ShareShop

Reflexion

2026-01-30

Inhaltsverzeichnis

1. Gruppenreflexion: Grober Überblick . . . 1
2. Top 3 Erfolge . . . 2
    2.1. Erfolg 1: Technische Schulden gezielt reduziert und Weiterentwicklung wieder möglich gemacht . . . 2
    2.2. Erfolg 2: Schnelles und erfolgreiches einarbeiten von zwei neuen Teammitgliedern . . . 3
    2.3. Erfolg 3: Flache Hierarchie und konstruktive Entscheidungsprozesse bei Produktfragen . . . 3
3. Top 3 Misserfolge . . . 5
    3.1. Misserfolg 1: Coding Standards und Tests wurden relativ spät eingeführt und blieben ohne Automatisierung . . . 5
    3.2. Misserfolg 2: Unter-der-Woche-Kommunikation und frühes Einholen von Feedback war schwierig . . . 5
    3.3. Misserfolg 3: Unsauberer Code verursachte Feature-Verschiebungen . . . 6
4. Fazit . . . 8
5. Einzelreflexion . . . 9
    5.1. Maximilian Paul . . . 9
    5.2. Pierre Kißling . . . 11
    5.3. Marlene Fritz . . . 15
    5.4. Levin Heinrich . . . 17
    5.5. Erik Wenke . . . 20
    5.6. Eric Hübel . . . 22
    5.7. Ahmad Alrmih . . . 24

Chapter 1. Gruppenreflexion: Grober Überblick

Im SE2-Projekt **ShareShop** haben wir als Team aus 7 Personen, darunter 2 neue Teammitglieder, gearbeitet. Wir haben in 4 Sprints à 3 Wochen gearbeitet.

Als technische Basis haben wir **Vue** im Frontend sowie **FastAPI**, **SQLAlchemy** und **MySQL** im Backend benutzt. Für Aufgabenverwaltung haben wir **GitHub Projects** benutzt. Die Themen und Diskussionspunkte für Meetings wurden vorab im **Miro** gesammelt.

Zur Qualitätssicherung gab es eine Definition of Done für Product Backlog Items (PBI):

- Alle Sub-Issues (Tasks oder Akzeptanzkriterien) sind abgeschlossen.
- Arbeit ist ins Remote Repository aufgenommen.
- PBI wurde im wöchentlichen Meeting besprochen.

Auf ein Kriterium für Code-Qualität (z.B. Linting) sind wir erst im Laufe des Semesters eingegangen und haben bewusst darauf verzichtet, um den Fokus auf Feature-Umsetzung zu legen.

Prüfungen erfolgten vor allem manuell. Im Laufe des Semesters haben wir aber Coding Standards eingeführt (Styleguidelines im Git-repository) und Tests begonnen (Unit-Tests sowie Tests mit gemockten Daten).

Chapter 2. Top 3 Erfolge

2.1. Erfolg 1: Technische Schulden gezielt reduziert und Weiterentwicklung wieder möglich gemacht

Welche Aufgaben wurden gut gelöst?

Ein wesentlicher Erfolg war, dass wir technische Schulden aus vorherigen Entwicklungsphasen gezielt reduziert haben. Vorher war der Code im Frontend teilweise unübersichtlich, es gab doppelte Logik und große Dateien ohne klare Struktur. Das führte dazu, dass Teammitglieder viel Zeit für die Einarbeitung in den Code benötigten und einzelne Features, insbesondere das Favoriten-Feature, deutlich länger dauerten als erwartet.

Wie wurde dieser Erfolg realisiert?

Wir haben erkannt, dass Feature-Entwicklung ohne eine saubere Codebasis langfristig zu noch mehr Problemen führt. Daher wurden 2-3 Verantwortliche bestimmt, die gezielt aufräumende Arbeiten übernehmen:

- Frontend-Code wurde stärker in Komponenten ausgelagert (mehr Wiederverwendbarkeit).
- Redundanter oder sinnloser Code wurde entfernt (z.B. doppelt definierte CSS-Regeln, wenn global bereits festgelegt).
- Für bestimmte Features wurden notwendige Datenbankanpassungen vorbereitet, damit darauf aufbauende Entwicklung möglich wurde.

Wichtig war dabei, dass die technischen Schulden priorisiert wurden, aber nicht in dem Umfang, alles komplett neu zu bauen. Stattdessen wurden die Teile überarbeitet, die die Weiterentwicklung konkret blockierten.

Was wurde gelernt (Konzepte/Praktiken)?

- Technische Schulden sichtbar machen und gezielt abbauen
- Refactoring zur Verbesserung von Wartbarkeit und Verständlichkeit
- Aufgabenverantwortung für Aufräumarbeiten festlegen

Wirkung:

Nach dieser Aufräumphase war die notwendige Einlesezeit in den Code geringer und die Entwicklung wurde spürbar effizienter. Dadurch konnten wir blockierte bzw. aufgeschobene Features umsetzen: Das Favoriten-Feature wurde anschließend möglich, und auch das Einkaufsarchiv ließ sich einfacher implementieren.

2.2. Erfolg 2: Schnelles und erfolgreiches einarbeiten von zwei neuen Teammitgliedern

Welche Aufgaben wurden gut gelöst?

Zu Semesterbeginn kamen zwei neue Personen ins Team. Wir haben es geschafft, sie schnell einzubinden und "produktiv" zu machen, ohne dass der Prozess nachhaltig ins Stocken geriet.

Wie wurde dieser Erfolg realisiert?

Das Einarbeiten wurde durch mehrere Faktoren unterstützt:

- Einzelne Teammitglieder nahmen sich aktiv Zeit, um Projektaufbau, Code und Workflow erklärt zu bekommen.
- Neue Teammitglieder erhielten zunächst kleinere Aufgaben, wurden aber relativ schnell auch an anspruchsvollere Aufgaben herangeführt, sobald es realistisch erschien.
- Es gab bereits eine Git-Workflow-Anleitung, die viele typische Einstiegshürden (Branching und Umgang mit Änderungen) reduzierte.
- Die neuen Teammitglieder brachten sich zusätzlich selbständig ein, stellten Fragen und lieferten eigene Ideen.

Was wurde gelernt (Konzepte/Praktiken)?

- Wissensmanagement (Doku + direkte Einführung)
- Schrittweises Heranführen über passende Task-Zuschnitte
- Teamkultur als Produktivitätsfaktor

Wirkung:

Bereits nach etwa einer Woche konnten die neuen Teammitglieder produktiv beitragen, was auch dadurch begünstigt wurde, dass das Projekt zu diesem Zeitpunkt noch relativ überschaubar war. Besonders positiv war, dass sie früh eigene Ideen einbgeracht haben und das Team eine offene Einstellung hatte, in der sämtliche Ideen und verbesserungsvorschläge willkommen waren.

2.3. Erfolg 3: Flache Hierarchie und konstruktive Entscheidungsprozesse bei Produktfragen

Welche Aufgaben wurden gut gelöst?

Ein dritter Erfolg war die Teamkultur: Entscheidungen wurden in der Regel demokratisch getroffen und Meinungsverschiedenheiten offen angesprochen. Das half besonders bei anspruchsvolleren Produktfragen, bei denen es mehrere plausible Varianten gab.

Wie wurde dieser Erfolg realisiert?

Obwohl der PO formal das letzte Wort hat, wurden Entscheidungen nicht von oben getroffen. Stattdessen wurden die Einschätzungen der Entwickler:innen aktiv eingeholt, insbesondere zu

Umsetzbarkeit und Aufwand. Da die Entwickler:innen auch gleichzeitig Stakeholder der Zielgruppe waren, wurden Diskussionen nicht nur fachlich sondern auch nutzerorientiert geführt.

