Architektur-Refactoring der Produktionsplanungs-Software EVOPRO

Patrick Kehrli

Masterarbeit

MAS in Software Engineering

2023 – 2025

Ostschweizer Fachhochschule (OST)

Referent: Stefan Kapferer

Co-Referent: Thomas Memmel

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit im MAS Software Engineering mit dem Titel «Architektur-Refactoring der Produktionsplanungs-Software EVOPRO» selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht haben. Weiterhin erkläre ich, dass ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise verwendet haben und in dieser Arbeit keine Adressen, Telefonnummern und andere persönliche Daten von Personen, die nicht zum Kernteam gehören, publiziere.

Ich erkläre zudem, dass ich für die Erstellung dieser Arbeit den KI-gestützten Schreibassistenten ChatGPT (GPT-5, OpenAI) als Hilfsmittel zur sprachlichen Überarbeitung, Strukturierung und Formulierungshilfe eingesetzt habe. Die inhaltliche Ausarbeitung, Analyse und Bewertung der Ergebnisse stammen vollständig von mir.

Darüber hinaus erkläre ich, dass ich für das Refactoring, insbesondere für die Unterstützung bei der Code-Erstellung und Überarbeitung, die KI-gestützten Werkzeuge Junie und ChatGPT verwendet habe.

Ort, Datum Weinfelden, 11.09.2025

Name, Unterschrift: Patrick Kehrli,

Architektur-Refactoring der Produktionsplanungs-Software EVOPRO

Diese Arbeit untersucht die Neugestaltung der über Jahre gewachsenen Produktionsplanungssoftware EVOPRO mit dem Ziel, Wartbarkeit, Erweiterbarkeit und Testbarkeit zu verbessern. Ausgangslage war ein monolithisches System mit unklarer Trennung der Verantwortlichkeiten, hoher Kopplung zwischen Fachbereichen und geringer Testabdeckung. Dies führte zu Instabilitäten nach Releases und langen Umsetzungszeiten bei neuen Features.

Im Projekt wurden die Architektur und die geforderten Qualitätseigenschaften analysiert. Auf Basis etablierter Architekturmuster – insbesondere Modularen Monolithen und Clean Architecture – entstanden modulare Strukturen mit klaren Verantwortlichkeiten und Schnittstellen. Fachlogik wurde in Use Cases gebündelt und die Architektur durch automatisierte ArchUnit-Tests abgesichert. Zudem entstand ein schrittweiser Plan zur Umsetzung des Refactorings.

Die prototypische Implementierung ausgewählter Module zeigt, wie die Architektur schrittweise modernisiert werden kann. Durch die Trennung von Fachlogik und Infrastruktur, Use-Case-zentriertes Design und automatisierte Tests konnten Abhängigkeiten reduziert und die Testbarkeit gesteigert werden. Das Vorgehen ist auf weitere Module übertragbar und bildet die Grundlage einer nachhaltigen Modernisierung von EVOPRO.

Verfasser:	Patrick Kehrli
Referent:	Stefan Kapferer
Co-Referent:	Thomas Memmel
Veröffentlichung (Jahr):	15.09.2025
Zitation:	Patrick Kehrli, 2025, Architektur-Refactoring der Produktionsplanungs-Software EVOPRO, OST-Ostschweizer Fachhochschule: Masterarbeit MAS Software Engineering
Schlagwörter:	Refactoring, Softwarearchitektur, Monolith, PPS

Management Summary

Ausgangslage
Die Produktionsplanungssoftware EVOPRO wurde über mehrere Jahre kontinuierlich erweitert und stark an individuelle Kundenanforderungen angepasst. Fehlende Architekturleitlinien führten zu enger Kopplung, redundanten Strukturen und geringer Testabdeckung. Dies erschwerte Wartung, Erweiterung und Betrieb.

Relevanz des Themas
In industriellen Produktionsbetrieben ist eine stabile, wartbare und erweiterbare Softwarearchitektur entscheidend für zuverlässige Abläufe. Das Refactoring von EVOPRO dient als Beispiel, wie ein bestehendes, komplexes System strukturell erneuert und langfristig zukunftsfähig gemacht werden kann.

Einsatzumfeld und Ziel
EVOPRO wird von Produktionsunternehmen zur automatisierten Planung von Aufträgen und Ressourcen eingesetzt. Ziel der Arbeit war die Überführung der bestehenden Architektur in eine modulare, entkoppelte Struktur, die Erweiterbarkeit, Testbarkeit und Zuverlässigkeit erhöht.

Zentrale Fragestellung
Wie kann eine historisch gewachsene, stark gekoppelte Anwendung in eine modulare Architektur überführt werden, die sowohl den betrieblichen Anforderungen der Kunden als auch den Entwicklungszielen des Herstellers entspricht?

Vorgehen
Die Arbeit umfasste eine Analyse des Ist-Zustands, die Definition einer Zielarchitektur auf Basis von Clean Architecture und modularer Monolith-Struktur sowie die prototypische Umsetzung ausgewählter Module. Als Methoden kamen Architektur- und NFA-Analysen, das C4-Modell, sowie Metriken aus SonarQube zum Einsatz. Technologien waren u.a. Spring Boot, Vaadin, MongoDB, Docker und Azure DevOps.

Erreichte Ziele und Erkenntnisse
Durch die Modularisierung der Module *Ressourcen*, *Produkt* und *Order* sowie die Einführung von Use Cases konnte eine klare Schichtung erreicht werden. ArchUnit-Tests sichern die Einhaltung der Architekturprinzipien. Die Code-Duplizierung wurde reduziert, Security-Hotspots eliminiert und eine vollständige Testabdeckung der Use Cases erzielt. Damit wurden die definierten Qualitätsziele weitgehend erfüllt. Das Refactoring zeigt, dass auch gewachsene Systeme erfolgreich in eine zukunftsfähige Architektur überführt werden können.

Literaturquellen
Die Arbeit stützt sich auf bewährte Konzepte aus der Fachliteratur, insbesondere *Clean Architecture* (Robert C. Martin), *Domain-Driven Design* (Eric Evans) und *Software Architecture for Developers* (Simon Brown). Ergänzend wurden praxisnahe Quellen wie die SonarQube- und ArchUnit-Dokumentation sowie Herstellerdokumentationen der eingesetzten Technologien berücksichtigt.

Inhaltsverzeichnis

1 Einleitung 6
1.1 Hintergrund & Kontext 6
1.2 Motivation & Problemstellung 6
1.3 Zielsetzung der Arbeit 7
1.4 Aufbau der Arbeit 7
2 Analyse 9
2.1 Stakeholder Analyse 9
2.2 Architekturanalyse 13
2.3 Performance Analyse 20
2.4 Code-Qualitätsanalyse 21
2.5 Fazit der Analyse 23
3 Zieldefinition 24
3.1 Übergeordnetes Projektziel 24
3.2 Architektonisches Ziel 24
3.3 Qualitätsziele 28
3.4 Abgrenzung 29
3.5 Erfolgskontrolle 29
3.6 Fazit der Zieldefinition 29
4 Vorgehensweise 31
4.1 Methodisches Vorgehen 31
4.2 Umsetzungsschritte 32
4.3 Risiken und Massnahmen 33
4.4 Erfolgskontrolle 34
4.5 Fazit Vorgehensweise 34
5 Umsetzung 35
5.1 Modul Ressourcen 35
5.2 Modul Produkt 40
5.3 Modul Auftrag 46
6 Ergebnisse und Fazit 50
6.1 Erreichung der Zielarchitektur 50
6.2 Erfüllung der Qualitätsziele 51
7 Schlusswort 53
8 Literaturverzeichnis 54
9 Abbildungsverzeichnis 54
10 Tabellenverzeichnis 54

1 Einleitung

Diese Masterarbeit beschäftigt sich mit dem Architektur-Refactoring der Produktionsplanungssoftware EVOPRO, um deren Stabilität, Wartbarkeit und Erweiterbarkeit zu verbessern und damit den steigenden Anforderungen von Kunden und Entwicklungsteams gerecht zu werden.

1.1 Hintergrund & Kontext

EVOPRO ist eine webbasierte, Produktionsplanung und -steuerung Software (kurz PPS) respektive ein Advanced Planning and Scheduling System (kurz APS) für kleine und mittlere Unternehmen in der Fertigung. Ihren Ursprung hat die Lösung in einer Simulation mit KI-gestützter Optimierung, um Produktionspläne zu berechnen und zu optimieren. Um diese Kernfunktion für Anwender nutzbar zu machen, wurde schrittweise eine Webanwendung entwickelt, die den Optimierungskern um Benutzeroberflächen, Datenhaltung, Integrationsschnittstellen und weitere Funktionen ergänzt.

