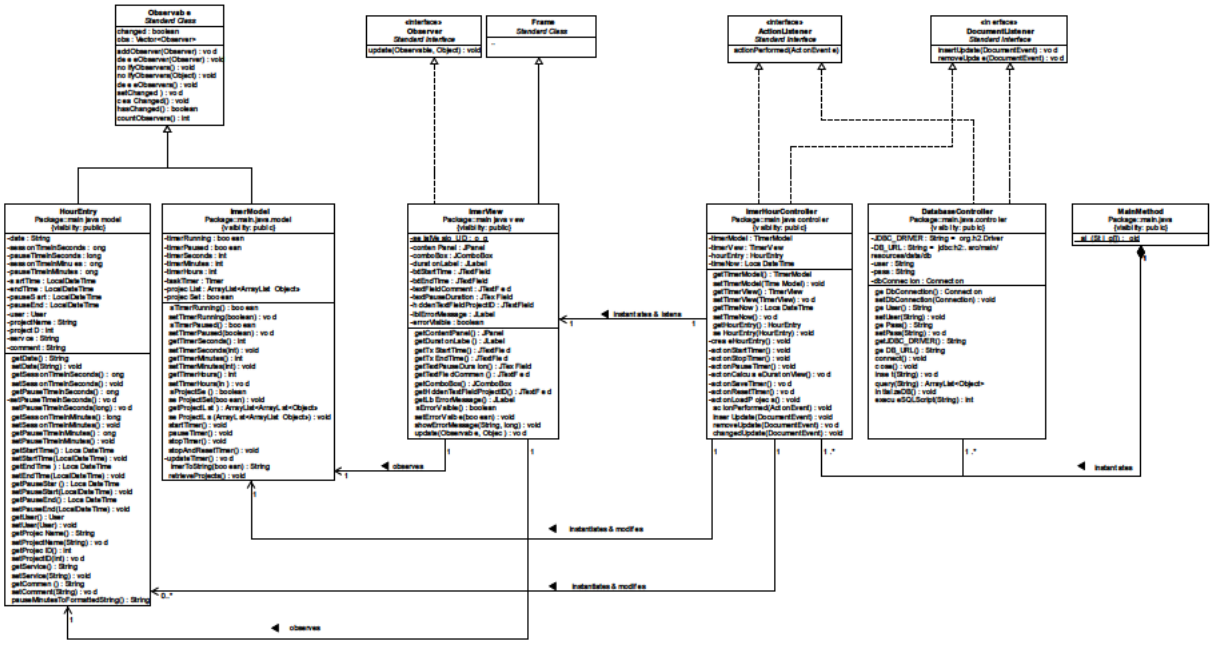


Abbildung 4.2 Klassendiagramm

4.4.3 Datenbank: Entity-Relationship-Modell

Als Basis für den Datenbankentwurf wurde zuerst ein Entity-Relationship-Modell (ERM) erstellt. Grundlage dafür bilden die Entitäten (Rechteck), Beziehungen (Raute) sowie die Attribute (kreisförmig). Unter Entitäten versteht man eindeutig identifizierbare Objekte, die verschiedene Eigenschaften (Attribute) haben können. Im Entity-Relationship-Modell zeigen wir, in welcher Beziehung die verschiedenen Entitäten zueinanderstehen.

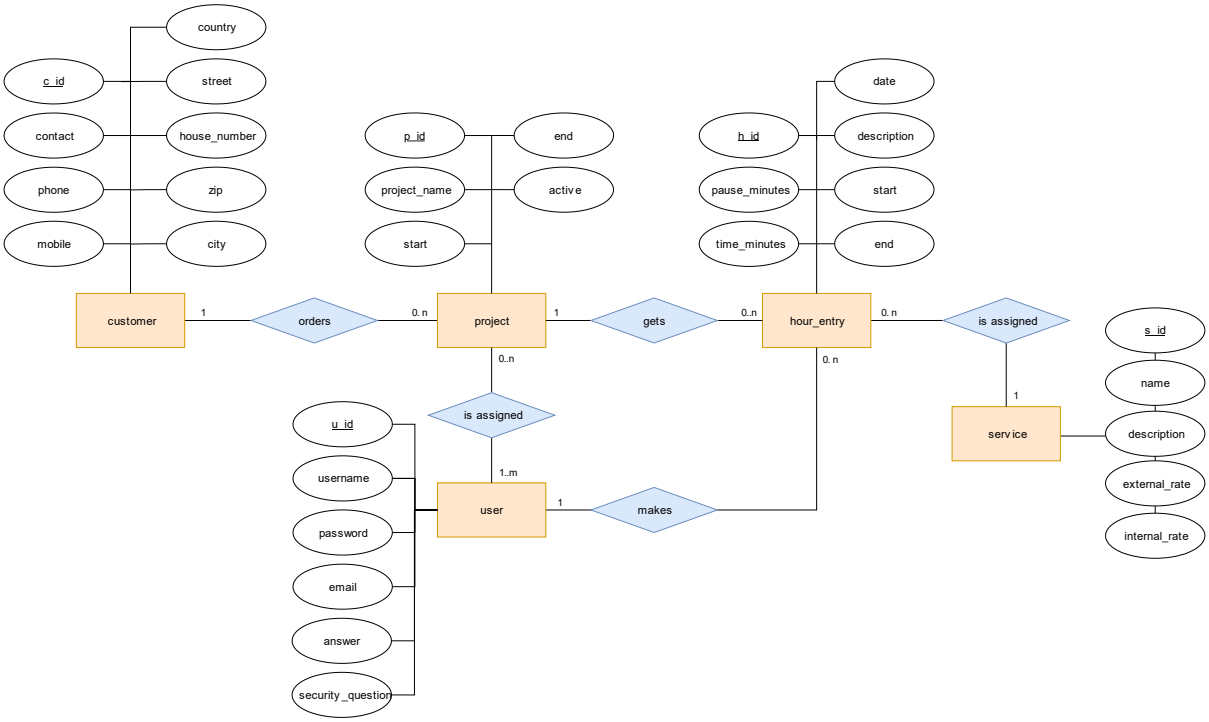
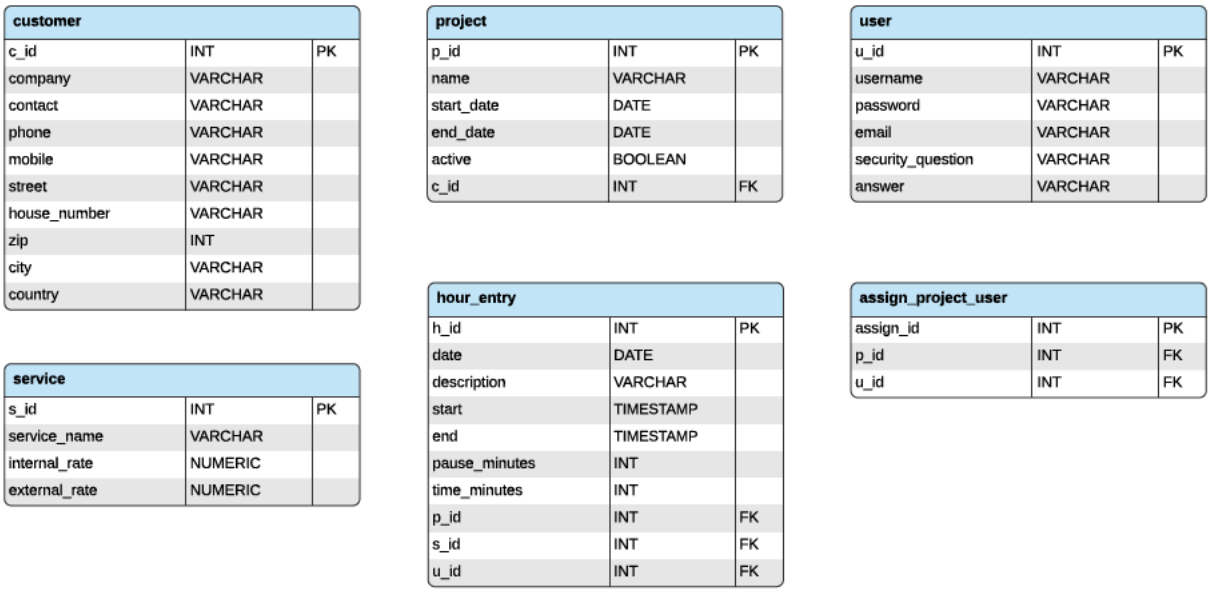


Abbildung 4.3 Entity-Relationship-Modell

4.4.4 Datenbank: Relationenmodell

Damit wir aus dem bestehenden ERM eine relationale Datenbank bauen können, haben wir zunächst das ERM in ein Relationenmodell überführt. Hierzu wurden alle Beziehungen zwischen den Entitäten entsprechend aufgelöst und in entsprechende Tabellen übertragen (Grundüberlegungen dazu siehe Punkt 1.3). Bei 1:n Beziehungen wurde die 1-Entität als Fremdschlüssel in der Tabelle der n-Entität eingefügt. Bei n:m Beziehungen wurde eine zusätzliche Tabelle erstellt, welche die Primärschlüssel der jeweiligen Entitäten als Fremdschlüssel enthält.