Ein konkretes Beispiel ist die Kostenaufteilung. Es gab mehrere mögliche Berechnungsmodelle und Umsetzungsmöglichkeiten. Hier wurde vom PO zusammen mit den Entwickler:innen eine gemeinsame Auswahl getroffen (realistisch für Nutzer:innen und gleichzeitig umsetzbar).

Was wurde gelernt (Konzepte/Praktiken)?

- Gemeinsames Verständnis über Vision und Anforderungen
- Abwägung des Mehrwerts für den Nutzer und technischer Realisierbarkeit
- Konfliktlösung durch direkte Kommunikation und Teamentscheidung

Wirkung:

Diese Arbeitsweise verhinderte längere Konflikte, reduzierte Missverständnisse, führte zu sinnvollen Entscheidungen und es hat Spaß gemacht als Team gemeinsam zu arbeiten. Teammitglieder fühlten sich sicher, Vorschläge einzubringen, ohne Angst vor negativen Reaktionen.

Chapter 3. Top 3 Misserfolge

3.1. Misserfolg 1: Coding Standards und Tests wurden relativ spät eingeführt und blieben ohne Automatisierung

Was ist nicht so gut gelaufen?

Coding Standards (Linter, Styleguidelines) und Tests wurden erst zur Mitte des Semesters eingeführt. Davor entstand inkonsistenter Code, und Prüfungen mussten überwiegend manuell erfolgen. Ohne automatisierte Checks war die Durchsetzung der Standards abhängig von der Disziplin und Aufmerksamkeit des Teams.

Gründe (Wie kam es dazu?):

- Die Relevanz und konkrete Ausgestaltung dieser Themen wurde in der Lehrveranstaltung/Praktika erst relativ spät behandelt, wodurch das Thema auch im Projekt später priorisiert wurde.
- Zu Beginn lag der Fokus stärker auf Feature-Umsetzung; Altlasten im Frontend haben zusätzlich Kapazitäten gefressen.
- Keine Tools, die Standards automatisch durchsetzen.
- Zudem waren die manuellen Tests, welche über die SwaggerUI oder direkt auf der App erfolgt sind, vom zeitlichen Aufwand überschaubar.

Was hätte man tun können, um das Problem zu verhindern?

- Standards und eine minimale Teststrategie bereits im ersten Sprint festlegen (auch ohne den Impuls aus den Vorlesungen).
- Definition of Done um Kriterien erweitern z.B. Linter läuft lokal, mindestens Basistests vorhanden.
- Automatisierte Prüfungen von Anfang an einführen, um menschliche Fehler zu vermeiden.

Was hätten wir tun können, um das Problem zu verhindern?

Als direkte Maßnahme würden wir GitHub Actions einführen, sodass nur Code gemerged wird, der zumindest buildbar/kompilierbar ist und grundlegende Checks besteht (Linting). Damit würden Qualitätsanforderungen verlässlich und unabhängig von manueller Kontrolle eingehalten werden.

3.2. Misserfolg 2: Unter-der-Woche-Kommunikation und frühes Einholen von Feedback war schwierig

Was ist nicht so gut gelaufen?

Die Kommunikation und Abstimmung unter der Woche funktionierte nur eingeschränkt. Durch unterschiedliche Zeitpläne war es schwierig, sich außerhalb des wöchentlichen Meetings

regelmäßig abzustimmen. Zwar funktionierten Abhängigkeiten wie z.B. die Datenbankänderungen sehr gut, aber bei einzelnen Themen litt die Qualität bzw. entstand Nacharbeit, weil Feedback zu spät kam.

Ein konkretes Beispiel war die Gestaltung von Wireframes. Entwicklermeinungen konnten häufig erst im Meeting eingeholt werden. Dadurch war gelegentlich bereits viel Arbeit in Ausarbeitungen geflossen, die anschließend verworfen wurden, weil eine Alternative aus dem Team besser erschien. Später fehlte Zeit, die Wireframes nachzuziehen mit dem Ergebnis, dass Wireframes und App nicht überall deckungsgleich sind.

Gründe:

- Stark asynchrone Zeitfenster. Das gemeinsame Meeting musste sehr spät stattfinden (Sonntag 18:00), um alle überhaupt zusammenzubekommen.
- Austausch unter der Woche war zwar über Whatsapp und Discord möglich, wurde aber nicht immer ausreichend genutzt, um frühes Feedback einzuholen.
- Manche Themen wie zum Beispiel das Design/UX profitieren stark von kurzen Feedbackschleifen, die hier erschwert waren.

Was hätten wir tun können, um das Problem zu verhindern?

- Frühere Feedback-Mechanismen einführen z.B. Kommentarfenster auf Miro/Issues, klare Deadline für Feedback.
- Kurze, optionale Umfragen bzw. schriftliche Entscheidungsrunden(z.B. Abstimmung in Discord).
- Designentscheidungen stärker als Entscheidungspunkte im Miro behandeln (Wireframe Review).

Was würden wir jetzt tun?

Wir würden für Design-/UX-Themen Feedbackprozesse einführen (Miro-Link + Deadline + kurze Entscheidungsdiskussion), damit Meinungen früh eingesammelt werden und nicht erst im Meeting zu spät kommen.

3.3. Misserfolg 3: Unsauberer Code verursachte Feature-Verschiebungen

Was ist nicht so gut gelaufen?

Altlasten im Frontend (doppelter Code, fehlende klare Struktur, zu viel Logik in einzelnen Dateien) führten dazu, dass Features nicht wie geplant umgesetzt werden konnten. Das Favoriten-Feature musste nach hinten verschoben werden, weil zunächst strukturelle Umbauten nötig waren.

Gründe:

- Im Vorsemester wurde viel implementiert, aber nicht sauber strukturiert (Separation of Concerns, Wiederverwendbarkeit).
- Ohne frühe Standards und Tests wuchsen die Inkonsistenzen schnell.

- Technische Schulden waren erst spürbar, als komplexere Features integriert werden sollten.

Was hätten wir tun können, um das Problem zu verhindern?

- Frühzeitig auf Separation of Concerns und Wiederverwendbarkeit achten (Komponenten/Module)
- Technische Schulden kontinuierlich in kleinen Anteilen abbauen, statt erst wenn es ernsthafte Probleme gibt.
- Definition of Done und Styleguidelines früher ernster nehmen.

Was würden wir jetzt tun?

Wir würden bereits zu Beginn des Projekts stärker auf eine klare Struktur achten, also mehr kleinere Komponenten, doppelte Logik unterbinden und regelmäßig Refactoring-Zeit einplanen. Außerdem würden wir die im Semester erarbeiteten Styleguidelines konsequent von Anfang an anwenden.

Chapter 4. Fazit

Die Gruppenarbeit hat uns deutlich gezeigt, dass Entwicklung nicht nur von der Umsetzung der Features abhängt, sondern auch stark von Prozess- und Qualitätsentscheidungen. Besonders erfolgreich waren wir beim gezieltem Abbau technischer Schulden, bei einer klaren Einführung von neuen Gruppenmitgliedern oder bei offenen Entscheidungsprozessen. Herausforderungen und Probleme hatten wir vor allem dort, wo bei uns Struktur und Automatisierung zu spät umgesetzt wurde (Stichwort Code-Standards/Tests) und wo Kommunikation außerhalb des einzigen wöchentlichen Meetings nur schwer möglich war.