Die Software unterstützt Unternehmen dabei, ihre Produktionsplanung zu optimieren, Ressourcen effizient einzusetzen und auf kurzfristige Änderungen im Fertigungsprozess flexibel zu reagieren. Dabei kombiniert EVOPRO die Funktionalität klassischer PPS-Systeme mit modernen Optimierungs- und Analyseverfahren. Technologisch basiert die Anwendung auf einem modernen Stack mit Vaadin[1] für das User Interface (kurz UI), Spring Boot[2] im Backend, Docker[3] für die Containerisierung und Azure DevOps[4] für die Build- und Deployment-Prozesse.

1.2 Motivation & Problemstellung

Die Architektur von EVOPRO ist über mehrere Jahre organisch gewachsen. Neue Funktionen und Kundenanpassungen wurden schrittweise ergänzt, ohne dass eine durchgängige, übergeordnete Architekturstrategie verfolgt wurde. Dies führte zu einer hohen Kopplung zwischen den einzelnen Komponenten und einer unklaren Trennung fachlicher Verantwortlichkeiten. Fachliche Logik verteilte sich teilweise bis in die

---

[1] Vaadin ist ein Java-basiertes Webframework: https://vaadin.com/

[2] Spring Boot ist ein Java Framework, dass die Entwicklung von Applikationen vereinfacht: https://spring.io/

[3] Docker ist eine Container-Plattform, die Anwendungen und ihre Abhängigkeiten isoliert verpackt und so portabel und skalierbar macht: https://www.docker.com/

[4] Azure DevOps ist eine Plattform von Microsoft, die integrierte Tools für Softwareentwicklungsteams bereitstellt: Azure DevOps

Benutzeroberfläche, und zentrale Geschäftsprozesse wurden vorwiegend durch CRUD-orientierte Services (Create, Read, Update, Delete) abgebildet.

Diese gewachsene Struktur brachte mehrere Nachteile mit sich:

- **Erhöhtes Risiko von Seiteneffekten**: Änderungen in einem Bereich wirkten sich häufig auf fachlich nicht verwandte Teile der Anwendung aus.
- **Erschwerte Testbarkeit**: Die fehlende Testabdeckung und die Vermischung von Logik und Infrastruktur erschwerten die gezielte Absicherung von Änderungen.
- **Begrenzte Erweiterbarkeit**: Neue Anforderungen oder kundenspezifische Erweiterungen konnten nur mit hohem Aufwand und Risiko umgesetzt werden.

Die Motivation für das Architektur-Refactoring entstand einerseits aus den gestiegenen Kundenanforderungen, andererseits aus den im MAS-Studium im Bereich Software Engineering gewonnenen Erkenntnissen, wie eine klar strukturierte, modulare und testgetriebene Architektur gestaltet werden kann.

1.3 Zielsetzung der Arbeit

Ziel dieser Masterarbeit ist es, die bestehende Softwarearchitektur von EVOPRO zu analysieren, deren Schwachstellen zu identifizieren und auf dieser Basis ein Architektur-Refactoring durchzuführen. Das Ergebnis soll eine moderne und wartbare Architektur sein, die den langfristigen Betrieb sichert und eine flexible Anpassung an die Bedürfnisse verschiedener Produktionsbetriebe ermöglicht.

Im industriellen Umfeld, in dem EVOPRO eingesetzt wird, sind Zuverlässigkeit und Stabilität zentrale Anforderungen. Daher beinhaltet das Refactoring neben der strukturellen Neuorganisation auch die Erhöhung der Testabdeckung sowie die Verbesserung der allgemeinen Code-Qualität. Die neuen Architekturprinzipien sollen gewährleisten, dass Erweiterungen und Anpassungen künftig stabiler und effizienter umgesetzt werden können.

Die Umsetzung erfolgt exemplarisch an ausgewählten Modulen, um die Übertragbarkeit der Architekturprinzipien auf das Gesamtsystem zu demonstrieren. Die Arbeit orientiert sich dabei an bewährten Konzepten wie dem Modularen Monolithen (Brown, Modular Monoliths, 2018) und der Clean Architecture (Martin, 2018) und kombiniert diese mit praxisorientierten Massnahmen zur Erhöhung der Wartbarkeit und Qualität.

1.4 Aufbau der Arbeit

Im Weiteren ist die Arbeit wie folgt strukturiert.

Kapitel 2 analysiert die bestehende Architektur und identifiziert deren Schwachstellen. Dabei werden sowohl technische als auch strukturelle Defizite betrachtet, die eine Grundlage für die spätere Zieldefinition bilden.

Kapitel 3 definiert die übergeordneten und architektonischen Ziele des Refactorings. Es beschreibt die Leitlinien, Architekturprinzipien und Qualitätsziele, die bei der Umsetzung berücksichtigt werden.

Kapitel 4 erläutert das methodische Vorgehen. Es beschreibt die gewählte Vorgehensweise, die Umsetzungsschritte pro Modul sowie die Massnahmen zur Qualitätssicherung und Erfolgskontrolle.

Kapitel 5 dokumentiert die praktische Umsetzung des Refactorings anhand ausgewählter Module. Für jedes Modul werden Ausgangslage, durchgeführte Schritte, technische Herausforderungen, erzielte Ergebnisse und ein Fazit dargestellt.

Kapitel 6 fasst die wichtigsten Ergebnisse zusammen, bewertet das Erreichen der gesetzten Ziele und gibt einen Ausblick auf mögliche zukünftige Weiterentwicklungen.

2 Analyse

Ziel dieser Analyse ist es, die fachlichen und technischen Grundlagen des Systems strukturiert zu erfassen und auf dieser Basis eine fundierte Bewertung und Weiterentwicklung zu ermöglichen.

Die Analyse orientiert sich an bewährten Methoden aus dem Requirements Engineering, der Softwarearchitektur und der nicht-funktionalen Anforderungsanalyse. Sie dient als Grundlage für die Konzeption einer Zielarchitektur, die den aktuellen und zukünftigen Anforderungen an das System gerecht wird.

Im Verlauf der Analyse werden folgende Aspekte betrachtet:

- Die Erhebung und Strukturierung fachlicher und technischer Anforderungen
- Die Definition und Abgrenzung des Systemkontexts
- Die Modellierung der Domäne zur Verdeutlichung zentraler Begriffe und Zusammenhänge
- Die Bewertung der bestehenden Architektur mit Fokus auf Stärken, Schwächen und Verbesserungspotenzialen
- Die Identifikation und Bewertung nicht-funktionaler Anforderungen (NFA)

Die Ergebnisse dieser Analyse bieten eine belastbare Basis für Architekturentscheidungen, Refaktorierungen oder Erweiterungen des Systems. Darüber hinaus unterstützen sie die Kommunikation zwischen Fachexperten, Entwicklerteams und weiteren Stakeholdern.

2.1 Stakeholder Analyse

Die Stakeholder Analyse dient dazu, alle relevanten Anspruchsgruppen zu identifizieren, ihre Interessen und Ziele zu verstehen sowie ihren Einfluss auf das Projekt einzuschätzen. Sie bildet die Grundlage dafür, Entwicklungsentscheidungen so zu treffen, dass die unterschiedlichen Erwartungen in Einklang gebracht werden.

Im Fokus steht dabei der Kunde, dessen Anforderungen an Stabilität, Performance und Benutzerfreundlichkeit massgeblich für den Projekterfolg sind. Gleichzeitig werden interne Stakeholder berücksichtigt, deren Ziel es ist, durch eine wartbare und erweiterbare Architektur langfristig die Produktqualität zu sichern, den Supportaufwand zu reduzieren und Raum für Innovationen zu schaffen.

2.1.1 Stakeholdergruppen

Produktionsleitung (Kunde)

Die Produktionsleitung ist verantwortlich für die operative Planung und Steuerung der Fertigung. Sie nutzt EVOPRO täglich und in vollem Funktionsumfang, um Produktionspläne zu erstellen, zu optimieren und bei Bedarf kurzfristig anzupassen. Ihre Arbeit ist stark abhängig von kurzen Lade- und Reaktionszeiten, einer stabilen

Lauffähigkeit und einer verlässlichen Terminplanung. Schon geringe Verzögerungen oder Systemausfälle wirken sich unmittelbar auf die Produktionsleistung aus und können zu Terminüberschreitungen, ineffizienter Ressourcennutzung und Kundenunzufriedenheit führen.

In der aktuellen Situation führen komplexe Planungsprozesse teilweise zu langen Ladezeiten, die den Arbeitsfluss verlangsamen und Entscheidungen verzögern. Zusätzlich kommt es nach Software-Updates vereinzelt vor, dass einzelne Funktionen nicht mehr ordnungsgemäss arbeiten und Fehler verursachen. Diese Probleme beeinträchtigen die Zuverlässigkeit der Anwendung, erhöhen den Abstimmungsbedarf zwischen den Beteiligten und können im ungünstigsten Fall zu fehlerhaften Planungen führen.