Das Relationenmodell zeigt also, wie sich aus dem ER-Modell (siehe 1.4.3) eine relationale Datenbank ableiten lässt. Es stellt dabei die Tabellen, Attribute, die zugehörigen Datentypen sowie alle benötigten Schlüssel dar, sodass nun eine praktische Umsetzung mit der von uns gewählten H2 Datenbank Engine erfolgen kann.



Relationenmodell

5 IMPLEMENTIERUNG

Im Nachfolgenden beschreiben wir Themen, die sich auf unser Vorgehen bei der Implementierung beziehen, und machen Anmerkungen zu Punkten, die uns während der Arbeit oder im Nachgang aufgefallen sind. Einer der größten Faktoren, die eine möglichst saubere Implementierung negativ beeinflusst haben, war neben der anfänglichen Unwissenheit im Umgang mit neuen Themen (z.B. Java Swing) der Faktor Zeit. Wie sich unserer Zeiterfassung entnehmen lässt (siehe 8.1 Zeiterfassung), haben wir nicht wenig Zeit investiert und dennoch hat sie uns immer wieder gefehlt. Das hat, wie sich nachfolgend lesen lässt, dazu geführt, dass uns viele Stellen mit Verbesserungspotenzial zwar oft bewusst sind, die Zeit zur Umsetzung der möglichen und teilweise benannten Korrektur aber nicht mehr ausgereicht hat.

5.1 Implementierung MVC

Das MVC-Modell wurde, wie in der Entwurfsphase geplant, für die Implementierung der geforderten Komponenten konsequent angewendet und das Programm nach dem Schema in Packages eingeteilt (*main.java.model*, *main.java.view*, *main.java.controller*), um die Einhaltung des Modells zu unterstützen. Dennoch ist die Art wie das MVC implementiert wurde nicht über alle Klassen hinweg gleich konsequent umgesetzt worden. Das liegt bzw. lag daran, dass im Team zwar ein grundlegendes Verständnis für das Modell vorhanden ist, die genaue Implementierungsweise sich aber erst mit steigendem Fortschritt langsam gefestigt hat. Strengere Codereviews hätten hier grundsätzlich sehr geholfen, waren aber aufgrund mehrerer Faktoren nicht in dem gewünschten Umfang immer realisierbar (siehe hierzu auch Abschnitt *5.3* Codereview*)*. Grundsätzlich wurde im aktuellen Stand zur Abgabe aber möglichst versucht, die Implementierung des MVC gleich zu halten.

Häufige Probleme dabei waren bspw. die anfängliche Unklarheit darüber, welche Klasse welche andere Klasse instanziieren darf bzw. soll oder in welcher Reihenfolge dies geschehen soll. Das Klassendiagramm (siehe 4.4.2) hat dabei geholfen sich den Aufbau immer wieder zu vergegenwärtigen: Der Controller instanziiert eine View und ggf. benötigte Models. Bei Instanziierung des View-Objekts, übergibt sich der Controller selbst über den Konstruktor der View, damit diese den Controller später zum Hinzufügen als *ActionListener* zur Verfügung hat.

Allerdings gibt es auch Stellen, an denen wir von der eigentlichen MVC-Implementierung abgewichen sind: So werden zwar grundsätzlich zur Laufzeit länger benötigte Daten in Objekten eigener dazu gehöriger Model-Klassen gehalten, gleichzeitig kommt es durch unsere Implementierung aber auch vor, dass der Controller bspw. direkt Daten aus der Datenbank abruft oder Fehlermeldungen generiert, die dann direkt an die View weitergegeben werden. Dies haben wir größtenteils bewusst so entschieden, um unserem Zeitmangel entgegenzuwirken und nicht für jeden kleinteiligen Datensatz auch

eine eigene Klasse erstellen und Objekte „um der Objekte willen“ instanziieren zu müssen. Letzteres hätte für stärkere Unübersichtlichkeit gesorgt, wie es beispielsweise noch in der Klasse *main.java.controller.TimerHourController* der Fall ist, die direkt am Anfang des Projekts entstand. Wir haben hier versucht möglichst strikt die Daten nur in einem Model zu speichern und niemals im Controller. Das führte aber schnell dazu, dass der Controller zunächst Daten aus der View ausliest, die die Benutzereingabeelemente bereitstellt, nur um sie dann direkt aus dem Model wieder auszulesen und in die Datenbank zu schreiben. Die Klasse wurde dann teilweise auch wieder etwas überarbeitet, da sie aber eine der Kernfunktionen implementiert und bereits recht früh bestand, haben wir damit weitergearbeitet, da sie bereits funktionierte. In den darauffolgend erstellten Klassen sind wir daher von dieser aus unserer Sicht etwas umwegigen Implementation abgewichen. Dass der korrekte Weg hier eigentlich die kleinteilige Implementierung (einer der Nachteile im MVC) wäre, ist uns bewusst.

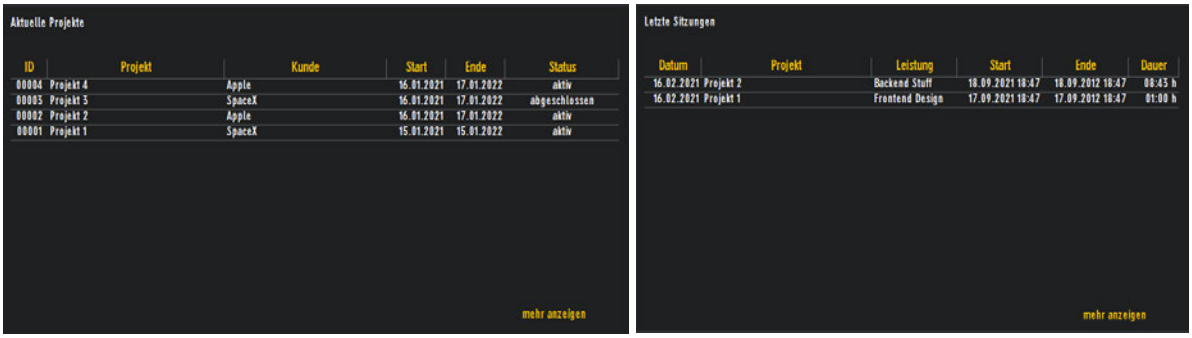
Der Plan, dass sich die Teamarbeit durch das MVC durch die Möglichkeit unabhängiger an Komponenten zu arbeiten vereinfacht, ist aber durchaus aufgegangen.

5.2 Abstraktion & Redundanzen

Eine starke Abstraktion verbessert die Arbeit an Code und die Wartbarkeit eines Software-Projekts sehr. Mag uns das im Entwurf und in der Planung des Projekts noch gut gelungen sein, hätte der Abstraktionsgrad in der Implementierung noch deutlich höher sein können.

Die Umsetzung von sich stark ähnelnden Komponenten durch unterschiedliche Teammitglieder hat bspw. oft dazu geführt, dass redundante Methoden (weil ähnlich oder gleich) geschrieben wurden, Views, die sich eigentlich nur bezogen auf die Art der darzustellenden Daten unterscheiden, mehrfach entwickelt oder unterschiedliche Klassen für gleiches Verhalten erstellt wurden. Refactorings wurden hier zwar teilweise unternommen, jedoch aufgrund der Zeit und des bis zum Bemerken bereits vorhandenen Fortschritts nicht in dem Umfang, in dem es sinnvoll gewesen wäre.

Ein Beispiel, das den Versuch einer kleinen Verbesserung durch Nutzung des *Strategie-Patterns* beschreibt, aber weiter gedacht mit zusätzlichen Komponenten der Anwendung noch stärker abstrahiert werden könnte:
Im *Dashboard* der Anwendung befinden sich zwei Tabellen (*Aktuelle Projekte* und *Letzte Sitzungen*), die grundsätzlich gleich aussehen, aber unterschiedliche Daten darstellen. Um hier nicht zweimal dieselbe View in separaten Klassen zu definieren, wurde eine übergeordnete „Listen-Klasse“ geschrieben, die von zwei unterschiedlichen Controllern (den Strategien) instanziiert wird und sich somit auch unterschiedlich verhalten sowie entsprechend andere Daten anzeigen kann.