Für zukünftige Projekte würden wir frühe Qualitäts- und Kommunikationsmechanismen stärker priorisieren (GitHub-Actions, eine strengere Definition of Done, Feedback asynchron einholen), um Komplexität über mehrere Sprints besser managebar zu halten. Insgesamt sind wir aber äußerst zufrieden mit dem Verlauf des Projekts und den erzielten Ergebnissen und freuen uns, die gewonnenen Erkenntnisse in zukünftigen Projekten anwenden zu können.

Chapter 5. Einzelreflexion

5.1. Maximilian Paul

5.1.1. Ausgangssituation: Welche Aufgaben habe ich im Projekt bearbeitet?

Im Verlauf des Moduls Software Engineering habe ich zwei unterschiedliche Rollen eingenommen, was mir einen umfassenden Einblick in den gesamten Entwicklungsprozess ermöglicht hat. Im ersten Semester agierte ich als Product Owner (PO) und war damit maßgeblich für die inhaltliche Ausrichtung unseres Projekts verantwortlich. In dieser Phase habe ich die Vision für „ShareShop“ entwickelt und damit den Grundstein für alle weiteren Arbeiten gelegt. Ein wesentlicher Teil meiner Arbeit bestand darin, die Anforderungen in Form von User Stories auf unserem Miro Board zu definieren. Dabei habe ich die wichtigsten Grundfeatures konzipiert, darunter die Prozesse für die Anmeldung, das Anlegen von Listen, die Mitgliederverwaltung sowie das Hinzufügen von Produkten und den eigentlichen Einkauf bis hin zum Archiv. Bei der Übertragung dieser Stories in das GitHub Backlog erhielt ich Unterstützung von Erik Wenke, weshalb im System nicht alle Einträge direkt unter meinem Namen laufen, obwohl die inhaltliche Ausarbeitung von mir stammte.

Mit Beginn des zweiten Semesters haben Pierre und ich die Rollen getauscht, sodass ich fortan die Funktion des Scrum Masters übernahm. In dieser Rolle lag mein Fokus weniger auf dem „Was“, sondern vielmehr auf dem „Wie“. Ich sorgte dafür, dass die Scrum Regeln eingehalten wurden und die Zusammenarbeit im Team reibungslos funktionierte. Ein wichtiger technischer Beitrag war die Einführung von Branching-Guidelines. Da wir im ersten Semester eher intuitiv gearbeitet hatten und neue Teammitglieder dazukamen, hielt ich es für wichtig, hier klare Regeln auf Basis von Best Practices zu etablieren, um einen einheitlichen Arbeitsstand zu garantieren. Zudem befasste ich mich mit den Styleguidelines, wobei ich einen Vorschlag von Eric Hübel aufgriff und diesen nach eigener Recherche für unser Team finalisierte. Außerdem habe ich auf dem Miro Board auch Regeln für das Code Review und auch für das Refactorn dokumentiert. Ein weiterer Schwerpunkt war der Fachaustausch zum Thema Projektmanagement, den wir gemeinsam im Team vorbereitet haben und den ich schließlich mit Pierres Unterstützung präsentierte.

5.1.2. Wie bin ich diese Herausforderungen angegangen?

Rollenverständnis: Der Wechsel vom Entscheider zum Begleiter

Eine der größten persönlichen Herausforderungen war der Umstieg von der entscheidenden, inhaltlich getriebenen PO-Rolle hin zu einer eher begleitenden Rolle als Scrum Master. Es fiel mir anfangs nicht leicht, mich voll und ganz auf die neue Rolle einzulassen und bei inhaltlichen Diskussionen lediglich meine Meinung zu äußern, die finale Entscheidung jedoch konsequent dem neuen PO zu überlassen. Dieser Prozess erforderte ein hohes Maß an Selbstdisziplin und Vertrauen in die Vision meines Nachfolgers.

Die Balance zwischen Methodik und Teamdynamik

Gleichzeitig stand ich vor der Aufgabe, die Balance zwischen der strikten Einhaltung von Scrum-Prozessen und unserer gewohnt entspannten Teamatmosphäre zu halten. Ich wollte vermeiden,

dass unsere Meetings zu einem reinen Abarbeiten von Checklisten verkommen. Dabei war es oft schwierig zu entscheiden, wann ich eingreifen sollte, wenn wir thematisch abschweiften. Gelegentlich wurden Themen vorgezogen, die eigentlich an anderer Stelle besprochen werden mussten. Rückblickend betrachtet war ich in diesen Momenten vielleicht manchmal zu entspannt, um den kreativen Fluss nicht zu stören, auch wenn dies die Prozessdisziplin kurzzeitig schwächte.

Standardisierung und Wissenstransfer

Um die Qualität im Team sicherzustellen, habe ich mich intensiv in Best Practices eingelesen und diese auf unsere spezifischen Bedürfnisse zugeschnitten. Besonders bei den Branching- und Styleguidelines war es mir wichtig, nicht einfach Regeln vorzugeben, wie sie laut Scrum Guide(oder anderen Quellen) empfohlen werden, sondern diese auf das Team zugeschnitten einzuführen. Auch beim Fachaustausch war unser Ansatz, dass die Vorbereitung eine gemeinsame Teamleistung sein sollte. Dies förderte nicht nur das Verständnis der Inhalte bei allen Beteiligten, sondern stärkte auch den Zusammenhalt, da die Verantwortung für den Wissenserwerb auf mehreren Schultern lastete.

5.1.3. Bewertung: Hat meine Lösung funktioniert? Warum (nicht)? Was habe ich gelernt?

Im Rückblick bin ich stolz, wie wir als Team zusammen gearbeitet haben. Es ist uns gelungen, eine gute Atmosphäre beizubehalten, in der konstruktiv gearbeitet wurde und Konflikte schnell und diplomatisch aus der Welt geschafft wurden. Die von mir eingeführten Guidelines haben dazu beigetragen, dass trotz wechselnder Rollen und neuer Mitglieder eine klare Struktur herrschte.

Meine wichtigste Erkenntnis aus diesem Jahr ist, dass die strukturelle Führung eines Projekts mindestens genauso fordernd ist wie die inhaltliche Gestaltung. Ich habe mich kommunikativ spürbar weiterentwickelt: Einerseits kann ich meine Gedanken heute präziser und für andere verständlicher formulieren, andererseits bin ich in meiner Ausdrucksweise deutlich diplomatischer geworden. Ich habe gelernt, dass man als Scrum Master oft im Hintergrund die Fäden ziehen muss, damit das Team im Vordergrund glänzen kann.

5.1.4. Belege

User Stories auf dem Miro-Board: https://miro.com/app/board/uXjVIEJUxFI=/

Dokumentation der Branching- und Styleguidelines im Repository: docs/project_management/Branching.adoc und docs/project_management/Styleguidelines.adoc

5.2. Pierre Kißling

5.2.1. Ausgangssituation: Welche Aufgaben habe ich im Projekt bearbeitet?

Als PO war meine zentrale Aufgabe, die Vision festzulegen und nicht aus den Augen zu verlieren: Was soll ShareShop lösen, welche Features bringen den WG-Mitgliedern den größten Nutzen oder sorgen dafür, dass WG-Bewohner die App mehr benutzen, wie schneide ich Anforderungen so zu, dass sie für das Team umsetzbar sind. Dazu gehört insbesondere:

- **Die Analyse:**
  - Problem- und Zielgruppenverständnis verbessern (z.B. WG-Kontext, gemeinsames Einkaufen, Kostenaufteilung).
  - Anforderungen als User Stories / Backlog-Items formulieren und priorisieren.
  - Feature-Alternativen abwägen (z.B. unterschiedliche Berechnungsmodelle für Kostenaufteilung).
- **Der Entwurf (Produktseitig):**
  - Produktentscheidungen treffen bzw. vorbereiten: nutzerflow, Screens/Verhalten, Definition of Done.
  - Abhängigkeiten erkennen (z.B. wenn ein Feature Datenbankänderungen erfordert).
- **Das Projektmanagement:**
  - Sprintziele und Umfang festlegen, Arbeitspakete für das Team aufteilbar machen.
  - Backlog pflegen, Fortschritt über Iterationen sichtbar machen.
  - Umgang mit technischen Schulden.