Produktionsmitarbeiter (Kunde)

Produktionsmitarbeiter sind die Endanwender der in EVOPRO erstellten Arbeitspläne. Sie greifen täglich auf die Anweisungen und Auftragsinformationen zu, die über die Software bereitgestellt werden. Für sie steht eine einfache, klare und konsistente Bedienoberfläche im Vordergrund, um ihre Arbeit ohne unnötige Unterbrechungen durchführen zu können.

Aus Sicht der Produktionsmitarbeiter läuft die Anwendung stabil und zuverlässig. Die Ansichten sind übersichtlich gestaltet, und Systemunterbrechungen treten nur sehr selten auf, sodass der Arbeitsfluss in der Regel nicht beeinträchtigt wird.

Geschäftsleitung (Kunde)

Die Geschäftsleitung auf Kundenseite nutzt EVOPRO in der Regel nicht selbst operativ, ist jedoch auf die von der Software bereitgestellten Kennzahlen angewiesen, um strategische Entscheidungen zu treffen. Hohe Verfügbarkeit, ausreichender Schutz sensibler Unternehmensdaten und eine schnelle Erstellung aussagekräftiger Reports sind dabei zentral.

Derzeit stehen die benötigten Auswertungen und Reports in angemessener Zeit zur Verfügung. Von Seiten der Geschäftsleitung werden gelegentlich kleinere Funktionsanpassungen gewünscht, die jedoch nicht kritisch für den laufenden Betrieb sind.

Geschäftsleitung (Eula Software AG)

Die Geschäftsleitung der Eula Software AG, Herstellerin von EVOPRO, definiert die strategische Ausrichtung des Produkts und priorisiert Entwicklungsressourcen. Ihr Ziel ist es, ein stabiles, wettbewerbsfähiges Produkt anzubieten, das langfristig Kundenzufriedenheit sichert und gleichzeitig wirtschaftlich nachhaltig ist.

Ein stabiler und wartbarer Systemkern reduziert den Supportaufwand, verkürzt Entwicklungszyklen und schafft Raum für Innovationen. Aktuell führen häufige Hotfixes

und ungeplante Wartungsarbeiten zu einem erhöhten Ressourcenverbrauch, was die Umsetzung neuer Produktideen verzögert.

Entwicklungsteam (Eula Software AG)

Das Entwicklungsteam ist für die Wartung, Weiterentwicklung und technische Qualität von EVOPRO verantwortlich. Es verfolgt das Ziel, eine modulare, klar strukturierte Architektur zu schaffen, die eine schnelle, stabile Umsetzung neuer Funktionen ermöglicht. Hohe Testabdeckung, klare Schnittstellen und lose Kopplung zwischen Modulen sind wesentliche Anforderungen, um langfristig Wartbarkeit und Erweiterbarkeit sicherzustellen.

Gegenwärtig leidet die Entwicklung unter einer stark gekoppelten Codebasis, die Änderungen erschwert und das Risiko von unbeabsichtigten Nebenwirkungen erhöht. Die fehlende oder unvollständige Automatisierung von Tests verstärkt dieses Problem. Die Folge sind zeitintensive Fehlerbehebungen, die Ressourcen für Neuentwicklungen blockieren.

Vertrieb (Eula Software AG & Vertriebs-Partner)

Der Vertrieb ist massgeblich für die Markterschliessung und Kundengewinnung verantwortlich. Während der interne Vertrieb der Eula Software AG in engem Austausch mit der Entwicklung steht und auch technische Machbarkeiten berücksichtigt, konzentrieren sich die Vertriebspartner stärker auf die Präsentation von Funktionen und die Vermarktung.

Hauptziele sind eine überzeugende Feature-Pipeline und reibungslose Produktdemos. Technische Stabilität wird dann relevant, wenn Probleme in Kundengesprächen sichtbar werden – etwa durch fehlerhafte Live-Demos oder fehlende Funktionen. Verzögerte Releases können Verkaufschancen mindern und die Glaubwürdigkeit gegenüber Bestandskunden beeinträchtigen.

2.1.2 Ziele der Stakeholdergruppen

In der Tabelle 1 werden die Ziele und aktuellen Herausforderungen nach Stakeholdergruppe beschrieben:

Stakeholder	Anforderungen	Aktuelle Herausforderungen
Produktionsleitung (Kunde)	Kurze Ladezeiten, stabile Lauffähigkeit, zuverlässige Planung	Lange Ladezeiten, Stabilität nach neuen Releases
Produktionsmitarbeiter (Kunde)	Intuitive Bedienung, zuverlässiger Zugriff auf Aufträge	keine
Geschäftsleitung (Kunde)	Verfügbarkeit, Datensicherheit, schnelles Reporting, Anbindung an ERP-Systeme	Flexibel neue Reports erstellen

Entwicklungsteam (Eula)	Wartbarkeit, Testbarkeit, modulare Architektur, einfacher Deployment-Prozess	Enge Kopplung im Code, geringe Testabdeckung, zu viele Hotfix
Geschäftsleitung (Eula)	Schneller Onboarding-Prozess, geringer Supportaufwand, hohe Innovationsfähigkeit	Hoher Ressourcenverbrauch für Hotfixes, schleppende Entwicklung neuer Features
Vertrieb (intern & Partner)	Feature-Vielfalt, störungsfreie Demos	Release-Verzögerungen, Demo-Probleme

Tabelle 1: Ziele nach Stakeholdergruppe

2.1.3 Stakeholder Kategorisierung

Die Einfluss-/Interesse-Matrix in Abbildung 1 zeigt die Positionierung der identifizierten Stakeholdergruppen bezogen auf das geplante Refactoring der EVOPRO Software.

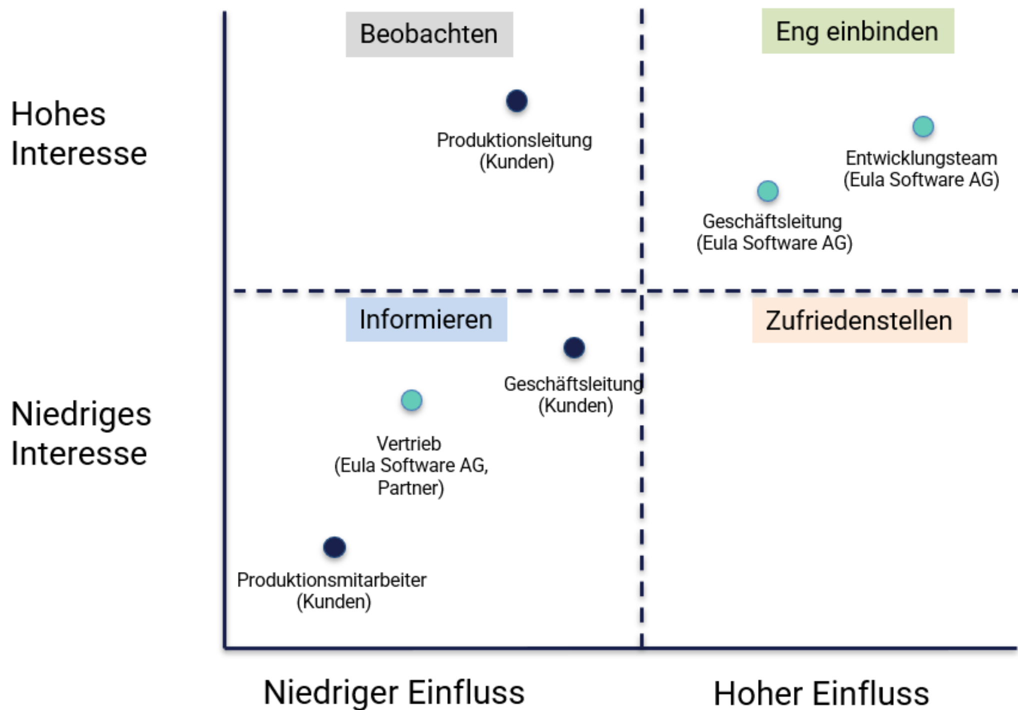

Abbildung 1: Interesse-Einfluss Matrix der Stakeholdergruppen

Auffällig ist, dass die Kunden – insbesondere Produktionsleitung und Geschäftsleitung – trotz ihres hohen Interesses am Projekterfolg mit einem eher geringen Einfluss kategorisiert sind.

Dies liegt nicht daran, dass ihre Bedürfnisse für das Projekt weniger wichtig wären – im Gegenteil: Das Refactoring hat zum Ziel, die von den Kunden geforderte Stabilität, Zuverlässigkeit und Flexibilität zu erreichen. Der geringe Einfluss bezieht sich ausschliesslich auf die operative Umsetzung des Refactorings. Entscheidungen über die konkrete Architektur, die Priorisierung der Massnahmen und die technische Vorgehensweise liegen primär beim Entwicklungsteam und der Geschäftsleitung der

Eula Software AG. Diese Stakeholdergruppen verfügen sowohl über das notwendige Fachwissen als auch über die Entscheidungskompetenz, um die Umsetzung inhaltlich zu steuern.