Die Controller hingegen weisen dann allerdings wieder starke Redundanzen auf, die man sich eigentlich ebenfalls noch sparen könnte.
Ansätze zur Verbesserung sind also vorhanden, aber noch nicht konsequent umgesetzt.

5.3 Scrumban & Teamarbeit

Die Arbeit und Organisation im Team haben grundsätzlich gut funktioniert. Wie unter Punkt *3.2 Arbeitsweise im Team* bereits beschrieben, sind wir vom eigentlichen Scrumban-Vorgehen etwas abgewichen und hatten Sprints, die eine Woche gedauert haben, sowie wöchentliche Meetings, die gleichzeitig als Review sowie zur Sprint-Planung gedient haben. Und hier liegt wohl auch eine der größten Herausforderungen zur guten Arbeit mit Scrum bzw. Scrumban. Um die Teamarbeit und damit die Implementierung noch weiter zu verbessern, wäre eine höhere Frequenz an Meetings deutlich besser gewesen. Da dies für uns zeitlich nicht machbar war, hat sich alles auf ein wöchentliches Meeting konzentriert: Statusupdates, Problembesprechung, Sprintplanung, Code-Review, Zusammenführen von Code. Die Fülle der Themen hat die Meetings entsprechend langgezogen, gleichzeitig konnten aber nicht immer sämtliche Themen komplett besprochen werden, worunter beispielsweise eben auch das Code-Review gelitten hat.

Mit der Aufwandsschätzung am Anfang hatten wir zunächst eine Velocity von ca. 60 Stunden pro Woche geschätzt, um dann nach jedem Sprint zu schauen, wie hoch dieser Wert tatsächlich liegt. Die Planung der Sprint-Items aufgrund der vorhergegangenen Velocity war in der Praxis aber gar nicht machbar, sodass wir diesen Wert nicht weiter beachtet haben. Der Zeitaufwand, den jeder in das Projekt pro Woche investieren konnte, hat so stark variiert, dass auf Basis der Zeiterfassung gar keine gelungene Planung hätte stattfinden können.
Auch muss gesagt sein, dass wir Sprints zwar nie offiziell von der festgelegten Dauer von einer Woche auf einen längeren Zeitraum gestreckt haben, wenn wir mit bestimmten Tasks nicht fertig geworden sind. Allerdings kam es recht häufig vor, dass wir mit den im jeweiligen Sprint geplanten Tasks nicht immer alle auch fertiggeworden sind, was dazu geführt hat, dass wir diese Tasks auch im darauffolgenden Sprint beibehalten haben. Abgesehen von den trotzdem wöchentlich erstellten lauffähigen Versionen der Anwendung, hat sich das wie ein verlängerter Sprint angefühlt.
Die Verzögerung von Tasks lag aber nicht immer an Zeitmangel, sondern auch daran,

dass sich teilweise neue Abhängigkeiten zwischen den Tasks herauskristallisiert haben, weshalb manche Tasks erstmal pausiert werden mussten.

Rückwirkend kann man sich die Frage stellen, ob ein plangesteuertes Vorgehensmodell für dieses Projekt (bezogen auf eine saubere Implementierung) so viel schlechter gewesen wäre. Unser Plädoyer für die agile Vorgehensweise mit Scrumban war, dass eine strenge Planung aufgrund anfangs fehlender Kenntnisse nicht gut genug möglich gewesen wäre, dass wir auf sich ändernde Anforderungen durch die Flexibilität mit Scrumban gut reagieren können. Und über das gesamte Projekt hat uns das auch tatsächlich diese Vorteile verschafft. Dennoch öffnet die Freiheit durch diese Flexibilität auch wieder Türen für unsauberes Vorgehen. Da die kleinen, eigentlich täglich anstehenden Meetings bei uns nicht stattfanden und alles auf einen wöchentlichen großen Termin fiel, konnten wir nicht genug vor jedem Sprint planen, wie die Umsetzung möglichst gut gestaltet werden kann. Gerade diese Flexibilität erfordert es wahrscheinlich, dass man sich gut mit der Materie auskennt, um zu wissen, wie man auf auftretende Herausforderungen reagiert. Da wir uns beispielsweise im Umgang mit Swing und dem MVC-Modell immer nur schrittweise von einem unbekannten Problem zur nächsten Lösung hangeln konnten, weil der große Überblick aufgrund fehlender Erfahrung noch nicht vorhanden ist oder war, haben sich eben die vorher benannten Unsauberkeiten in das Projekt eingeschlichen.

Dennoch lässt sich festhalten, dass die allgemeine Arbeitsweise im Team sowie Organisation der Aufgaben mit Scrumban gut geklappt hat. Durch das Kanban-Board war jedem bekannt, wer mit welchen Aufgaben betreut ist und durch die Arbeit mit Trello als Kanban-Tool konnte teilweise auch task- oder User-Story-spezifische Kommunikation hierüber stattfinden, was die Zusammenarbeit, die ja durchgehend „remote“ stattgefunden hat, deutlich erleichtert hat.

5.4 Code-Conventions & Code-Review

Was die allgemeinen Java-Code-Conventions[26] betrifft, haben wir versucht uns alle möglichst an die üblichen Vorgaben zu halten. Zwar wird man bei Durchsicht des Codes noch Variablen finden können, die nicht immer streng nach Lower-Camel-Case definiert (bspw. mit *Underline* zur Trennung von Worten) oder Methoden, die nicht perfekt kommentiert worden sind. Auch ist bspw. eine maximale Zeilenlänge von bis zu 80 Zeichen nicht streng eingehalten worden und nur an manchen Stellen nachträglich korrigiert worden. Grundsätzlich sollten aber die „wichtigsten“ Konventionen von uns eingehalten worden sein. Wichtige oder umfangreichere Methoden wurden mit Kommentaren beschrieben, um möglichst schnell erfahren zu können, welche Funktion diese haben.

[26] Vgl. Code Conventions for the Java TM Programming Language in: Oracle, 1999, https://www.oracle.com/java/technologies/javase/codeconventions-contents.html (abgerufen am 17.01.2022).

Was die Code-Reviews betrifft, so ließ sich in den vorhergehenden Punkten bereits lesen, dass diese nicht in dem Umfang, der nötig gewesen wäre, stattfinden konnten. So wäre es wünschenswert gewesen, dass bspw. mit jedem Hochladen in die Versionskontrolle oder aber spätestens vor einem Code-Merge mit weiteren Komponenten der geschriebene Programmcode vorher noch einmal genau „unter die Lupe“ genommen und korrigiert wurde. Praktisch sah es allerdings so aus, dass der Code vor allem durch den verantwortlichen Programmierer selbst noch einmal geprüft und hinterfragt wurde, ein Review durch eine andere Person aber erst sehr viel später und dann aufgrund der Menge meist auch nur recht oberflächlich stattfand. Dadurch wurde oftmals nur sehr offensichtliches korrigiert.

5.5 Versionierung

Die Versionierung mit Git hat stets gut funktioniert und durch klare Aufgabenverteilung sowie durch Verwendung Feature-spezifischer Branches gab es nur sehr wenige und vor allem anfangs Merge-Konflikte.
Zu Beginn wurde noch mit sehr vielen kleinen Branches gearbeitet, die mit Fortschritt des Projekts dann aber wieder aufgelöst und zusammengeführt wurden, bis am Ende nur noch sehr wenige verwendet wurden.

Der Plan war mit jedem Sprintende auch eine lauffähige Version in den main-Branch, der nur mit Pull-Requests aktualisiert werden konnte, hochzuladen. Diese lauffähige Version war am Ende eines Sprints zwar stets vorhanden, jedoch wurde nicht immer konsequent daran gedacht, diese Neuerungen auch im main-Branch zu veröffentlichen.

Was die *Commit-Messages* betrifft, so beschreiben diese überwiegend die gemachten Änderungen. Trotzdem (vor allem in den Nachrichten zum Anfang des Projekts) lassen sich auch Nachrichten finden, die sehr allgemein gehalten sind.
Verbesserungspotential haben die Commits ohnehin, da diese deutlich atomarer gestaltet sein könnten. Der Großteil davon betrifft nämlich oftmals recht große und nicht immer zwingend zusammenhängende Änderungen. Das hat zwar in unserem Fall keine großen Auswirkungen auf die Projektarbeit gehabt, wird aber dann problematisch, wenn einige der Änderungen fehlerhaft sind und zurückgesetzt werden müssten. Durch das Zurücksetzen würden entsprechend sämtliche Änderungen eines Commits wieder verworfen werden.