5.2.2. Wie bin ich diese Herausforderungen angegangen?

Mein größter Lernmoment war die Erkenntnis, dass ich als PO nicht dauerhaft überblicken kann, was jede Person im Team im Detail umsetzt oder wie jedes Software-Teil intern funktioniert. Das war zu Beginn leichter (aufgrund des geringeren Umfangs und geringerer Komplexität), wurde aber im Semesterverlauf realistisch unmöglich. Ich habe daraus zwei Dinge gelernt:

1. **Vertrauen in den Rest meines Teams:** Technische Details müss ich als Po nicht wissen, solange Abhängigkeiten und Risiken für mich immer noch klar sind. Ich muss darauf vertrauen, dass meine Entwickler ihre Aufgaben sauber erledigen.
2. **Fokus auf Schnittstellen statt auf den ganzen Code:** Ich muss nicht wissen, wie etwas implementiert wird, aber ich muss wissen, wenn ein Feature von bestimmten Schnittstellen (z.B. Datenbankänderungen) betroffen ist.

Mein Fokus lag daher vor allem mehr darauf, das Produktziel im Auge zu behalten, User-stories und Akzeptanz-kriterien zu formulieren und einer sauberen Priorisierung.

Ein zweiter Lernmoment war, wie kreativ und zeitintensiv PO-Arbeit ist. Ein Beispiel ist die Kostenaufteilung: Es gibt mehrere Berechnungsmodelle und UX-Varianten, aber bevor man priorisieren kann, muss man sich diese Möglichkeiten überhaupt erst erschließen, vergleichen und

gegeneinander abwägen. Das kostet sehr viel Zeit, weil es keine "richtige" Antwort gibt, sondern viele verschiedene Faktoren, die man schlecht messen kann (Fairness, Verständlichkeit, Aufwand, Fehleranfälligkeit, Komplexität). Meine Vorgehensweise wurde dabei im Verlauf des Semesters zunehmend effizienter und routinierter:

Erst Optionen sammeln (Modelle/Varianten), dann Kriterien definieren (Fairness, Verständlichkeit, Aufwand, Fehleranfälligkeit, Komplexität). Nicht zu lange im Kopf eine Idee groß ausarbeiten, sondern früh mit Feedback arbeiten: andere Personen, die der Zielgruppe entsprechen, direkt fragen, statt nur meine eigene Intuition als Maßstab nehmen.

Das war für mich ein wetvoller Lernaspekt: Produktqualität entsteht nicht nur durch technische rafinesse, sondern auch durch sinnvolle Entscheidungen.

Ein konkreter Misserfolg war meine anfänglich zu optimistische Zeitplanung: Ich habe zu Beginn des Semesters eine zu große Aufgabenmenge als Sprintumfang gewählt. Diese konnten dann nicht vollständig abgeschlossen werden, weil technische Schulden den tatsächlichen Aufwand erhöht haben. Dadurch mussten Aufgaben in den nächsten Sprint verschoben werden.

Wichtig, was ich für mich gelernt habe, war hier, wie ich technische Schulden in der Sprintplanung berücksichtigen kann. Meine Anpassungen waren:

Technische Schulden Ernst nehmen: In der Sprintplanung muss man Schulden/Refactoring als echte Arbeit anerkennen. Das ist keine mal-Nebenbei-Tätigkeit. Realistischere Sprints planen: lieber weniger Items mit höherer Abnahmequalität als viele Items, die nur halb fertig sind. Umsetzbarkeit geht vor Feature-Wunsch: Man muss erst Grundlagen schaffen und kann dann erst große Features implementieren.

Ein dazu passender Erfolg war, dass wir die technischen Schulden innerhalb einer Iteration stark reduzieren konnten und zwar in einem Maß, dass anschließend wieder ein großes neues Feature möglich wurde. Für mich hat das gezeigt: Priorisierung von Qualität ist auf lange Sicht sehr viel Wert.

Ein weiterer Erfolg war die Einführung einer In-Review-Section auf dem Board. Dadurch wurde der Fortschritt innerhalb der Iteration deutlich besser sichtbar. Items sind dadurch nicht mehr direkt in "Done" verschwunden, sondern es gab eine klare Phase für Abnahme & Feedback.

Aus PO-Sicht hat das drei positive Effekte gebracht:

Frühere Feedback-Schleifen: Ich konnte Probleme erkennen, bevor sich zu viel aufgestaut hat. Bessere Kommunikation: Das Team kann klar sehen, was noch auf Prüfung wartet. Realistischere Fortschrittsanzeige: Die Review-Phase wird dadurch als Arbeit sichtbar.

5.2.3. Bewertung: Hat meine Lösung funktioniert? Warum (nicht)? Was habe ich gelernt?

Was gut funktioniert hat: Ich habe ein realistischeres Verständnis für die PO-Rolle entwickelt. Mein Fokus muss mehr auf Produktzielen, Akzeptanzkriterien und der Priorisierung liegen und nicht auf technischen Details. Die technische-Schulden-Iteration hat gezeigt, dass Qualitätsarbeit den Projektfortschritt beschleunigen kann. Mit der "In-Review"-Spalte wurde der Prozess transparenter und die Abnahme strukturierter.

Was nicht sofort funktioniert hat: Sprintplanung war anfangs zu ambitioniert. Technische Schulden wurden unterschätzt. Das führte zu Verzögerungen und erzeugte unnötigen Druck. Scrum-Events waren nur teilweise klar abgegrenzt. Weekly-Meetings fanden regelmäßig klar abgetrennt statt, aber die Zuordnung (Planning/Review/Retro) war nicht immer sauber, was aus meiner Sicht die Prozessdisziplin geschwächt hat. Hier hätte eine klare Dokumentation der Meetings geholfen die Zuordnung transparenter zu gestalten, sodass jedem Klar ist, welche Absprachen zu welchem meeting gehört haben. Das war allerdings auch zu einem Teil der Teamgröße und der damit verbundenen Herausforderung der Zeitplanung geschuldet. Es war wirklich schwierig, alle Teammitglieder zu einem festen Zeitpunkt zu koordinieren und es war daher oft sinnvoll, die Meetings flexibel zu halten und nicht zu starr an Scrum-Regeln zu binden, da wir so viel Zeit verloren hätten.

Was ich über Software Engineering gelernt habe

Rollen und Arbeitsteilung Systeme werden irgendwann einfach zu komplex, um sie zentral verstehen zu können. Gute Prozesse schaffen Vertrauen und Transparenz. Anforderungen müssen verifizierbar werden: "Klingt sinnvoll" reicht oft nicht, sondern es braucht Akzeptanzkriterien und eine klare Abnahme. Priorisierung ist irgendwo auch Risikomanagement: Technische Schulden sind ein echtes nicht zu unterschätzendes Hindernis, denn sie beeinflussen die Produktivität direkt. Nutzerfeedback spart Arbeitszeit: Validierung mit Zielgruppen-Personen ist oft effektiver als langes internes Grübeln. Transparenz erhöht Qualität: Sichtbare Review-Phasen und klare Board-Zustände machen Probleme früher erkennbar.