Damit macht die Matrix deutlich: Die Kunden stehen im Zentrum der Motivation für das Refactoring, üben aber keinen direkten Einfluss auf die technische Ausgestaltung aus. Die Verantwortung liegt bei den internen Stakeholdern, die im Sinne der Kunden die Architektur nachhaltig verbessern.

2.1.4 Fazit Stakeholder Analyse

Die Stakeholder Analyse verdeutlicht, dass insbesondere die Produktionsleitung der Kunden ein hohes Interesse am Refactoring von EVOPRO hat. Für sie sind vor allem Ladezeiten, Stabilität und Erweiterbarkeit entscheidend, um im täglichen Betrieb zuverlässig arbeiten und auf neue Anforderungen reagieren zu können. Aufgrund ihrer operativen Rolle verfügt die Produktionsleitung jedoch nur über geringen Einfluss auf die technische Umsetzung und ist daher auf die Initiative der Eula Software AG angewiesen.

Aus Sicht aller beteiligten Anspruchsgruppen besteht der grösste Handlungsbedarf in der Verbesserung der drei nicht-funktionalen Anforderungen Wartbarkeit, Testbarkeit und Erweiterbarkeit. Es zeigt sich, dass eine spürbare Steigerung der Performance ebenfalls hohe Relevanz besitzt, um Ladezeiten zu reduzieren und Arbeitsprozesse effizienter zu gestalten.

Das Refactoring bietet damit die Gelegenheit, genau diese Kernaspekte gezielt zu adressieren und so nicht nur die Kundenzufriedenheit zu erhöhen, sondern auch den internen Entwicklungs- und Supportaufwand nachhaltig zu reduzieren.

2.2 Architekturanalyse

2.2.1 Systemkontext

EVOPRO ist ein webbasiertes APS (Advanced Planning System), das Produktionsbetriebe bei der termin- und ressourcengerechten Planung unterstützt. Das System wird von unterschiedlichen Rollen im Kundenunternehmen genutzt (v. a. Produktionsleitung und Produktionsmitarbeitende) und steht im Austausch mit externen Diensten für die Authentifizierung sowie mit betrieblichen Drittsystemen (z. B. Enterprise-Resource-Planning, kurz ERP). Die Anwendung läuft in einer Cloud-Umgebung und speichert Daten in einer verwalteten Datenbank.
Abbildung 2 zeigt den Kontext gemäss C4 Modellierung (Brown, Software Architecture for Developers Vol. 2, 2016) auf einen Blick: zentrale Benutzergruppen, angebundene Fremdsysteme und die wichtigsten Kommunikationsbeziehungen.

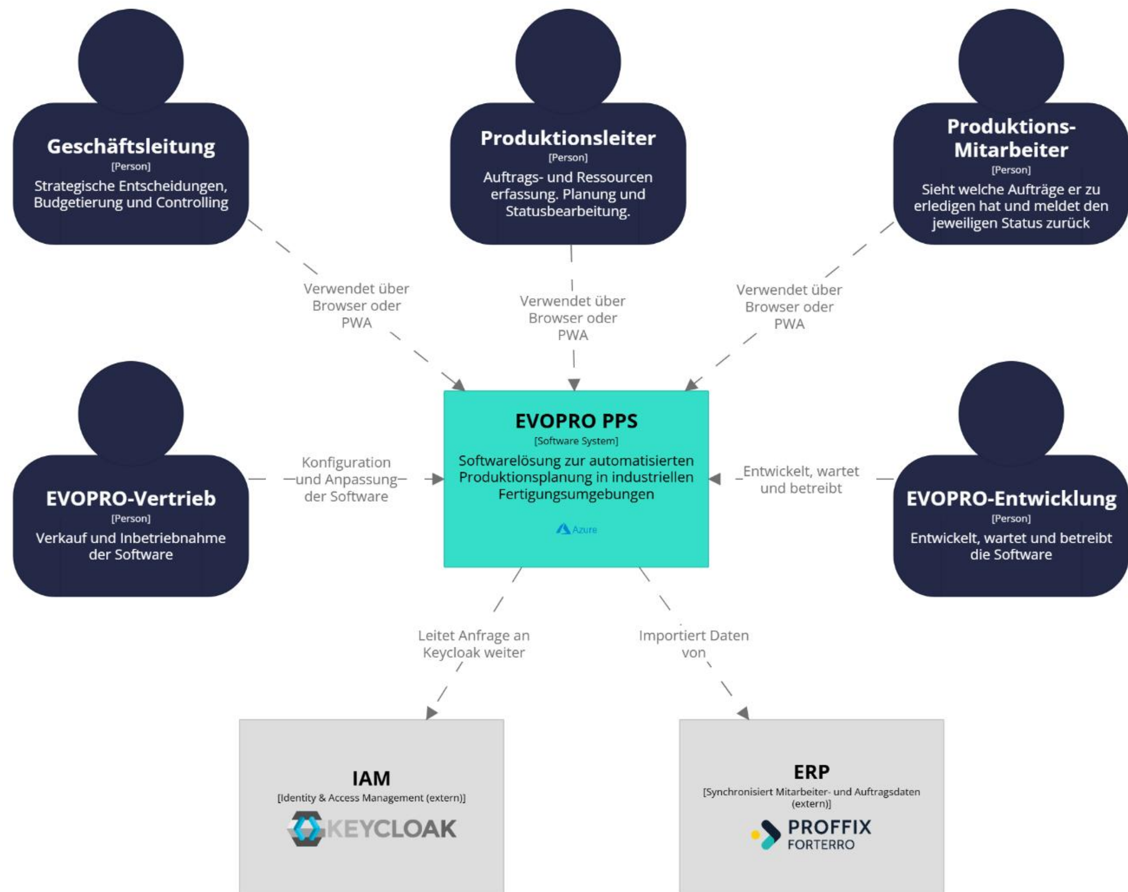

Abbildung 2: EVOPRO Systemkontext

Akteure und externe Systeme (Überblick):

- Benutzergruppen: Produktionsleitung (Planung und Steuerung), Produktionsmitarbeitende (Abarbeitung/Aktualisierung), Geschäftsleitung (Auswertungen/Entscheidungen), EVOPRO-Vertrieb (Demo/Inbetriebnahme), EVOPRO-Entwicklung (Entwicklung/Wartung)
- Externe Services:
  – Identity & Access Management: z. B. Keycloak für Anmeldung und Rollen.
  – ERP-System: Austausch von Stammdaten und Aufträgen (Import/Export via RESTful API über HTTPS).

Zentrale Datenflüsse:

Die Benutzer melden sich über ein externes Identity-&-Access-Management (IAM) an, das für Authentifizierung und rollenbasierte Autorisierung zuständig ist. Nach erfolgreicher Anmeldung greifen sie auf die verschiedenen Funktionen von EVOPRO zu. Für den Austausch betriebsrelevanter Informationen wie Stammdaten oder Aufträge kommuniziert EVOPRO direkt mit dem ERP-System der Kunden über standardisierte Web-Schnittstellen (RESTful API über HTTPS). Dabei werden Daten

bidirektional übertragen, sodass sowohl EVOPRO als auch das ERP jederzeit auf aktuelle Informationen zugreifen können.

2.2.2 Container-Diagramm

Das Containerdiagramm (Brown, Software Architecture for Developers Vol. 2, 2016) in Abbildung 3 zeigt die zentralen Bausteine von EVOPRO und deren Interaktion innerhalb des Systems. Die Architektur ist auf dieser Ebene in klar abgegrenzte Container unterteilt, die jeweils für eine definierte Aufgabe zuständig sind und über wohldefinierte Schnittstellen miteinander kommunizieren.