5.6 Javadoc

Zu Dokumentationszwecken haben wir Methoden, die etwas komplexer sind, mit Javadoc-Kommentaren versehen. Diese sind einerseits in dem exportierten Javadoc zu finden, werden andererseits aber auch in der IDE angezeigt, um besser zu veranschaulichen was eine Methode für Argumente erwartet, was sie zurückgibt und wie sie (im Groben) funktioniert.

6 TESTING

6.1 Grundsätzliche Vorgehensweise

Als Basis hat während der Entwicklungszeit jedes Team seine Methoden eigenständig getestet. Dies passierte in den meisten Fällen manuell. Da bei vielen unserer Methoden der Nutzen von isolierten Tests nicht sinnvoll erscheint, haben wir uns dazu entschieden, die Haupttestaufgaben mittels Integrations-/GUI-Test zu realisieren. Dieser testet mittels eines Robots der User Eingaben simuliert, einzelne Funktionen der jeweiligen Module, sowie das Zusammenspiel aller Klassen und Methoden durch verschiedene Nutzungsszenarien. Dabei werden z.B. alle Elemente auf Sichtbarkeit und Funktionalität durch simulierte User-Interaktionen, wie das Klicken auf Buttons und verschiedene Eingabe-Abfolgen getestet. Da in den Konstruktoren teilweise einiges passiert, gibt es zusätzlich für viele Klassen eine Testklasse, die zumindest den Konstruktor aufruft. Dort wo es sinnvoll erscheint, finden zudem isolierte Unit Tests statt (z.B. DatabaseControllerTest). Private Methoden und Methoden ohne Seiteneffekte werden nicht getestet.

6.2 Tools

Zur Implementierung der Tests wurden JUnit und AssertJ-Swing verwendet. AssertJ-Swing enthält Klassen und Methoden, die das Testen von Swing Applikationen vereinfachen.

6.3 Test-Implementierung

Die Testfälle besitzen sprechende Methodennamen, die von der ansonsten verwendeten CamelCase-Notation abweichen. Dieses dient der Übersichtlichkeit, da ansonsten aufgrund der Länge der Methodennamen die Lesbarkeit schlecht ist. Die Methodennamen sind nach folgendem Schema aufgebaut:

- MethodName_StateUnderTest
- MethodName_StateUnderTest_ExpectedBehavior
- MethodName_ExpectedBehavior_WhenTheseConditions

6.3.1 Integrations-/ GUI-Test

Damit mehrere Aktionen hintereinander getestet werden, ohne eine neue Instanz zu bilden, testet der Integrationstest anhand verschiedener Hilfsmethoden alle Testszenarien hintereinander.

Video: Demonstration des Integrationstests

Man sieht in vielen Testklassen, dass im Gegensatz zu klassischen JUnit-Tests verhältnismäßig wenig Assert-Methoden zu sehen sind. Das ist aufgrund der Verwendung des AssertJ-Swing Frameworks auch nicht nötig. Ein Test schlägt fehl, sobald z.B. ein

Button nicht geklickt werden kann, der eigentlich klickbar sein sollte oder ein Element nicht sichtbar ist, welches sichtbar sein sollte.

Im folgenden Screenshot ist zu sehen, wie in Zeile 41 mittels der Klasse GuiActionRunner der Zugriff auf unsere Swing Komponenten hergestellt wird. Der Zugriff auf die Elemente passiert dann durch Verwendung der AssertJ-Swing API.

```
29  class IntegrationTest {
30          private FrameFixture window;
31
32          @BeforeAll
33          public static void setUpOnce() {
34                  FailOnThreadViolationRepaintManager.install();
35          }
36
37          @BeforeEach
38          public void setUp() {
39                  new LayoutManager();
40                  User.setUser(new User(1, "Bob", "bob@msn.com"));
41                  AppMainView frame = GuiActionRunner.execute(() -> new AppMainController().getAppMainView());
42                  window = new FrameFixture(frame);
43                  window.show(new Dimension(1850, 1080)); // shows the frame to test in needed Dimension (see AppMainView Class)
44                  frame.setFrameToLeftUpperCorner();
45          }
46
47          @AfterEach
48          public void tearDown() {
49                  // After each test case we close all windows, deallocate Mouse keys etc.
50                  window.cleanUp();
51          }
```

Die Klassen für Login und Registrierung werden autark mittels GUI-Tests getestet. Die Funktionalität des Logins, Registrierung, Hashing etc. wurde manuell getestet, da hier aus Sicherheitsgründen die meisten Methoden von außerhalb der Klasse oder des Pakets nicht zugänglich sind.

Separate Unit Tests wurden z.B. für die Klassen DatabaseController und LayoutManager implementiert.

Die Funktion „Passwort vergessen“ wurde zu einem späten Zeitpunkt hinzugefügt und wurde daher manuell getestet.

6.3.2 Whitebox Testing

Zusätzlich zu den automatisierten Tests haben alle Beteiligten ihre Methoden und Programmbausteine eigenständig auf Funktionalität geprüft. Dazu wurden immer der „happy path“ getestet, bei dem alles wie gewünscht ausgeführt wird, sowie mögliche Ausnahmezustände provoziert, durch verschiedene Kombinationen von Fehleingaben.

6.3.3 Blackbox Testing

Um einen weiteren Test eines nicht am Projekt beteiligten Benutzers zu erhalten, wurde einmal ein Test aus normaler Benutzersicht durch einen Kommiltonen durchgeführt. Ein DAU-Test (Dümmster anzunehmender User) wurde ebenfalls durchgeführt. Dazu haben wir einen weiteren Kommilitonen bemüht, der versucht hat, das Programm durch viele mögliche Eingaben zum Absturz zu bringen bzw. Fehler zu provozieren.

6.3.4 Greybox Testing

Ein DAU-Test wurde zusätzlich noch von einem Teammitglied (in-house-Tester), durchgeführt.

Alle Fehler, die uns bekannt geworden sind, wurden in eine Liste eingetragen. Zu großen Teilen wurden diese bereits behoben, jedoch gibt es immer noch ein paar Fehler, die zwar bekannt sind, aber noch gefixt werden müssten.

6.3.5 Bekannte Fehler (noch nicht behoben)

- Das Dropdown-Feld „Projekte“ im Dashboard aktualisiert sich, nachdem ein neues Projekt angelegt wurde, nicht automatisch. Man muss erst den aktualisieren Button klicken
- In der Projektübersicht wird das Dropdown-Feld „Kunde“ erst nach erneutem Einloggen aktualisiert

7 FAZIT

7.1 Endprodukt

Schon zu Beginn der Planung war uns klar, dass wir nicht sämtliche Anforderungen an die Anwendung auch in der gegebenen Zeit realisieren können werden. Wir haben uns deshalb auf die Umsetzung der User-Stories mit einer hohen Priorität fokussiert und diese auch planmäßig umsetzen können. Teilweise haben wir auch User-Stories abarbeiten können, die eine niedrigere Priorität hatten und damit für die Anwendung nicht essenziell waren (z. B. Design oder Benutzerfreundlichkeit, die definiert war als „nicht mehr als drei Klicks zu jeder Funktion“).