Qualitätssicherung: was da war und was ich mitnehme

Aus meiner Perspektive wurde Qualität vor allem manuell geprüft. Gleichzeitig haben wir begonnen, Tests zu schreiben, Linter einzuführen und Styleguidelines festzulegen. Mein Lernpunkt daraus:

Qualität entsteht eher schrittweise. Auch wenn Automatisierung nicht sofort vollständig greift, schaffen gemeinsame Standards bereits eine Basis. Für kommende Projekte würde ich zusätzlich befürworten, dass "In-Review" klarer an Kriterien gekoppelt wird (z.B. Linter grün, Tests laufen), damit Reviews sich weniger subjektiv anfühlen.

5.2.4. zusätzlicher Beitrag

Zusätzlich zu den typischen PO-Aufgaben habe ich im Verlauf des Semesters auch die Vorträge/Präsentationen bzw. Fachaustausche für das Projekt vorbereitet. Dazu gehörte, die Inhalte strukturiert und verständlich aufzubereiten. Das war für mich eine wichtige Ergänzung zur PO-Rolle, weil ich dadurch nicht nur die Produktvision innerhalb des Teams vertreten habe, sondern sie auch nach außen verständlich kommunizieren konnte und bei Fachaustauschen Feedback von außen einholen konnte, was unter anderem auch dafür gesorgt hat, dass ich das Product-Backlog, welches wir vom vorherigen Semester übernommen haben, von Grund auf neu aufgebaut habe, um es an die neuen Erkenntnisse und Anforderungen anzupassen.

Ein wichtiger Stabilitätsfaktor im Team war, dass DB-Änderungen zuverlässig und schnell umgesetzt wurden. Unser "Datenbank-Beauftragter" konnte fundamentale Änderungen oft direkt nach dem Meeting erledigen. Dadurch waren Abhängigkeiten früh geklärt, und andere konnten am Folgetag auf einer aktualisierten Datenbank weiterarbeiten.

5.2.5. Belege

- Board mit "In-Review"-Einführung
- Sprintplanung / Sprintziele
- Issues zu technischen-Schulden-Iteration: 1, 2, 3

5.3. Marlene Fritz

Im Projekt Shareshop war ich als Entwicklerin in diesem Semester im Frontend dafür zuständig den Code zu strukturieren, neue Features zu implementieren und gegen Ende die Tests zu reviewen und zu dokumentieren.

5.3.1. Wiederverwendbarkeit vom Code (Issue 295, PR)

Zu Beginn des Semesters habe ich mich mit der Wiederverwendbarkeit und Struktur unseres Codes beschäftigt. Zu dem Zeitpunkt war unsere Komponente List.vue riesig und umfasste die gesamte Logik einer Einkaufliste unserer App. Andere Komponenten, wie die Einkauf.vue enthielt duplizierten Code, da sie ebenfalls eine Liste mit all ihren Artikeln anzeigt. Zusätzlich war die Top Bar in mehreren Komponenten direkt implementiert. Zunächst habe ich analysiert, in welchen Komponenten welcher Code doppelt vorkommt und welche Verantwortlichkeiten sinnvoll ausgelagert werden können. Anschließend habe ich mir ein Konzept überlegt, wie durch neue, klar abgegrenzte Komponenten die Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit des Codes verbessert werden kann. Umgesetzt habe ich dies durch die Einführung zweier neuer Komponenten: eine für den App Header und eine für die Darstellung der Liste. Diese Komponenten können nun durch wenige Zeilen Code an die benötigten Stellen eingebunden werden. Dieses Vorgehen hat sehr gut funktioniert, doch ich habe gesehen, wie schnell Code unübersichtlich werden kann, obwohl man selbst viel dazu beiträgt. Durch die Umstrukturierung hat unser Code deutlich an Qualität gewonnen und es wurde leichter sich darin zurechtzufinden.

5.3.2. Naviagtionsleiste (Issue 278, Issue 277)

Ein weiterer Aufgabenbereich war die Implementierung der Navigation Bar. Dabei konnte ich mich gut an den Wireframes orientieren. Die in den Wireframes vorgesehenen Seiten sind jetzt direkt über die Navigation Bar erreichbar, sodass der Nutzer sich intuitiver durch die App bewegen kann. Die Leiste wird in jeder Komponente angezeigt. Die technische Umsetzung war grundsätzlich nicht kompliziert, da es sich um einfache Buttons handelt, die das jeweilige Routing auslösen. Herausfordernder war hingegen die korrekte Darstellung der Navigation Bar: die Buttons sollten nebeneinander angezeigt werden, die Ansicht musste auch auf mobilen Endgeräten korrekt funktionieren, und der Seiteninhalt durfte nicht teilweise von der Leiste überdeckt werden. Diese Aufgaben erforderten mehrere Anpassungen am Layout und am Zusammenspiel von Navigation und Seiteninhalt.

5.3.3. Kostenaufteilung (Issue 291, PR)

Diese obigen Aufgaben konnte ich ohne vorgelagerte Abhängigkeiten zu anderen Teammitgliedern bearbeiten. Insbesondere die Auslagerung der Komponenten war jedoch für das gesamte Team von großer Bedeutung, da es die Projektstruktur nachhaltig verändert hat. Diese neue Struktur wurde im weiteren Verlauf des Projekts konsequent beibehalten. Die letzte Aufgabe, bei der ich programmierend zum Projekt beigetragen habe, war die Umsetzung der ersten Variante der Kostenaufteilung. Die Logik wurde vorher im Meeting festgelegt. Für die Umsetzung des Features im Backend war Levin verantwortlich und wir haben gemeinsam die benötigten Funktionen festgelegt und eine grobe Gestaltung im Frontend entworfen (Wireframes waren nicht ausgereift). Die Fertigstellung des Backends war eine Voraussetzung für meine Arbeit, stellte jedoch kein Problem dar, da Levin die nötigen Funktionen sehr schnell implementierte und mich informierte

Trotzdem nahm die Umsetzung dieses Features mehr Zeit in Anspruch als ursprünglich geplant. Mir waren nicht alle internen Abläufe in der Einkaufskomponente vollständig klar, weshalb es schwierig war den passenden Punkt für die Erweiterung zu finden. Zudem habe ich zum ersten Mal mit Promise.all, map und filter gearbeitet. Da das Backend korrekt reagiert hat und keine offensichtlichen Fehler auftraten, war die Fehlersuche teilweise anspruchsvoll.

5.3.4. Tests und Dokumentation

Nachdem die Kostenaufteilung implementiert war, wurden keine weiteren Features mehr entwickelt. In der verbleibenden Zeit habe ich mich daher mit dem Testen beschäftigt. Ich habe die bestehenden Tests geprüft, Vorschläge für zusätzliche Tests relevanter Funktionen gemacht und war für die Testdokumentation verantwortlich.

5.4. Levin Heinrich

Ich war im Projekt hauptsächlich für die Implementierung des Backends zuständig. Dabei habe ich mich um die Datenbank gekümmert und die API_Endpunkte, sowie die ORMs für die verschiedenen Modelle erstellt. Zusätzliche habe ich jedoch auch Aufgaben im Frontend übernommen, wie zum Beispiel das Archiv-Feature, aber auch die Mitarbeit an der List- und Einkauf-Komponente.