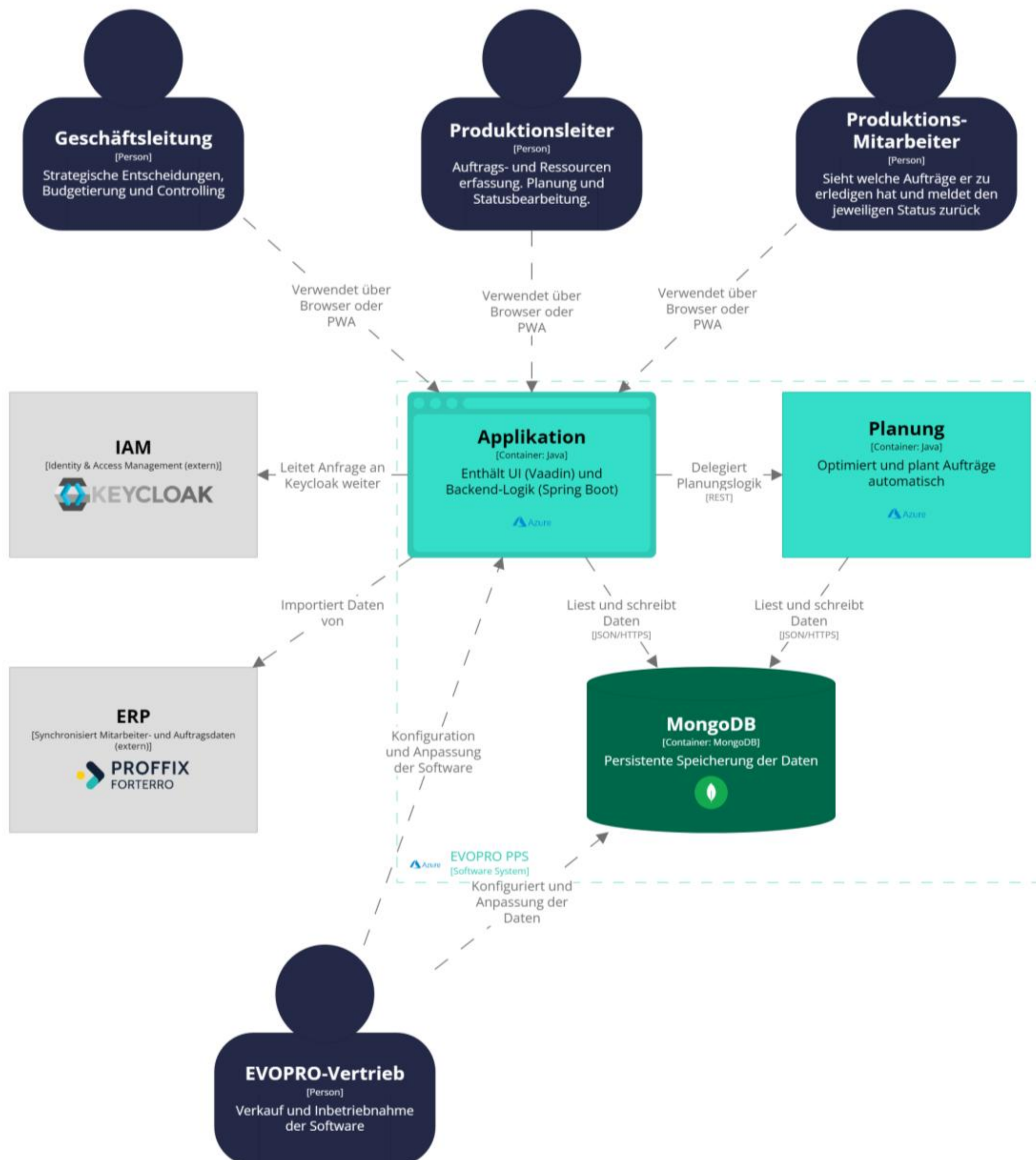

Abbildung 3: EVOPRO Container-Diagramm

Applikation (Spring Boot & Vaadin)

Die zentrale Webanwendung bündelt sowohl die Benutzeroberfläche (Vaadin) als auch die Backend-Logik (Spring Boot). Sie stellt sämtliche Funktionen für die Planung, Auftragserfassung, Rückmeldungen und Auswertungen bereit. Darüber hinaus koordiniert sie den Datenaustausch mit den internen Komponenten, delegiert Planungsaufgaben und steuert alle Lese- und Schreibzugriffe auf die Datenbank.

Planung

Eine spezialisierte Komponente, die Planungsaufträge entgegennimmt und automatisch optimierte Produktionspläne erstellt. Die Berechnungslogik ist vom restlichen System entkoppelt, sodass Änderungen oder Erweiterungen an der Planungsfunktionalität unabhängig von der Hauptanwendung umgesetzt werden können.

MongoDB

Die persistente Speicherung sämtlicher fachlicher und technischer Daten erfolgt in einer MongoDB-Instanz. Sie verwaltet unter anderem Stammdaten, Planungsstände, Konfigurationen und Rückmeldungen. Der Zugriff erfolgt ausschliesslich über die Applikations- oder Planungskomponente, wodurch Datenkonsistenz und Zugriffskontrolle gewährleistet werden.

Betriebshinweis

Die Applikation und Planungs-Komponente werden als Docker-Container in Azure betrieben (pro Kunde eine Instanz). Die Datenhaltung liegt zentral in MongoDB Atlas, wobei der Zugriff verschlüsselt und mandantenspezifisch getrennt ist.

2.2.3 Komponentenmodell

Das aktuelle Komponentenmodell (Brown, Software Architecture for Developers Vol. 2, 2016) von EVOPRO in Abbildung 4 zeigt eine monolithische Webanwendung auf Basis von Spring Boot und Vaadin, die in klassische technische Schichten unterteilt ist. Diese Schichtung – bestehend aus *UI, Service, Domain/Data, Repository*, technischen Hilfskomponenten und Konfiguration – ist grundsätzlich sinnvoll, wurde im Laufe der Zeit jedoch durch pragmatische Erweiterungen und das Fehlen verbindlicher Leitlinien teilweise inkonsistent umgesetzt.

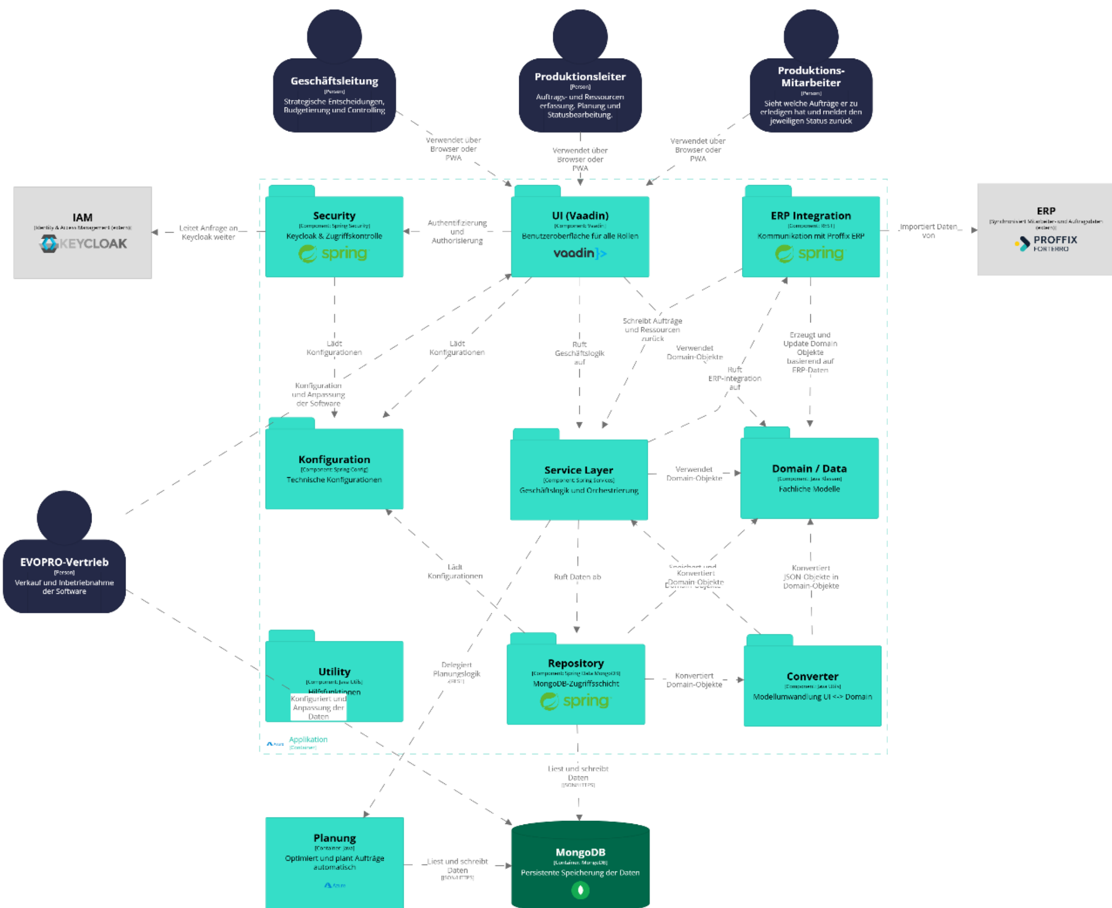

Abbildung 4: EVOPRO Ist-Komponentenmodell

Ein zentrales Problem besteht darin, dass Domänenobjekte unkontrolliert durch sämtliche Schichten „wandern". Die gleiche Klasse wird in UI, Service und Persistenzschicht verwendet, was zu hoher Kopplung führt und Änderungen an einem Ort häufig unerwartete Nebeneffekte in anderen Bereichen auslöst. Die Services selbst bilden nur selten vollständige Anwendungsfälle ab, sondern erfüllen überwiegend CRUD-Funktionen. Fachliche Logik verteilt sich so auf mehrere Klassen und dringt teilweise bis in die Frontend-Implementierung vor, was gezieltes Testen erschwert und das Risiko von Regressionen bei Änderungen erhöht.