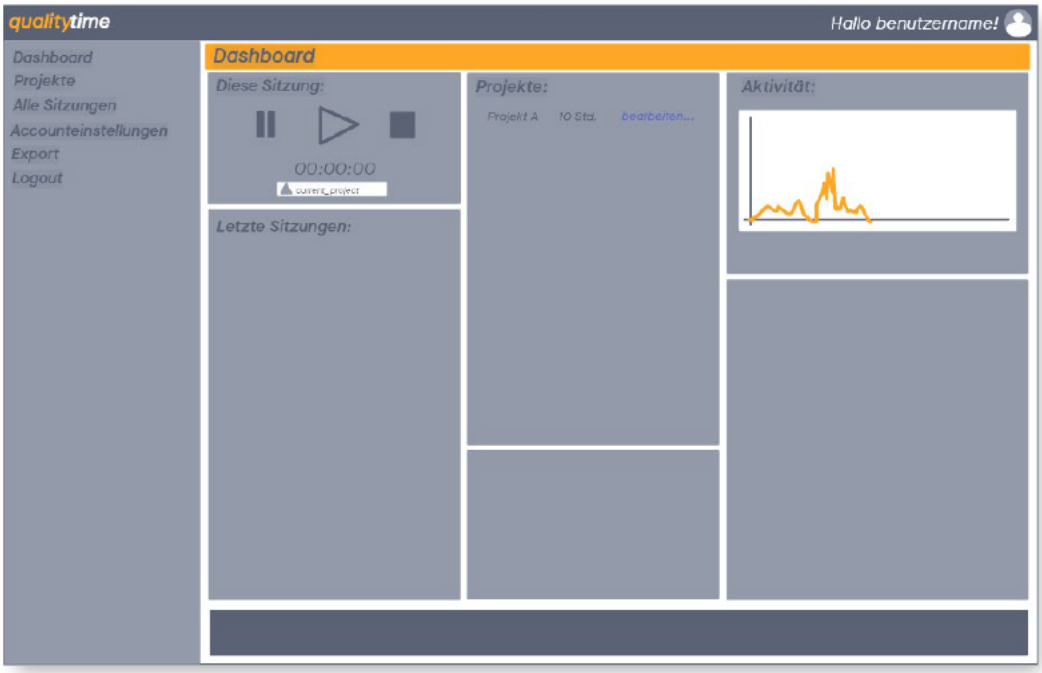


Abbildung 7.1 Mockup zu Beginn des Projekts

Insgesamt ist durch den zwar durchaus noch verbesserungswürdigen, aber insgesamt recht reibungslos durchlaufenen Prozess des Software Engineerings eine lauffähige Anwendung entstanden mit der wir mehr als zufrieden sind: Sie erfüllt die Akzeptanzkriterien der für die Anwendung wichtigsten User-Stories sowie unseren eigenen Anspruch (der sich zwar mit den Akzeptanzkriterien deckt, aber auch noch darüber hinaus geht).

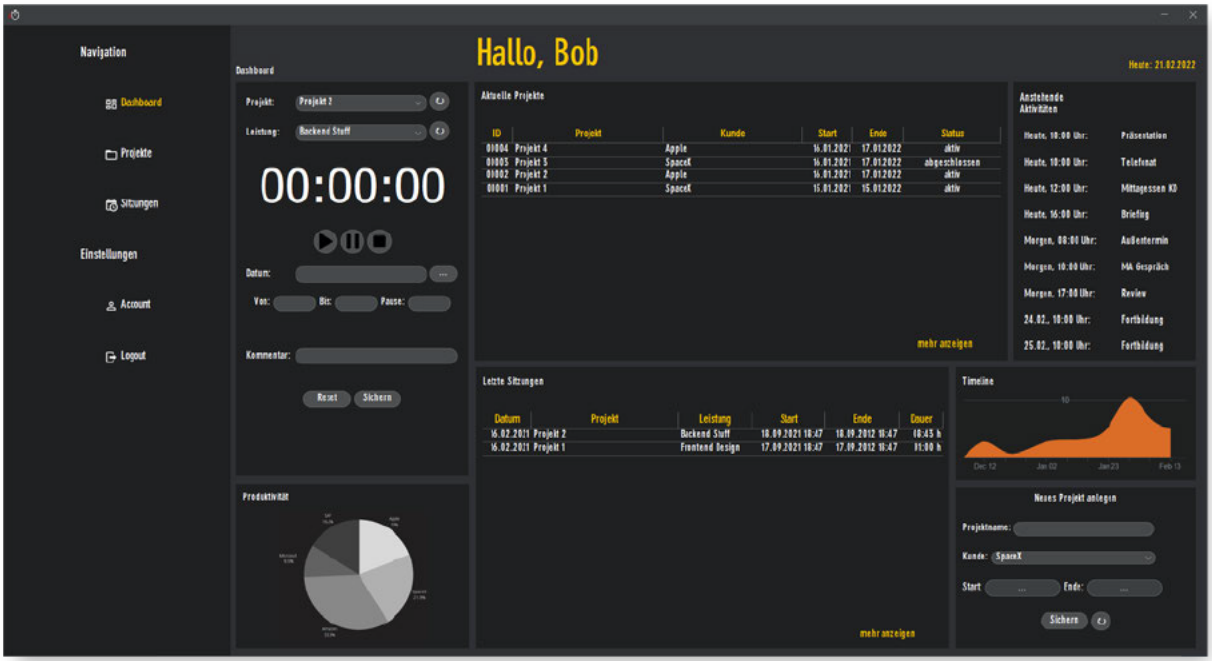


Abbildung 7.2 Screenshot Finalstand

Neben den Learnings zum Thema Software Engineering, die im nachfolgenden Punkt genauer beschrieben sind, hat jeder von uns durch die gesamte Projektarbeit noch sehr viel über die Themen „Drumherum“ lernen können.
Durch die praktische Arbeit mit Java, das Anwenden und Umsetzen von Patterns und Modellen sowie dem selbstständigen Einarbeiten in neue Themen und Herausforderungen haben sich die Programmierfähigkeiten sowie das Verständnis der Konzepte dahinter noch einmal deutlich verbessert.
Waren bspw. Java Swing oder das Ansteuern einer Datenbank aus einer Java-Applikation heraus anfangs noch eine große Blackbox für uns, sind wir nun deutlich sicherer im Umgang damit geworden und haben zumindest eine deutlich bessere Vorstellung davon, was möglich ist und vor allem wie Ideen ggf. umgesetzt werden könnten.

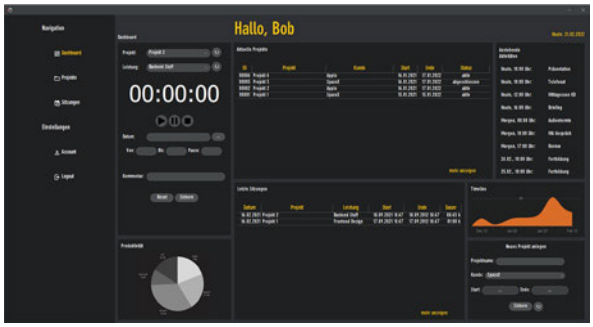
7.2 Learnings

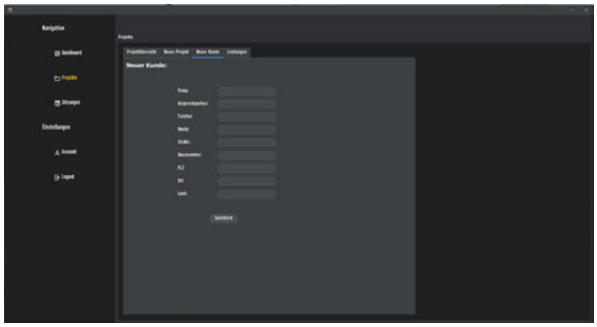
Nicht nur das Endprodukt, sondern auch der Weg dorthin sollte zufriedenstellend sein oder zumindest die Stellschrauben aufdecken, die im nächsten Projekt dabei helfen können, den Prozess und die Software zu verbessern.
Hierzu können wir folgendes festhalten:

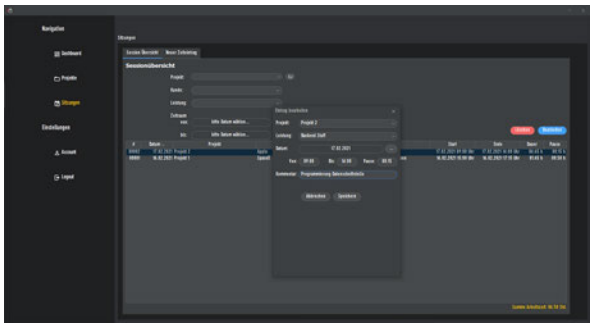
- Mit einer agilen Methode wie Scrum(ban) kann man flexibel auf Änderungen reagieren. Doch auch wenn hierbei nicht gleich am Anfang jeder Bestandteil des Projekts streng durchgeplant wird, ist ein guter Entwurf und eine solide Wissensbasis über das, was man vorhat, entscheidend für den Erfolg
- Zeit ist nicht nur ein Faktor, der darüber entscheidet, wie viele „Arbeitspakete" bearbeitet werden können. Zeitmangel sorgt auch dafür, dass sich bestimmte Vorgehensmodelle ggf. mehr oder weniger eignen. So konnten wir durch wenig Zeit nicht den optimalen Scrumban-Ablauf beibehalten oder immer alles besprechen und reviewen was evtl. nötig gewesen wäre.
- Abstraktion und eine saubere Implementierung mit einer guten Dokumentation helfen nicht nur enorm beim Implementieren neuer Funktionen und dem Vermeiden von Redundanzen, sondern auch beim reibungsloseren Bereinigen von Fehlern
- Code-Reviews sind zwar sehr zeitintensiv, aber für eine gute und gut wartbare Implementierung unverzichtbar. Statt bei Zeitmangel gänzlich oder fast darauf zu verzichten, sollte eine gute Balance zwischen hohem Zusatzaufwand und Verzicht gefunden werden.
- Mehr Commits bei der Versionsverwaltung sind durchaus erwünscht. Sie unterstützen die Versionierung durch eine bessere Übersicht und eine klarere Trennung von unterschiedlichen Änderungen, machen Reverts einfacher. Mit wenig Erfahrung neigt man zu eher seltenen, umfangreichen Commits, dabei sind kleine Commits nichts Schlechtes! Auch muss nicht auf jeden Commit ein Push folgen. Commits erfolgen zunächst lokal und können dann gesammelt in das Repository gepusht werden. Häufige, atomare Commits helfen außerdem beim Formulieren klarer Commit-Nachrichten
- Durch das Arbeiten ohne Build-Werkzeuge wie Maven, haben wir gelernt, welche Arbeit diese Tools einem abnehmen können und warum es in größeren Projekten sinnvoll ist, solche Tools zu nutzen
- Die Projektarbeit insgesamt ist ein großes Learning. Probleme und Herausforderungen, die bei der Implementierung, bei der Versionierung oder in der Teamzusammenarbeit auftauchen sind nicht schlecht, sondern fördern genau solche Learnings, wie wir sie beschrieben haben.