5.4.1. Datenbankmodell + Relationen (DB Backup)

Ausgangssituation

Zu Beginn ging es darum eine Struktur zu entwickeln, die es ermöglicht die verschiedenen Domänenobjekten und deren Kardinalitäten (die durch die Planung der APP entstanden sind) in einer Datenbank abbilden zu können. Im Laufe des Projekts musste diese Struktur dann angepasst oder um neue Tabellen ergänzt werden. Ich ordne diese Aufgabe in Analyse (Domänenverständnis), Entwurf (Datenmodell/ER-Modell) und Implementierung (Schema/Constraints) ein.

Lösungsweg

Als die Datenbank erstellt bzw. erweiter werden musste begann ich mit einem ER-Diagramm (grobe Skizze) der Tabellen un deren Beziehungen. Sommit konnte ich mir einen Überblick über die geplante Struktur verschaffen und bereits entscheiden wo eventuell eine Zwischentabelle nötig ist. Zudem mussten die Werte, die gespeichert werden sollten, vorher klar im Zweck definiert werden, um eine Aussage über den Datentyp treffen zu können. Nachdem die Struktur stand, habe ich die Tabellen in MySQL erstellt und dort einmal getestet, um zu prüfen ob Foreign Keys und andere Einschränkungen wie NULL/NOT NULL oder UNIQUE richtig funktionierten.

Bewertung und Lernerfahrungen

Rückblickend mkann ich sagen, dass mein Vorgehen grundätzlich gut funktioniert hat. Die grafische Abbildung (auch wenn es nur eine grobe Skizze war) hat besonders am Anfang beim verstehen geholfen. Nahezu am Ende des Projekts habe ich festgestellt dass ich mir jedoch Zeit hätte sparen können, wenn ich die Tabellen direkt in SQLAlchemy erstellt hätte. Da ich mich zu Beginn des Projekts noch nicht so gut mit dem ORM auskannte, habe ich mich für den "sicheren" Weg entschieden. Dies hat zwar funktioniert, aber im Nachhinein gesehen war es unnötig aufwendig.

5.4.2. CRUD-Endpunkte + ORMs | z.B. (PR-Bedarfsvorhersage, PR-Kostenaufteilung)

Ausgangssituation

Bereitstellen der API-Endpunkte für das Frontend. Dabei ging es darum die verschiedenen Funktionalitäten, die im Frontend benötigt werden, über Endpunkte verfügbar zu machen. Ich ordne diese Aufgabe in die Implementierungs- und Testphase ein.

Lösungsweg

Da ich immer vorher die Datenbankstruktur erstellte, hatte ich bereits ein gutes Verständnis über die Entitäten und deren Beziehungen. Das ORM-Mapping in SQL-Alchemy war somit relativ einfach. Anschließend habe ich die Endpunkte für die verschiedenen Modelle erstellt „im ressourcenorientierten REST-Stil (CRUD auf /resource). Dabei stand ich meistens immer im Austausch mit der Person, die das Frontend dazu entwickelte, um sicherzustellen, dass die Endpunkte auch alle benötigten Funktionalitäten abdecken so zu sagen die Abstimmung des API-Contracts (Request/Response-Strukturen). Dabei habe ich die Separation of Concerns abgewogen (z. B. Validierung/Business-Logik serverseitig vs. Darstellung/Interaktion clientseitig). Wenn ich die Endpunkte erstellt hatte, habe ich diese meist mit der Swagger-UI getestet (weil der Großteil der Arbeit im 1. Semester bzw. am Anfang vom 2. Semester stattfand). In den meisten Fällen mussten dann noch ein paar Anpassungen vorgenommen werden.

Bewertung und Lernerfahrungen

Was positiv war, war die Kommunikation mit dem Frontend-Team. Diese Kommunikation war auch nötig, da mann sicherstellen musste, dass beide Seiten (Frontend/Backend) die gleichen Vorstellungen von der Funktionalität haben. Nachteilig war, dass ich mir am Anfang etwas unsicher war, welche Berechnungen/Konvertierungen/Validierungen im Backend oder im Frontend gemacht werde sollten. Das wurde mit der Zeit klarer und die Qualität der API-Endpunkte wurde somit auch besser. Als Ergebnis habe ich den API-Contract stabiler gehalten und serverseitig stärker auf Validierung und konsistentes Error Handling gesetzt.

5.4.3. Archiv-Feature (Frontend) | (Issue-Archiv, Issue-Archiv-Filter, PR-Archiv, PR-Archiv-Filter)

Ausgangssituation

Das Archiv-Feature ermöglicht es dem Nutzer abgeschlossene Einkäufe, mit allesn Produkten die in diesem Einkauf eingekauft wurden, im nachhinein anzusehen. Im Archiv kann man nach den einzelnen Listen filtern sowie nur Einkäufe anzeigen lassen die von mir selbst getätig wurden. Ich ordne diese Aufgabe in die Implementierungs- und Testphase ein.

Lösungsweg

Zuerst habe ich mir überlegt wie das UI ausehen soll und welche Funktionalitäten es bieten soll. Anschließend habe ich die API-Endpunkte erstellt, die benötigt werden. Danach habe ich das Frontend entwickelt. Das Ergebnis habe ich dann im Team vorgestellt und Feedback eingeholt. Da ich selbst als auch das Team mit dem Ergebnis mittelmäßig zufrieden waren haben wir uns im Review mit den anderen Projektgruppen ausgetauscht und ich habe mir Anregungen geholt, wie das Archiv verbessert werden könnte. Dort entstanden dann gute Ideen, die ich dann auch umgesetzt habe. Das Endergebnis wurde dann in der App getestet und vom gesamten Team als gut befunden.

Bewertung und Lernerfahrungen

Bei dieser Aufgabe wurde mir bewusst, wie wichtig Feedback ist, welches von Nutzern der App kommt. Diese kennen nämlich die Architektur hinter der App nicht und fokussieren sich lediglich

auf die Anforderungen bzw. ihre Wünsche. Somit entstand nämlich eine Idee der Filterfunktion welche grundlegend einen anderen Ansatz verfolgte. Gerade bei UX/Usability liefern externe Nutzer früh Hinweise, die im Entwicklerteam sonst untergehen. Daraus resultierte ein Ergebnis, welches ich selbst aus Nutzersicht besser fand, aber aus Entwicklersicht garnicht so betrachtet hatte.

5.5. Erik Wenke

Im Rückblick auf den Verlauf des zweiten Teiles von Software Engineering habe ich hauptsächlich an zwei Features/Anforderungen gearbeitet. Einmal galt es, zusätzlich zur schon vorhandenen manuellen Installation, eine weitere Deployment-Strategie mittels Docker zu entwickeln. Zudem habe ich Zeit dafür investiert, die UI für die Favoriten zu implementieren. Die Favoriten kann man sich dabei als Templates für häufig gekaufte Produkte vorstellen.

5.5.1. Docker

Der zentrale Aufwand bei der Entwicklung der Strategie lag im Aneignen von Wissen. Docker war für mich ein neues Tool. Zudem hatten wir in der bisherigen Entwicklung wenige Gedanken im Hinblick auf Deployment-Strategien investiert. Um einen Einblick in Docker zu bekommen und meinen Lernprozess anzustoßen, hatte sich gut das angebotene Praktikum geeignet. Bei allen weiteren Fragen konnte mir die Docker Dokumentation oder spezifische Anleitungen (z.B. Vue.js) helfen.