Hinzu kommt, dass der separate Planungsservice fachlich eng an den Monolithen gekoppelt ist, dabei aber angepasste Kopien zentraler Domänenklassen enthält. Diese Redundanz steigert den Pflegeaufwand und birgt ein erhöhtes Inkonsistenzrisiko. Technische Komponenten wie die MongoDB-Converter – die in erster Linie der Schema-Migration dienen, etwa bei der Umstellung von primitiven Typen auf Value Objects – sind ein nützliches Werkzeug, werden aktuell jedoch ohne klar definierten Decommission-Plan eingesetzt. Dadurch besteht das Risiko, dass temporär gedachte Konverter langfristig im System verbleiben und die Komplexität unnötig erhöhen.

In der Praxis führen diese strukturellen Schwächen dazu, dass kleine Änderungen grosse Auswirkungen haben, neue Releases instabil werden können und

Performanceprobleme auftreten, etwa wenn synchrone ERP-Abfragen direkt im UI-Flow ausgeführt werden. Trotz dieser Herausforderungen bietet die bestehende Schichtung eine solide Grundlage, deren Potenziale aktuell jedoch nicht vollständig ausgeschöpft werden.

2.2.4 Domänen- / Klassenmodell

Das aktuelle Domänenmodell von EVOPRO dargestellt in Abbildung 5 orientiert sich stark an den im Produktionsumfeld benötigten fachlichen Entitäten. Es umfasst zentrale Konzepte wie Aufträge, Produkte mit zugehörigen Stücklisten, Ressourcen wie Maschinen und Mitarbeitende, Prozesse und Operationen, sowie ergänzende Strukturen zur Abbildung von Kunden, Adressen und Rollen. Die Modellierung bildet die fachlichen Anforderungen grundsätzlich vollständig ab und deckt sowohl produktionsrelevante als auch betriebswirtschaftliche Aspekte ab.

Die Entitäten sind in einem objektorientierten Klassenmodell dargestellt, das im Wesentlichen als Datenstruktur dient und in allen Schichten der Anwendung wiederverwendet wird. Domänenobjekte enthalten neben den Attributen teilweise auch fachliche Logik, werden jedoch ohne klare Abgrenzung zwischen interner Repräsentation, Persistenzmodell und externen Schnittstellen eingesetzt. Dadurch werden dieselben Klassen sowohl im UI als auch in den Services und Repositories verwendet.

Ein auffälliges Merkmal ist die enge Verflechtung zwischen einzelnen Domänenobjekten. Beziehungen wie zwischen Aufträgen, Produkten, Prozessen und Ressourcen sind direkt modelliert und führen zu einer starken Kopplung. Diese Kopplung erhöht die Komplexität bei Änderungen, da Anpassungen an einer Entität häufig Anpassungen an mehreren weiteren Klassen erforderlich machen.

Zudem spiegeln einzelne Teile des Domänenmodells Strukturen wieder, die auch im Planungsservice vorhanden sind. Hierbei handelt es sich jedoch nicht um gemeinsame zentrale Modelle, sondern um teilweise angepasste Kopien, die inhaltlich eng verwandt sind. Diese Duplizierung birgt die Gefahr von Inkonsistenzen, wenn Änderungen nicht synchron in allen betroffenen Bereichen erfolgen.

Insgesamt zeigt das Domänenmodell eine breite Abdeckung der relevanten Geschäftsobjekte, weist jedoch eine sehr enge Kopplung und eine fehlende Trennung zwischen den verschiedenen Verwendungskontexten auf. Dies erschwert die gezielte Weiterentwicklung, die Einführung neuer Funktionen und kann bei Änderungen zu unbeabsichtigten Seiteneffekten führen.

Abbildung 5: EVOPRO Ist-Klassenmodell

2.2.5 Fazit Architekturanalyse

Insgesamt zeigt die Architekturanalyse von EVOPRO ein System, das auf einer prinzipiell soliden technischen Schichtung und einem umfassenden Domänenmodell basiert, dessen Potenzial jedoch durch gewachsene Strukturen und fehlende Abgrenzungen zwischen den Schichten eingeschränkt wird. Die Analyse der Komponentenebene verdeutlicht, dass Domänenobjekte ohne klare Schnittstellen über alle Schichten hinweg verwendet werden und Services nur selten vollständige Anwendungsfälle abbilden. Dies führt zu hoher Kopplung, erschwert gezieltes Testen und erhöht das Risiko von Instabilitäten und Performanceproblemen. Auf Ebene des Domänenmodells wird diese Kopplung durch direkte, teils komplexe Beziehungen zwischen den Entitäten verstärkt, während redundante Modellteile im Planungsservice zusätzlich Inkonsistenzrisiken bergen. Diese strukturellen Gegebenheiten erklären, warum Änderungen am System oftmals weitreichende Auswirkungen haben und neue Releases vereinzelt zu unerwarteten Fehlern oder Leistungsverlusten führen.

2.3 Performance Analyse

Die Analyse der Serverreaktionszeiten in Abbildung 6 in zwei unterschiedlichen Systeminstanzen zeigt deutliche Unterschiede in der Performance. Während in System A (ohne ERP-Anbindung) die meisten Ansichten nahezu verzögerungsfrei geladen werden, treten in System B (mit ERP-Anbindung) spürbare Verzögerungen auf. Besonders auffällig ist dies in der Mitarbeiterübersicht, die in System B über drei Sekunden benötigt und damit rund 20-mal länger lädt als in System A, obwohl dort weniger Mitarbeitende verwaltet werden.

Die Ursache für diese Abweichungen liegt in der Architektur der Anwendung: In System B werden beim Öffnen der Mitarbeiteransicht direkte Abfragen an das ERP-System ausgeführt, um aktuelle Mitarbeiter- und Produktdaten zu laden. Diese synchrone Kopplung verlängert nicht nur die Antwortzeiten erheblich, sondern wirkt sich auch negativ auf die Skalierbarkeit und Wartbarkeit der Anwendung aus.

Die Vergleichbarkeit der beiden Systeme ist insofern eingeschränkt, als System A mehr aktive Aufträge und Mitarbeitende, dafür aber deutlich weniger Maschinen und Produkte aufweist wie in Tabelle 2 dargestellt wird. Dennoch ist der Performanceunterschied signifikant, da die betroffene Ansicht in System B trotz geringerer Datenmenge erheblich langsamer reagiert.

Hinweis	**Ohne ERP-Anbindung (System A)**	**Mit ERP-Anbindung (System B)**
Aktive Aufträge	87	19
Mitarbeitende	38	16
Maschinen	2	40

Hinweis	Ohne ERP-Anbindung (System A)	Mit ERP-Anbindung (System B)
Produkte	45	291

Tabelle 2:Vergleich der zwei Systeme

Ein weiterer Befund betrifft die Ansicht der Produktionsplanung: Diese benötigt in beiden Systemen zwischen fünf und sechs Sekunden Ladezeit. Die Ursache hierfür liegt in der aufwendigen Generierung der Ansicht und ist nicht unmittelbar in der Architektur begründet. Daher steht diese Optimierung nicht im Fokus der vorliegenden Arbeit.