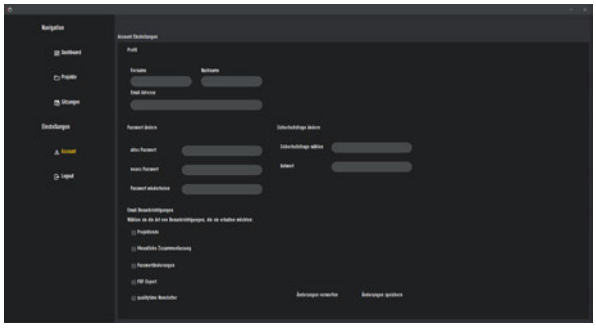
8 ANHANG

8.1 Screenshots der Anwendung









8.2 Zeiterfassung

Nachfolgend findet sich die Tabelle unserer Zeiterfassung mit allen bisher erfassten Tätigkeiten. Dabei ist zu berücksichtigen, dass Zeiten, in denen sich zwar mit der Thematik des *Software Engineering* befasst wurde, aber nicht unmittelbar Arbeiten am Projekt selbst stattfanden, nicht vollständig erfasst wurden. Auch gibt es u. U. Diskrepanzen durch vergessene (und somit nicht eingetragene) oder nicht genau verfolgte Zeiten.

Eine Zusammenfassung der Zeitaufwände nach Person vermeiden wir an dieser Stelle bewusst, da wir keine nach Zeitaufwand unterschiedliche Bewertung wünschen und die Bearbeitung aller Aufgaben zum aktuellen Zeitpunkt als Teamleistung ansehen.

Datum	Person	Von	Bis	Pause	Was	Beschreibung	Dauer	
dd.mm.jjjj	Dropdown	hh:mm	hh:mm	hh:mm	Aufgabenbereich	Kommentar	hh:mm	Std.
04.11 2021	Leander	12:30	13:30	-	Projektmanagement	Dokument für Zeiterfassung und Zeitplan aufgesetzt	01:00	1,00
05.11 2021	gemeinsam	17:00	18:30	-	Anforderungen	Brainstorming: Kernfunktionen & Zusätze, Vorgehensmodell	07:30	7,50
06.11 2021	Kevin	20:00	22:00	-	Anforderungen	User Stories inkl. Akzeptanzkriterien formulieren und Product Backlog erstellen	02:00	2,00
07.11 2021	Leander	16:15	17:30	-	Anforderungen	User-Stories entwickeln auf Grundlage des Brainstormings	01:15	1,25
08.11 2021	Paul	11:00	12:00	-	Anforderungen	User Stories entwickeln	01:00	1,00
12.11 2021	Mo	10:00	11:00	-	Anforderungen	User-Stories entwickeln	01:00	1,00
13.11 2021	gemeinsam	11:00	14:00	-	Anforderungen	User-Stories gemeinsam besprechen und in Backlog einfügen	15:00	15,00
13.11 2021	Sven	9:00	11:00	-	Anforderungen	User Stories entwickeln	02:00	2,00
14.11 2021	Kevin	10:00	11:00	-	Anforderungen	Akzeptanz-Kriterien formulieren	01:00	1,00
17.11 2021	Paul	16:00	17:30	-	Entwurf	Grunddesign Konzept für Gesprächsrunde	01:30	1,50
19.11 2021	Leander	9:15	12:45	0:30	Anforderungen	Akzeptanz-Kriterien & Überlegungen DoD zu User-Stories	03:00	3,00
19.11 2021	Mo	12:00	15:00	-	Anforderungen	Akzeptanz-Kriterien & Entwurf Gestalltung	03:00	3,00
19.11 2021	gemeinsam	17:00	18:45	-	Anforderungen	Akzeptanzkriterien und Wireframes besprechen	08:45	8,75
19.11 2021	Sven	10:00	12:00	-	Anforderungen	Akzeptanzkriterien formulieren	02:00	2,00
25.11 2021	gemeinsam	9:00	12:15	-	Anforderungen	Planning Poker, Sprechstunde, Präsentationsorganisation	16:15	16,25
25.11 2021	Mo	13:00	15:00	-	Anforderungen	UML use case Diagramm erstellen	02:00	2,00
25.11 2021	Mo	15:00	16:00	-	Entwurf	View und Export Seite Entwurf erstellt	01:00	1,00
26.11 2021	Leander	11:30	12:30	-	Anforderungen	Überlegungen/Stichpunkte zur Präsentation der Anforderungsarten	01:00	1,00
27.11 2021	Leander	12:00	12:45	-	Anforderungen	Anforderungsarten: Präsentationsinhalte mit Paul besprechen	00:45	0,75
27.11 2021	Leander	12:00	16:00	-	Anforderungen	Entwurf für Präsentationsslides Anforderungsarten, PPT	04:00	4,00
27.11 2021	Paul	12:00	12:45	-	Anforderungen	Anforderungsarten: Präsentationsinhalte mit Leander besprechen	00:45	0,75
28.11 2021	Sven	17:00	19:00	-	Anforderungen	Präsentation vorbereiten	02:00	2,00
29.11 2021	Kevin	12:30	15:00	-	Anforderungen	Präsentationsinhalte schreiben	02:30	2,50
30.11 2021	gemeinsam	18:00	19:15	-	Anforderungen	Präsentationsinhalte durchsprechen	06:15	6,25

03.12 2021	gemeinsam	17:00	18:30	-	Projektmanagement	Besprechung weiteres Vorgehen, Auswahl Datenbank, UI Framework	07:30	7,50
04.12 2021	Leander	11:30	13:15	-	Projektmanagement	Erster Entwurf Bericht: Projektmanagement & Vorgehensmodell	01:45	1,75
04.12 2021	Leander	17:15	18:45	-	Projektmanagement	Ersten Entwurf Bericht fertigstellen PM & Modell	01:30	1,50
05.12 2021	Paul	11:00	13:00		Sonstige	Swing&Components Recherche	02:00	2,00
06.12 2021	Mo	12:00	14:00	-	Entwurf	UML UseCaseDiagramm erstellen	02:00	2,00
06.12 2021	Leander	14:45	16:15	-	Entwurf	Überlegungen/Recherche Architektur- & Design Pattern sowie Datenbanken in Java	01:30	1,50
07.12 2021	gemeinsam	18:00	20:00	-	Entwurf	Diskussion Softwarearchitektur & Design Pattern, Start Klassendiagramm	10:00	10,00
10.12 2021	Leander	11:00	15:00	-	Entwurf	Überlegungen zu MVC Aufbau, Klassendiagramm, Test in Eclipse, HSQLDB Install	04:00	4,00
11.12 2021	Paul	16:00	20:00		Implementierung & Test	Timer Prototyp Entwurf	04:00	4,00
12.12 2021	Kevin	14:00	15:30	-	Entwurf	Recherche Datenbank, Ausprobieren HSQLDB	01:30	1,50
13.12 2021	gemeinsam	18:30	20:15	-	Projektmanagement	Besprechung Datenbank, Prototypen, Bericht, Arbeitsweise Kanban & Git	08:45	8,75
14.12 2021	Leander	18:00	21:00	-	Projektmanagement	Überarbeitung Bericht gem. Team-Feedback, Quellenangabe	03:00	3,00
15.12 2021	Kevin	10:00	12:00	-	Implementierung & Test	Einarbeitung H2 und Umstellung H2 Datenbank von "server" auf "lokal"	02:00	2,00
15.12 2021	Leander	9:00	21:00	2:00	Entwurf	MVCPrototyp implementieren mit Timer-Funktion, Klassendiagramm grob definieren	10:00	10,00
17.12 2021	Paul	12:00	14:00		Entwurf	präziseres finales Design für Dashboard und einzelne Reiter zur konkreten Implementierung	02:00	2,00
18.12 2021	Kevin	8:30	12:30	-	Entwurf	Einarbeitung DB Grundlagen & ER-Modell, Relationenmodell erstellen	04:00	4,00
18.12 2021	Kevin	17:30	19:30	-	Implementierung & Test	SQL Dateien für Datenbank Initialisierung schreiben	02:00	2,00
18.12 2021	Paul	8:00	12:00		Entwurf	präziseres finales Design für Dashboard und einzelne Reiter zur konkreten Implementierung	04:00	4,00
19.12 2021	Kevin	9:15	13:15	-	Implementierung & Test	Datenbank in Java initialisieren und SQL Dateien überarbeiten	04:00	4,00
01.01 2022	Leander	14:15	18:00	0:30	Implementierung & Test	Refactoring zu Maven File-Structure, Timer ergänzen, Timer-View ergänzen	03:15	3,25
02.01 2022	Leander	12:00	14:00	-	Implementierung & Test	Ergänzungen Timer: Anpassung TimerHourController, View, Model um weitere Buttons/Felder	02:00	2,00
02.01 2022	Leander	15:15	0:15	3:00	Implementierung & Test	Timer: Hour Entries in DB übergeben, Projekte aus DB auslesen und in Dropdown übergeben	06:00	6,00
02.01 2022	Mo	11:00	13:00	-	Implementierung & Test	ProjectView , ProjectModel , Project Controlller angelegt	02:00	2,00
02.01 2022	Paul	12:00	15:00	-	Implementierung & Test	Dashboard Model/View/Controller	03:00	3,00
03.01 2022	Leander	20:00	0:45	2:00	Implementierung & Test	DatabaseController: query-Methode überarbeiten; Timer abfragen überarbeiten	02:45	2,75