Währender der Umsetzung ist uns aufgefallen, dass wir bis jetzt die URL der von uns bereitgestellten API direkt in den Frontendquelltext geschrieben haben.

```
const response = await axios.get(`http://141.56.137.83:8000/einheiten`);
```

Diese müssten bei einer Installation ausgetauscht werden, wenn man die API und Datenbank selber hosten möchte. Das Erstellen einer Axios Instanz mit einer Konfiguration, welche die Basis URL als Umgebungsvariable bekommt, löste das Problem.

Mit der Umgebungsvariable ensteht aber ein weiteres Problem. Das Frontend-Building-Tool Vite benötigt die Umgebungsvariable zur Build-Zeit. Dies erschwert es, ein Docker Image zu erstellen, welchem erst zur Laufzeit die Basis URL übergeben bekommt. Im Moment ist eine unsaubere Lösung implementiert, welche mithilfe von Search-and-Replace die Basis-API-URL einfügt.

Die Beschreibung des Docker Images für die API hat ohne Zwischenfälle funktioniert. Hier wurde nur hinzugefügt, dass die mit der API verbundene Datenbank noch nicht verhandene, aber in der SQLAlchemy MetaData beschriebene, Tabellen direkt erstellen soll. So muss das Datenmodell nicht zusätzlich durch ein SQL-Skript beschrieben werden. Eine Beispiel-Installation der gesamten Anwendung auf einem System ist über einem Docker Compose File realisiert.

Damit hat die Deployment-Strategie mit Docker, die manuelle Installation abgelöst. Im Rückblick freue ich mich darüber, die Möglichkeit bekommen zu haben, mich mehr mit Docker auseinandersetzen zu können. Ich würde behaupten, dass ich einen guten Einblick in Docker bekommen habe, auch wenn mir im Moment noch ein bisschen der Überblick fehlt, um die Gesamtheit und Sinnigkeit von Docker als Tool einzuschätzen. Gerade in der Abtrennung von anderen Möglichkeiten wie z.B. Nix fehlt mir noch ein bisschen die Übersicht, um eine fundierte und disskursfähige Meinung zu entwickeln.

5.5.2. Favoriten

Bevor ich die UI implementierte, waren mir schon zwei Sachen im Kopf. Erstens sollten die

einzelnen Komponenten übersichtliche sein, um zu verhindern, dass wir eine weitere Komponente erstellen, welche 900+ Zeilen Quelltext beinhaltete (siehe List.vue). Zudem war ein weiteres Ziel, bei der Erstellung des UI mehrere wieder verwendbare Komponenten wie zum Beispiel Input Felder zu erstellen.

Die Herausforderung war, dass je kleinteiliger ich die Komponenten voneinander trenne, desto mehr Gedanken muss ich mir über die Kommunikation dieser miteinander machen. Möchte ich zum Beispiel ein allgemeines SingleChoice-Feld erstellen, dann muss sein interner Zustand abgekoppelt von dem der Elternkomponente sein. So mussen Funktionalitäten wie Auswahlmöglichkeiten und Standardwert übergeben und deren Veränderung kommuniziert werden.

Eine Lösung war die Favoriten über globales Prop-Drilling, allen Kind Komponenten der App-Komponente als reaktive Variable zur Verfügung zu stellen.

```
// App.vue
const favorites = ref([]);
provide("favorites", favorites);

// Favorites.vue
inject("favorites");
```

Somit können alle Komponenten die Favoriten bearbeiten und teilen einen gemeinsamen Zustand der Favoriten.

Neben den von mir entwickelten Input Feldern, könnte ich auch auf bereits implementierte Komponenten zurückgreifen. Die Komponente, welche die Produkte in den Einkauflisten als Karte darstellt, konnte ich auch hier nutzen, da auch die Favoriten als Gitter dargestellt werden sollten.

Abschießend sehe ich auf die Implementierung des UI eher kritisch. Die von mir entwickelten allgemeinen Komponenten (NumInput, TextInput, SelectObjectArray) wurden nicht nochmal an anderer Stelle eingebunden. Der Grund dafür ist einfach. Es gibt Component-Libraries. Diese hatten wir nicht von Anfang an bei der Entwicklung unseres UI bedacht. Ein klassiches Ziel für ein zukünftiges umfangreiches Refactoring. Generell würde ich bei einer Überarbeitung des gesamten Projektes gerne weitere Frameworks und Tools einbinden (z.B. Nuxt oder Tailwind CSS), um Funktionalitäten aus- und zuverlagern. Auch würde ich mich bei einer Überarbeitung für einen Store (z.B. Pinia) anstatt des Prop-Drillings entscheiden, um den Anwendungszustand zu verwalten.

Diese Verbesserungsmöglichkeiten sind jedoch nicht nur negativ zu sehen. Durch verschiedene andere Module ist der Umfang meines/unseres Wissens dieses Semester stark gestiegen und unterscheidet sich deutlich von dem, was zum Anfang des Semesters vorhanden war. Die Implementierung nur auf HTML, CSS und JavaScript (im Rahmen des Vue Frameworks) aufzubauen, würde ich schlussendlich als gute Erfahrung einschätzen.

5.6. Eric Hübel

Meine Rolle im Team 4B war Entwickler mit Fokus auf Frontend und Design, das Reviewen von Pull Requests und die Entwicklerdokumentation. Also Implementierung, Code Review und Dokumentation. Ich war sehr motiviert, da ich sofort gemerkt habe, dass das Team wirklich Lust auf das Projekt hatte im Gegensatz zu meinem alten Team, welches immer so schnell wie möglich mit allem fertig werden wollte. Außerdem hatte ich persönliches Interesse am Projekt, da ich gerne Einkaufen gehe und das Listenschreiben manchmal nervig ist.

5.6.1. Designinkonsistenzen (Issue 195, PR 198)

Als neues Teammitglied habe ich zuerst eine Analyse vom Frontend der App vorgenommen, wodurch mir viele kleine Probleme auffielen, wie fehlendes CSS auf manchen Seiten, Rechtschreibfehler und eine schlechte mobile Ansicht. Deshalb wollte ich als meine erste Aufgabe diese technischen Schulden aus dem letzten Semester beseitigen. Es gab nicht wirklich eine User Story zu dem Zeitpunkt dafür, also habe ich ein Issue dazu selber erstellt und alle Probleme als Checkliste aufgeschrieben. Insgesamt hatte ich dann das CSS nochmal gerefactort und vor allem doppelten Code entfernt. Zum Beispiel gab es die ProductCard doppelt, da Einkauf.vue einfach eine geänderte Kopie von List.vue war. Ich habe besonders viel die App auf meinem Handy getestet, da es schließlich eine Handyapp werden sollte und es unwahrscheinlich ist, dass man beim Einkaufen einen Laptop mitnimmt.

Durch Refactorn und das Wiederverwenden von Komponenten ist die Code-Qualität deutlich verbessert worden, wodurch das Implementieren zukünftiger Features, wie der Light-/Dark-Mode, deutlich einfacher wurde und man langfristig motivierter war am Frontend weiterzuarbeiten. Außerdem hatte ich erste Erfahrungen mit Pull Requests gemacht und wie man solche erstellt. Ein erster Fallstrick war hier, dass ich vergessen hatte den Base Branch auf den developer-Branch zu stellen, denn standardmäßig nimmt es main.