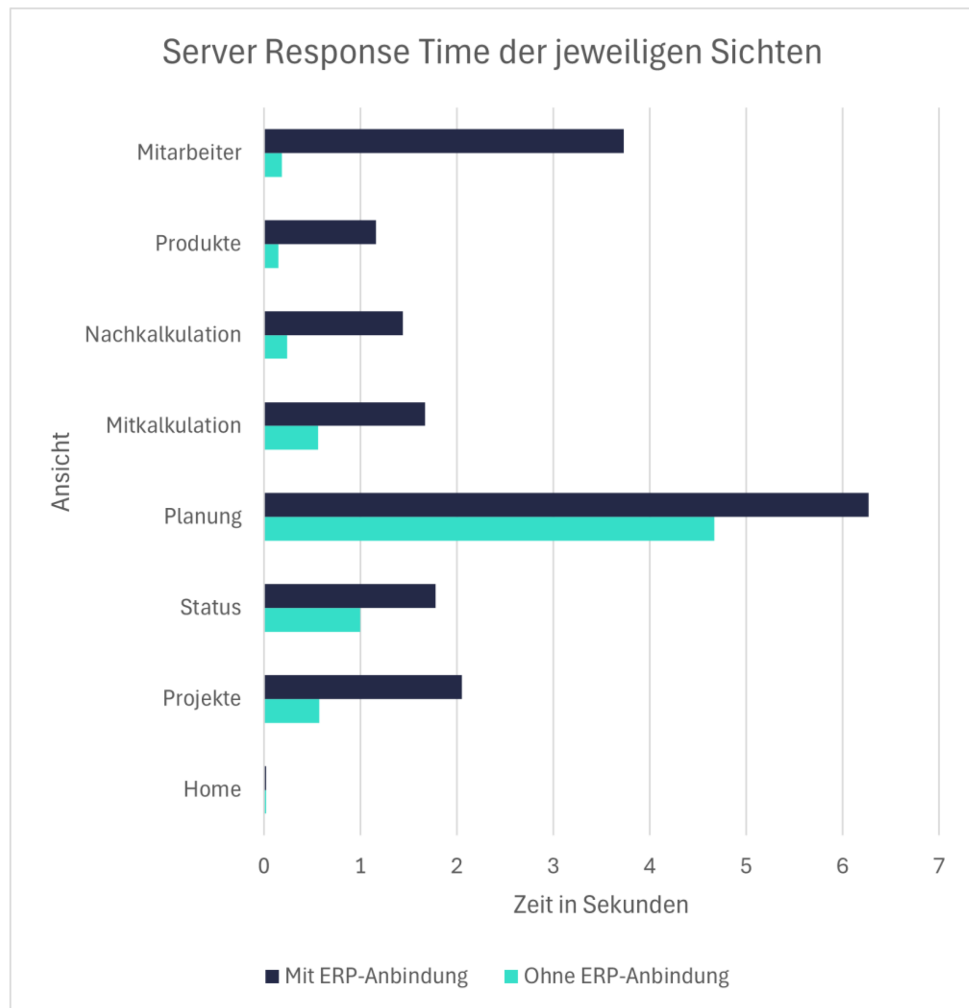

Abbildung 6: Server Response Time der aktuelle EVOPRO Applikation

2.4 Code-Qualitätsanalyse

Die statische Codeanalyse mit SonarQube[5] in Abbildung 7 zeigt, dass EVOPRO in Bezug auf Sicherheit ein sehr gutes Ergebnis erreicht (Rating A, keine offenen Issues).

[5] Vgl. Quelle: Offizielle Dokumentation: https://www.sonarsource.com

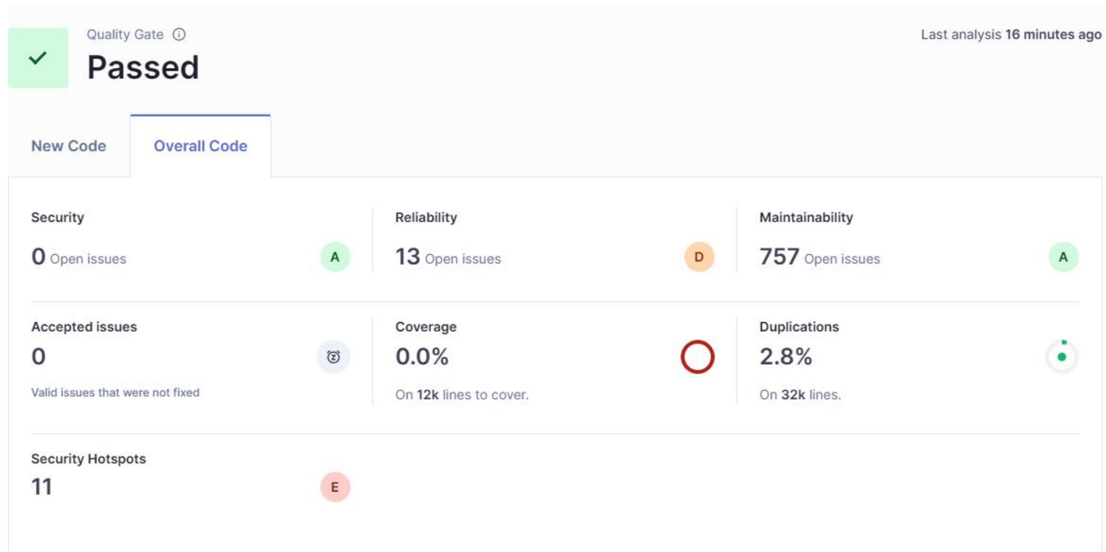

Abbildung 7: Übersicht der SonarQube Qualitäts Analyse

Auch die Maintainability wird insgesamt mit A bewertet, weist jedoch 757 offene Maintainability-Issues auf, die sich in Summe zu einer geschätzten technischen Schuld von mehreren Arbeitstagen addieren. Die Analyse der Maintainability Overview in Abbildung 8 verdeutlicht, dass der Grossteil der Klassen im grünen Bereich liegt, jedoch einzelne Klassen mit deutlich höherem Wartungsaufwand existieren. Besonders grosse Klassen mit über 300 Zeilen Code sind dabei häufiger betroffen und verursachen jeweils mehrere Stunden potenziellen Korrekturaufwand.

Abbildung 8: Maintainability Overview der SonarQube Qualitätsanalyse

In der Kategorie Reliability wird die Codebasis mit D bewertet, was auf 13 offene Issues zurückzuführen ist. Diese sind über verschiedene Module verteilt und weisen überwiegend einen geringen bis mittleren Behebungsaufwand auf. Die Reliability Overview in Abbildung 9 zeigt, dass vor allem mittelgrosse Klassen mit 100 bis 250

Zeilen Code in diesem Bereich auffallen und überwiegend in den gelben bis orangen Bewertungsbereich fallen.

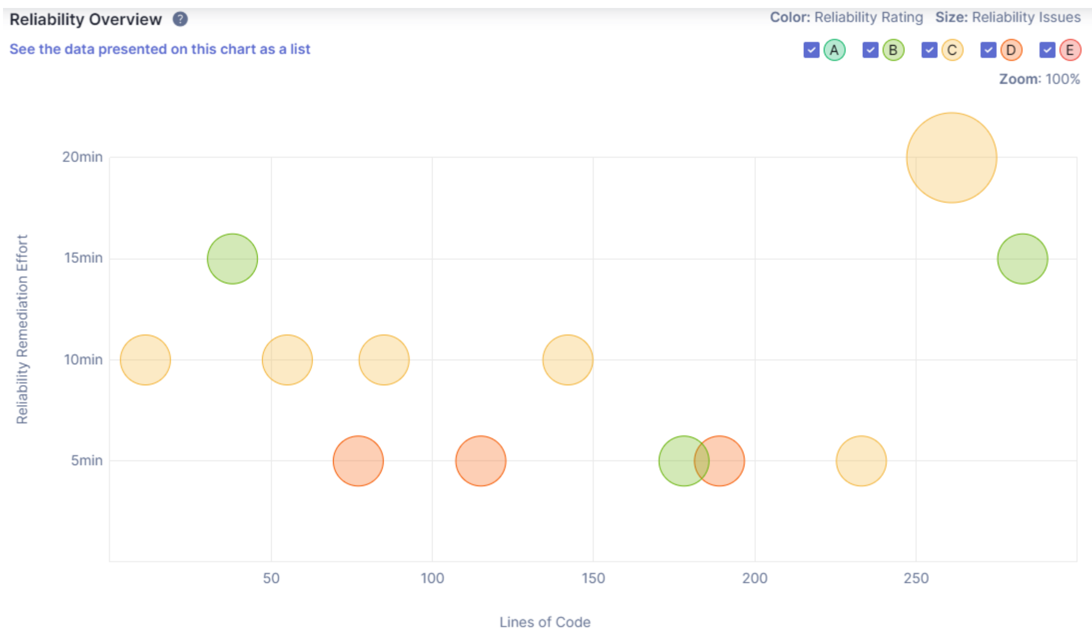

Abbildung 9: Reliability Overview der Sonar Qube Qualitäts Analyse

Auffällig ist die vollständige Abwesenheit automatisierter Tests im analysierten Quellcode (0 % Test Coverage bei rund 12 000 zu testenden Zeilen). Dies bedeutet, dass mögliche Regressionen oder Seiteneffekte bei Änderungen nur schwer automatisiert erkannt werden können. Darüber hinaus weist der Code einen moderaten Anteil an Duplikationen auf (2,8 % bei rund 32 000 Zeilen Code).

Im Bereich Security Hotspots wurden elf Stellen identifiziert, die potenziell sicherheitsrelevant sein könnten, jedoch nicht zwingend Sicherheitslücken darstellen. Hierbei handelt es sich um Codeabschnitte, die in einem sicherheitskritischen Kontext stehen und manuell geprüft werden müssen.

Insgesamt bestätigen die Ergebnisse, die in der Architekturanalyse identifizierten, strukturellen Schwächen: eine teilweise hohe Komplexität einzelner Klassen, Code-Duplizierungen und fehlende Testabdeckung. Diese Faktoren wirken sich direkt auf die Wartbarkeit und Stabilität der Anwendung aus.

2.5 Fazit der Analyse

Die Analyse verdeutlicht, dass EVOPRO zwar auf einer soliden technischen Basis steht, jedoch durch enge Kopplungen, fehlende Schichtentrennung und mangelnde Testabdeckung in seiner Wartbarkeit und Stabilität eingeschränkt ist. Besonders kritisch wirken sich die synchrone ERP-Anbindung und die fehlenden automatisierten Tests aus. Damit wird klar, dass die Hauptursachen für Instabilitäten und hohen Wartungsaufwand weniger in den Funktionen selbst, sondern in strukturellen Schwächen der Architektur liegen.