03.01 2022	Paul	19:00	22:00	-	Implementierung & Test	Dashboard Model/View/Controller	03:00	3,00
04.01 2022	Leander	17:00	21:15	-	Implementierung & Test	Timer: manuelle Felder automatisch befüllen, Code kommentieren, DB anpassen, Daten in DB schreiben für hour_entries	04:15	4,25
05.01 2022	Kevin	9:15	10:00	-	Entwurf	Beschreibung für Dokumentation: Datenbankaufbau, Normalisierung & Relationenmodell	00:45	0,75
05.01 2022	Kevin	10:00	10:45	-	Implementierung & Test	Datenbank -Relationen: Kurz besprechen zu Funktionen der Integrität beim Löschen von Projekten	00:45	0,75
05.01 2022	Leander	10:00	10:45	-	Implementierung & Test	Datenbank -Relationen: Kurz besprechen zu Funktionen der Integrität beim Löschen von Projekten	00:45	0,75
07.01 2022	Mo	11:30	13:30	-	Implementierung & Test	ProjectView , ProjectModel , Project Controlller verknüpft und überarbeite	02:00	2,00
11.01 2022	Leander	17:45	23:15	-	Implementierung & Test	Bugfixing & Erweiterung Timer-Logik, FlatLaf Look & Feel Install, manuelle Timerfelder aktivieren	05:30	5,50
11.01 2022	Paul	10:00	15:00		Implementierung & Test	Hauptfenster AppMainView und entsprechendes Model und Controller implementiert	05:00	5,00
12.01 2022	Leander	11:00	18:00	2:00	Implementierung & Test	Timer: Validierung Eingabefelder, Warnung bei Fehlern; Arbeit über Nacht wird erkannt; Testing/Bugfixing	05:00	5,00
13.01 2022	Mo	13:00	18:30	1:30	Implementierung & Test	ProjectView Tabellen Sortierung und Filter hinzugefügt	04:00	4,00
14.01 2022	Leander	10:30	14:00	0:30	Entwurf	Vorbereitung Bericht, Klassendiagramm	03:00	3,00
14.01 2022	gemeinsam	14:00	15:30	-	Implementierung & Test	Nächste Sprintplanung; Stand besprechen, Review; Abgaben/Präsentation Entwurf diskutieren	07:30	7,50
14.01 2022	Leander	16:30	19:45	0:45	Entwurf	Bericht zu "Enwurf"	02:30	2,50
17.01 2022	Kevin	11:45	15:15	0:30	Entwurf	Klassendiagramm	03:00	3,00
17.01 2022	Leander	11:00	23:45	3:00	Entwurf	Bericht zu "Enwurf" zuende schreiben, Timer in Dashboard integrieren; Komponentendiagramm, Präsentationsslides	09:45	9,75
17.01 2022	Mo	14:00	15:00	-	Implementierung & Test	project View Filter Reset	01:00	1,00
18.01 2022	gemeinsam	17:00	20:00	-	Entwurf	Klassendiagramm besprechen, aktuelle Stände mergen, Bericht schreiben, Code Review	15:00	15,00
19.01 2022	Leander	9:00	14:00	-	Entwurf	Komponentendiagramm, Präsentation, Bericht finalisieren	05:00	5,00
19.01 2022	Leander	16:00	16:30	-	Implementierung & Test	Fonts installieren Dashboard, externe TimerView entfernen	00:30	0,50
19.01 2022	Mo	15:00	18:00	-	Implementierung & Test	Neues Projekt in Datenbank schreiben	03:00	3,00
19.01 2022	Kevin	21:45	22:45	-	Entwurf	Bericht zu Entwurf überarbeiten	01:00	1,00
21.01 2022	gemeinsam	12:00	13:00	-	Implementierung & Test	Sprintplanung, Code-Review, Merging	05:00	5,00
26.01 2022	Kevin	10:45	18:45	2:00	Implementierung & Test	Merging, Dependencies einbinden, Einarbeitung AssertJ-Swing, GUI Test für TimerView	06:00	6,00
26.01 2022	Kevin	19:30	22:30	-	Implementierung & Test	createTables.sql und insertDummyData.sql	03:00	3,00

						geupdated auf aktuelle Version (v2), GUI Test für LoginView		
27.01 2022	gemeinsam	11:15	11:45	-	Implementierung & Test	Besprechung und merging	02:30	2,50
27.01 2022	Kevin	10:00	11:15	-	Implementierung & Test	GUI Test für ProjectView	01:15	1,25
27.01 2022	Leander	11:45	0:00	3:00	Implementierung & Test	Projekt- & Zeiteintragliste für Dashboard	09:15	9,25
28.01 2022	Kevin	9:15	10:45	-	Implementierung & Test	Unit Tests fixen nach Umstellung auf LayoutManager, GUI Test für ProjectView	01:30	1,50
28.01 2022	gemeinsam	14:00	15:00	-	Implementierung & Test	Sprintplanung, Code-Review, Merging	05:00	5,00
30.01 2022	Leander	23:00	0:00	-	Implementierung & Test	Dashboard: Listen aktualisieren via Listener automatisch	01:00	1,00
31.01 2022	Leander	0:00	1:30	-	Implementierung & Test	DatabaseController: query-Methode ergänzen (Einzelergebnis); Registrierung: Username-Duplicate-Check mit DB	01:30	1,50
31.01 2022	Leander	10:30	13:15	-	Implementierung & Test	Login & Registrierung an Datenbank anschließen und testen	02:45	2,75
03.02 2022	gemeinsam	11:00	11:30	-	Implementierung & Test	Besprechung weiteres Vorgehen, Fragen klären	02:30	2,50
03.02 2022	Kevin	11:30	12:15	-	Implementierung & Test	Unit Test Anpassungen, Trello Tasks schreiben und aktualisieren	00:45	0,75
03.02 2022	Leander	20:15	0:00	-	Implementierung & Test	Merge-Konflikte; Timer: Visual Info Buttons, u_id setzen, DatabaseController Singleton Pattern implementieren	03:45	3,75
04.02 2022	Leander	0:00	0:15	-	Implementierung & Test	User Klasse anpassen, damit überall dieselbe Instanz verfügbar	00:15	0,25
04.02 2022	Leander	10:30	13:00	-	Implementierung & Test	Passwort-Hashing implementiert	02:30	2,50
04.02 2022	gemeinsam	14:00	15:15	-	Implementierung & Test	Besprechen der neuen Stände, weitere Planung	06:15	6,25
05.02 2022	Paul	13:00	15:00		Implementierung & Test	AppMainView/Controller Bugfixing, Project/TimerView/Controller Änderungen	02:00	2,00
06.02 2022	Leander	21:30	22:00	-	Sonstige	Arbeit am Abschlussbericht	00:30	0,50
11.02 2022	Leander	15:30	0:00	3:30	Implementierung & Test	Diverse Anpassungen: Timer mit DatePicker, DatePicker Aussehen, Aufräumen, u_id überall referenzieren etc.	05:00	5,00
11.02 2022	gemeinsam	17:00	18:00	-	Implementierung & Test	Besprechen der neuen Stände, weitere Planung	05:00	5,00
13.02 2022	Kevin	10:00	14:30	-	Implementierung & Test	Umstrukturieren der UnitTests nach Integration in AppMainView	04:30	4,50
14.02 2022	Kevin	9:30	17:00	2:30	Implementierung & Test	"Kunde anlegen" und "Leistung anlegen" im Projekt Modul geschrieben	05:00	5,00
15.02 2022	gemeinsam	19:00	20:45	-	Implementierung & Test	Besprechung Abschlusspräsentation, Bugfixing	08:45	8,75
15.02 2022	Leander	15:00	17:00	-	Implementierung & Test	Dashboard: NewProject Datenbankanbindunge, Dropdown Client-Liste, kleinere Anpassungen	02:00	2,00
16.02 2022	Leander	15:00	20:00	-	Implementierung & Test	"NewProject" Datenbankverknüpfung, Feldvalidierung, DatePicker, Refactoring, Dashboardlisten; LeistungenTimetracker	05:00	5,00
17.02 2022	Kevin	17:00	22:00	1:00	Implementierung & Test	Letzte funktionale Tests schreiben, Recherche für	04:00	4,00