5.6.2. Dropdown mit Favoriten (Issue 226, 227, 228; PR 271)

Eine schwierigere Aufgabe war ein Dropdown mit Favoriten zu erstellen. Denn um Favoriten im Dropdown anzuzeigen, war es nötig Favoriten für einen Nutzer zu erstellen. Leider hat es 4 Wochen gedauert auf die Favoritenkomponente des anderen Teammitglieds zu warten. Es wäre theoretisch auch möglich die Favoriten über die API zu erstellen, nur ist das umständlich und fehleranfällig. Es hat solange gedauert, da er viele andere Dinge geändert hat, die nicht Teil der Aufgabe waren. (PR 302) Ich hatte mich dann mit ihm 3 Std. im Discord hingesetzt und darüber geredet, denn die anderen Änderungen hatten teilweise leider mehr Bugs mit reingebracht. Da ich keine Regression mergen wollte, hatten wir uns schlussendlich aus Zeitgründen dazu entschlossen eine andere Branching-Strategie zu nehmen, indem wir einen neuen Branch und damit PR erstellten. Dort wurden nur die geänderten Dateien zu den Favoriten hinzugefügt. (PR 311) Dies diente auch zur Risikovermeidung, da wir sonst jede Out-of-Scope-Änderung durchgehen und eventuelle Bugs fixen müssten. Das kostet nicht nur viel Zeit sondern ist auch fehleranfällig, da man bei 25 geänderten Dateien schnell Bugs übersehen kann.

Mir fiel die Entscheidung nicht leicht vorzuschlagen, dass wir alles andere, was er erstellt hatte und nicht Teil der Aufgabe war, verwerfen. Aber ich finde es war die beste Lösung, da ich nicht noch länger auf das Feature warten konnte. Auch wenn es oft gut geht, weitere Änderungen neben der

eigentlichen Aufgabe vorzunehmen, sieht man an diesem Beispiel, dass es unnötige Verzögerungen dadurch geben kann. Dies ist besonders problematisch bei Aufgaben, die auf einer anderen aufbauen oder sonst abhängig sind. Deshalb halte ich es für sinnvoll, dass jede Aufgabe unabhängig ist und alle Anforderungen im Voraus fertiggestellt sind. Eine andere Möglichkeit wäre das Erstellen einer Regel, welche Scope Creep verhindert.

5.6.3. Reviewen von Pull Requests (PRs with Review req.)

Eine weitere kontinuierliche Aufgabe war das Reviewen von Pull Requests von Entwicklern im Frontend. Dabei habe ich immer alle Änderungen manuell getestet und geschaut ob die Akzeptanzkriterien, welche in den User Stories beschrieben waren, erfüllt wurden. Dies war Teil der Qualitätssicherung, die fehlerhafte oder unvollständige Implementierungen frühzeitig erkennen sollte. Ich habe dabei oft nicht die Github Features zum Kommentieren genutzt, da die meisten nicht so oft ins Github reinschauen und deshalb schrieb ich die zuständigen Entwickler direkt über WhatsApp an. Dies hat eine schnellere Kommunikation ermöglicht, da man hier auch einfacher Screenshots teilen konnte und damit Probleme schneller zeigen konnte. Rückblickend finde ich aber, dass dadurch es zwar kurzfristig effizient war, aber man langfristig nicht mehr weiß worüber man sich unterhalten hatte. Denn wenn man über Github kommuniziert, sind Entscheidungen, Fragen und Probleme auch Jahre später für alle noch sichtbar.

Zudem habe ich bemerkt, dass ich mir viel Zeit hätte sparen können, indem wir automatische Tests in Form von Github Actions eingeführt hätten, anstatt immer alles manuell durchzugehen. So wären schon viele kleine Sachen wie z.B. eine Code-Formatierung, die nicht den Styleguidelines entspricht, schon vor dem Review erkannt wurden und ein Merge wäre überhaupt nicht möglich gewesen. Bisher bestand immer das Risiko, dass trotzdem jemand aus Versehen auf "Commit Merge" klicken könnte und somit fehlerhaften oder unsauberen Code in den developer-Branch gemerged hätte.

5.6.4. Fazit

Insgesamt bin ich relativ zufrieden mit dem Endergebnis, denn die App funktioniert stabil und sieht gut aus. Trotzdem hätte man aber noch mehr refactorn und sich stärker an die Coding-Styleguidelines halten müssen. Und wir hätten von Anfang an eine UI-Library wie Vuetify nutzen sollen und nicht alles von Hand erstellen, wodurch wir mehr Zeit für andere Dinge, wie Tests oder das Refactorn haben könnten. Aber immerhin habe ich dadurch viel über CSS lernen können, was man beim bloßen Nutzen von solchen Libraries nicht tut.

5.7. Ahmad Alrmih

Im zweiten Teil von **Software Engineering 2** habe ich mich hauptsächlich mit der **Implementierung und Absicherung der Backend-Funktionalitäten** der ShareShop-Anwendung beschäftigt. Mein Schwerpunkt lag dabei auf der **Entwicklung von Unit-Tests**, der **Mitarbeit an API-Endpunkten** sowie der **Implementierung von Validierungslogik**.

5.7.1. Ausgangssituation und Einordnung in den SE-Prozess

Meine Aufgaben lassen sich vor allem den Phasen **Implementierung** und **Test** zuordnen. Konkret habe ich:

- an mehrere **FastAPI-Endpunkten** mitgearbeitet
- **Validierungsfunktionen** für Eingaben entwickelt
- eine **umfassende Suite von Unit-Tests** für die API erstellt

Die getesteten Endpunkte wurden teilweise oder größten teils von anderen Teammitgliedern entworfen oder implementiert, sodass meine Arbeit direkt auf bestehenden Analyse- und Entwurfsentscheidungen aufbaute.

5.7.2. Lösungsweg

Für die Tests habe ich **pytest** in Kombination mit **unittest.mock** verwendet. Ziel war es, die Geschäftslogik der API **isoliert von der Datenbank** zu testen. Dazu habe ich:

- die SQLAlchemy-Session mit `@patch` gemockt
- `MagicMock` genutzt, um verschiedene Datenbank-Szenarien (Erfolg, „nicht gefunden“) zu simulieren
- Hilfsfunktionen erstellt, um Mock-Objekte für Nutzer, Produkte, Listen und weitere Modelle wiederverwendbar zu erzeugen

Die Tests folgen konsequent dem **Arrange–Act–Assert-Prinzip**, was die Lesbarkeit und Wartbarkeit verbessert. Zuvor habe ich Validierungsfunktionen entwickelt, um fehlerhafte Eingaben frühzeitig abzufangen und konsistentes Verhalten der Endpunkte sicherzustellen.

5.7.3. Bewertung und Lernerfahrungen

Meine Lösung hat gut funktioniert: Alle implementierten Tests nun laufen zuverlässig und decken sowohl Erfolgs- als auch Fehlerfälle ab. Dadurch konnten mehrere logische Fehler frühzeitig erkannt werden, ohne die Anwendung manuell testen zu müssen.

Besonders gelernt habe ich:

- wie wichtig **korrektes Mocking** komplexer Datenbank-Abfragen ist
- dass gut geschriebene Tests als **Dokumentation des erwarteten Verhaltens** dienen
- dass Validierung ein zentraler Bestandteil robuster API-Entwicklung ist

Rückblickend sehe ich meinen Beitrag als wertvoll für die **Qualitätssicherung** und **Wartbarkeit** des Projekts. Die enge Verzahnung meiner Arbeit mit den Implementierungen anderer Teammitglieder hat mir zudem geholfen, den gesamten Entwicklungsprozess besser zu verstehen.