						alternative GUI Testmöglichkeiten		
17.02 2022	Leander	11:00	19:00	1:30	Implementierung & Test	SessionView und NewProject fixes; Filterfunktionen SessionView	06:30	6,50
18.02 2022	Leander	9:45	18:00	2:00	Implementierung & Test	Filterfunktionen SessionView; GUITests	06:15	6,25
18.02 2022	gemeinsam	16:00	16:30	-	Implementierung & Test	Planung letzte Schritte, Main-Merge etc.	02:30	2,50
18.02 2022	Kevin	9:00	16:00	1:00	Implementierung & Test	Fehler abfangen ProjectController & ProjectView, GUI Test Login- & RegistrationView	06:00	6,00
18.02 2022	Kevin	18:00	18:45	-	Implementierung & Test	Fehlerbehandlung Project Controller, Refactoring	00:45	0,75
19.02 2022	Mo	11:00	16:00		Implementierung & Test	Anpassen der Filter Funktion im Project-View	05:00	5,00
19.02 2022	Mo	18:00	20:00		Implementierung & Test	Anpassen des ProjectView Layouts	02:00	2,00
19.02 2022	Kevin	9:00	13:00	-	Implementierung & Test	Code formatieren, unnötige Imports entfernen, javadoc Kommentare, GUI Tests Bugfixing, Integrationstest	04:00	4,00
19.02 2022	Kevin	14:00	22:00	2:30	Implementierung & Test	Integrationstest	05:30	5,50
19.02 2022	Leander	11:00	0:00	2:00	Implementierung & Test	Sessions: Delete & Edit, Datenbankanbindung für neuen Eintrag; Kosmetische Anpassungen App; Lock File	11:00	11,00
20.02 2022	Leander	0:00	2:15	-	Implementierung & Test	Diverse Fixes und Dashboard-Anpassung	02:15	2,25
20.02 2022	Kevin	13:00	15:00	-	Implementierung & Test	Bugfixes verschiedene Testklassen, Javadoc Kommentare, Bericht	02:00	2,00
20.02 2022	gemeinsam	17:00	19:30	-	Implementierung & Test	Main Merge, Build erstellen	12:30	12,50
20.02 2022	Leander	11:00	23:00	3:30	Implementierung & Test	Bug fixing, Bericht, Präsentation, Tests anpassen, Database Dummy Daten einmal schreiben via Methode etc	08:30	8,50
20.02 2022	Mo	16:00	17:00		Implementierung & Test	Kommentare für Methoden schreiben	01:00	1,00
21.02 2022	Leander	6:15	22:30	3:00	Implementierung & Test	Bericht Implementierung, Bug Fixing, Final Build & Release mit Kevin, Präsentationsslides	13:15	13,25
21.02 2022	Paul	15:00	18:00		Sonstige	Bericht/Präsentation Vorbereitung	03:00	3,00
21.02 2022	Kevin	9:00	12:00		Implementierung & Test	Greybox Testing und Bugfixing + Release mit Leander, Blackbox Test Besprechung	03:00	3,00
21.02 2022	gemeinsam	13:00	14:30		Sonstige	Abschlussbericht, Build, Präsentation	07:30	7,50
21.02 2022	gemeinsam	18:00	18:45		Sonstige	Abschlussbericht und Präsentation	03:45	3,75
21.02 2022	Kevin	15:30	18:00		Sonstige	Readme schreiben, letzte Bugfixes, Abschlussbericht	02:30	2,50
Gesamt:								**468,50**

8.3 Quellen

1 H2 Database: http://www.h2database.com/html/main.html

2 FlatLaf: https://www.formdev.com/flatlaf/

3 AssertJ Swing: https://joel-costigliola.github.io/assertj/assertj-swing.html

4 Junit5: https://junit.org/junit5/

5 Dijkstra, Edsger W.: Selected Writings on Computing: A Personal Perspective, Softcover reprint of the hardcover 1st edition 1982, 1982, S. 63

6 Sommerville, Ian: Software Engineering, 10. Aktualisierte Auflage, S. 100 Hallbergmoos, Deutschland: Pearson Deutschland, 2018.

7 Schwaber, Ken/Jeff Sutherland: The Scrum Guide – The Definitive Guide to Scrum: The Rules of the Game, S. 9, in: scrumguides, 10.2020, https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf (abgerufen am 14.12.2021).

8 Reddy, Ajay/Jack Speranza: Scrumban [R]Evolution, The: Getting the Most Out of Agile, Scrum, and Lean Kanban, Kap. 5, 1. Aufl., London, England: Pearson Education, 2016.

9 Attlasian, 2021. Trello [Software]. https://trello.com/ (abgerufen am 14.12.2021)

10 Software Freedom Conservancy, 2021. Git-Scm [Software] https://git-scm.com/ (abgerufen am 14.12.2021)

11 Balzert, Helmut: Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb, 3. Aufl. 2012, S. 23, Heidelberg, Deutschland: Spektrum Akademischer Verlag, 2011.

12 Balzert, Helmut: Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb, 3. Aufl. 2012, S. 24, Heidelberg, Deutschland: Spektrum Akademischer Verlag, 2011.

13 Goll, Joachim: Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java, 2., aktualisierte Aufl. 2014, Wiesbaden, Deutschland: Springer Vieweg, 2014, S. 288.

14 Goll, Joachim: Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java, 2., aktualisierte Aufl. 2014, Wiesbaden, Deutschland: Springer Vieweg, 2014, S. 289.

15 Dijkstra, Edsger W.: Selected Writings on Computing: A Personal Perspective, Softcover reprint of the hardcover 1st edition 1982, 1982, S. 60-61

16 Reade, Chris: Elements of functional programming 1. Auflage 1989, S. 2-3, Wokingham, England: Addison-Wesley Publishers Company

17 Balzert, Helmut: Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb, 3. Aufl. 2012, S. 23, Heidelberg, Deutschland: Spektrum Akademischer Verlag, 2011.

18 Goll, Joachim: Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java, 2., aktualisierte Aufl. 2014, Wiesbaden, Deutschland: Springer Vieweg, 2014, S. 289.

19 Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John; Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, “Behavioral Patterns, Observer”

20 Vgl. Observable (Java SE 16 & JDK 16): in: Oracle | JDK 16 Documentation, o. D., https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Observable.html (abgerufen am 17.01.2022).

21 Vgl. Observer (Java SE 16 & JDK 16): in: Oracle | JDK 16 Documentation, o. D., https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Observer.html (abgerufen am 17.01.2022).

22 Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John; Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, “Behavioral Patterns, Observer”

23 Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John; Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, “Structural Patterns, Composite”

24 Goll, Joachim: Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java, 2., aktualisierte Aufl. 2014, Wiesbaden, Deutschland: Springer Vieweg, 2014, S. 385.

25 Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John; Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, “Behavioral Patterns, Strategy”

26 Code Conventions for the Java TM Programming Language in: Oracle, 1999, https://www.oracle.com/java/technologies/javase/codeconventions-contents.html (abgerufen am 17.01.2022).