3 Zieldefinition

3.1 Übergeordnetes Projektziel

Das Architektur-Refactoring von EVOPRO verfolgt das Ziel, eine zukunftsfähige, modulare und wartbare Architektur zu schaffen, die langfristig den Betrieb sichert, die schnelle Umsetzung neuer Kundenanforderungen ermöglicht und den Wartungs- und Supportaufwand deutlich reduziert.

Wesentliche Leitlinien dabei sind:

- Klare Entkopplung von Komponenten und Schichten, um Abhängigkeiten zu minimieren und Änderungen sicherer umzusetzen.
- Saubere Trennung von Verantwortlichkeiten durch eindeutige Architektur- und Code-Strukturen.
- Hohe Code-Qualität mit einheitlichen Standards und geringem Anteil an Duplikationen.
- Hohe Testabdeckung zur Sicherstellung von Stabilität und Vermeidung von Regressionen.

3.2 Architektonisches Ziel

Ziel des Refactorings ist es, EVOPRO so zu restrukturieren, dass eine klar gegliederte, wartbare und langfristig erweiterbare Architektur entsteht. Dabei wird die Software nicht vollständig neu entwickelt, sondern in ihrer Struktur so angepasst, dass Änderungen sicherer und isoliert durchgeführt werden können. Grundlage bilden bewährte Architekturstile wie der Modulare Monolith (Brown, Modular Monoliths, 2018) und die Clean Architecture (Martin, 2018).

3.2.1 Modularisierung in vertikale Einheiten

Die Codebasis wird in fachlich klar abgegrenzte Module überführt, die jeweils alle für ihren Funktionsbereich benötigten Schichten beinhalten. Diese Aufteilung orientiert sich am im Komponentenmodell in Abbildung 10 dargestellten Aufbau.

Als Leitprinzip dient der Modulare Monolith nach Simon Brown (Brown, Modular Monoliths, 2018). Dieses Architekturkonzept kombiniert die Vorteile einer monolithischen Anwendung – Einfachheit, gemeinsame Deployment-Pipeline, konsistente Datenhaltung – mit den Vorteilen einer internen Modularisierung. Jedes Modul ist weitgehend unabhängig, Änderungen in einem Modul sollen keine unbeabsichtigten Auswirkungen auf andere Module haben.

Die Modularisierung folgt dabei den fachlichen Verantwortlichkeiten, wie sie im Komponentenmodell abgebildet sind. Beispiele für eigenständige Module sind etwa Auftrag, Produkt, Ressourcen, Planung, Finanzen, Reporting, User und die ERP-

Integration. Jedes dieser Module kapselt seine eigene Logik, seine Datenzugriffe sowie eine klar definierte Schnittstelle zu anderen Modulen.

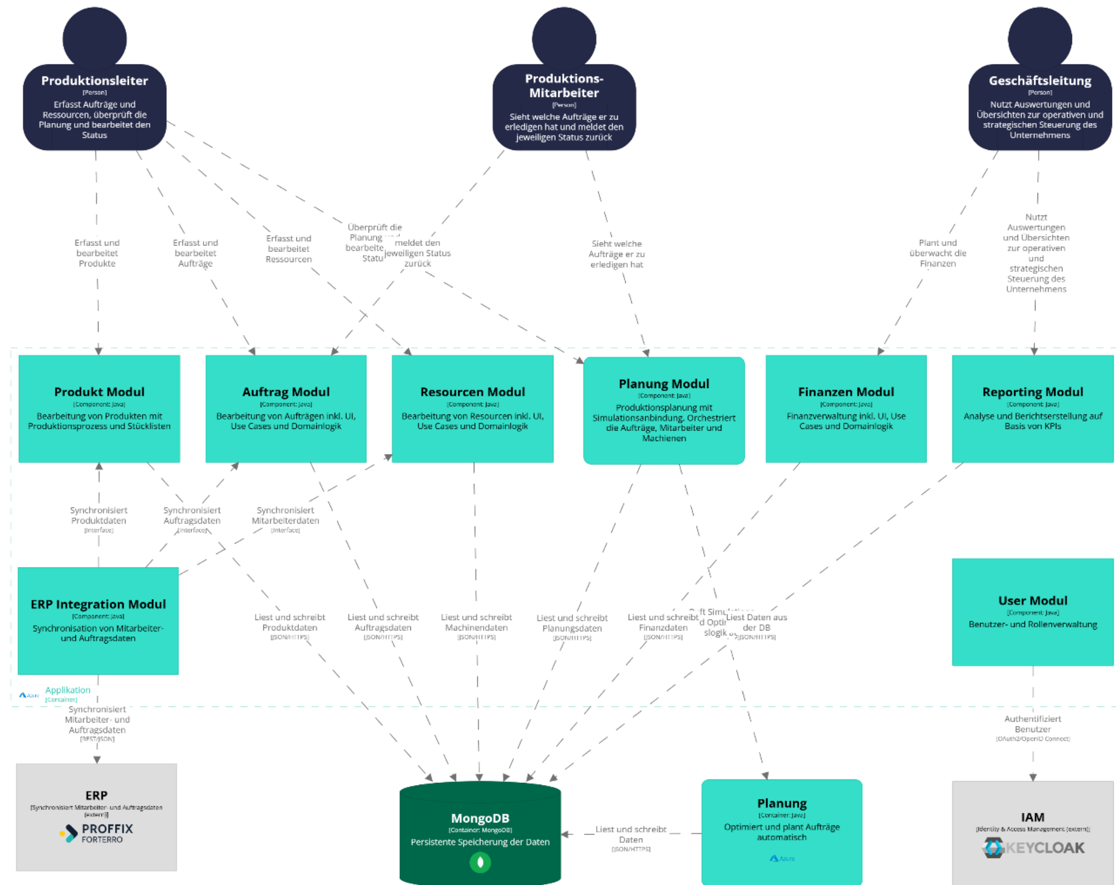

Abbildung 10: EVOPRO Soll-Komponentenmodell

3.2.2 Horizontale Schichtung nach Clean Architecture

Innerhalb jedes Moduls wird die Architektur nach den Prinzipien der Clean Architecture (Martin, 2018) gestaltet. Die Schichten sind klar voneinander getrennt und übernehmen spezifische Rollen. Das Naming im Code ist teilweise abweichend zum Clean Architecture Konzept. Die Abbildung 11 zeigt in Klammer den jeweilige Clean Architecture Namen, die im folgenden beschrieben werden:

- *Domain*: entspricht der Entitäten-Schicht. Hier werden die zentralen fachlichen Klassen modelliert, die keinerlei Abhängigkeiten zu externen Frameworks aufweisen.
- *Application*: entspricht der UseCase-Schicht. Hier sind die Anwendungsfälle definiert, welche die fachliche Logik kapseln.
- *Application.in*: Interfaces für die Use-Case-Eingabeports, über die Use Cases von aussen angesteuert werden (z. B. vom Controller).

- *Application.out*: Interfaces für die Use-Case-Ausgabeports, über die externe Ressourcen wie die Datenbank angebunden werden.
- *Repository*: Implementiert die Out-Ports und entspricht dem Gateway. Hier werden DAOs, Mappings und die konkreten Datenbankzugriffe realisiert.
- *Controller*: übernimmt die Rolle der Eingabeschicht. Er ruft die definierten Use Cases an und dient als Bindeglied zwischen Webebene und Application Layer.
- *Web*: entspricht der Präsentationsschicht. Diese wird zweigeteilt umgesetzt:
    - *UI*: realisiert mit Vaadin, stellt die grafische Benutzeroberfläche bereit und kommuniziert ausschliesslich über Controller mit den Use Cases.
    - *REST*: implementiert die RESTful API mit Spring Boot, stellt Endpunkte für den externen Zugriff bereit und bindet ebenfalls nur die Controller-Schicht an.

Die zentrale Regel der Clean Architecture wird strikt eingehalten: *Domain*- und *Application*-Schichten haben keine Abhängigkeiten zu externen Bibliotheken. Sie arbeiten ausschliesslich mit Plain Java Objekten und definierten Schnittstellen.

Zulässig sind lediglich Abhängigkeiten zum Common-Modul innerhalb derselben Abstraktionsebene oder tiefer. So darf z. B. eine Klasse aus der *Domain*-Schicht auf eine abstrakte Klasse im Common-Domain-Bereich zugreifen oder ein Use Case in *Application* auf eine abstrakte Definition im Common-Application-Bereich. Querabhängigkeiten über unterschiedliche Ebenen hinweg sind hingegen nicht erlaubt.

Durch diese klare Strukturierung wird die Trennung von Fachlogik und technischer Infrastruktur gewährleistet, die Testbarkeit erhöht und die langfristige Unabhängigkeit von spezifischen Frameworks sichergestellt.

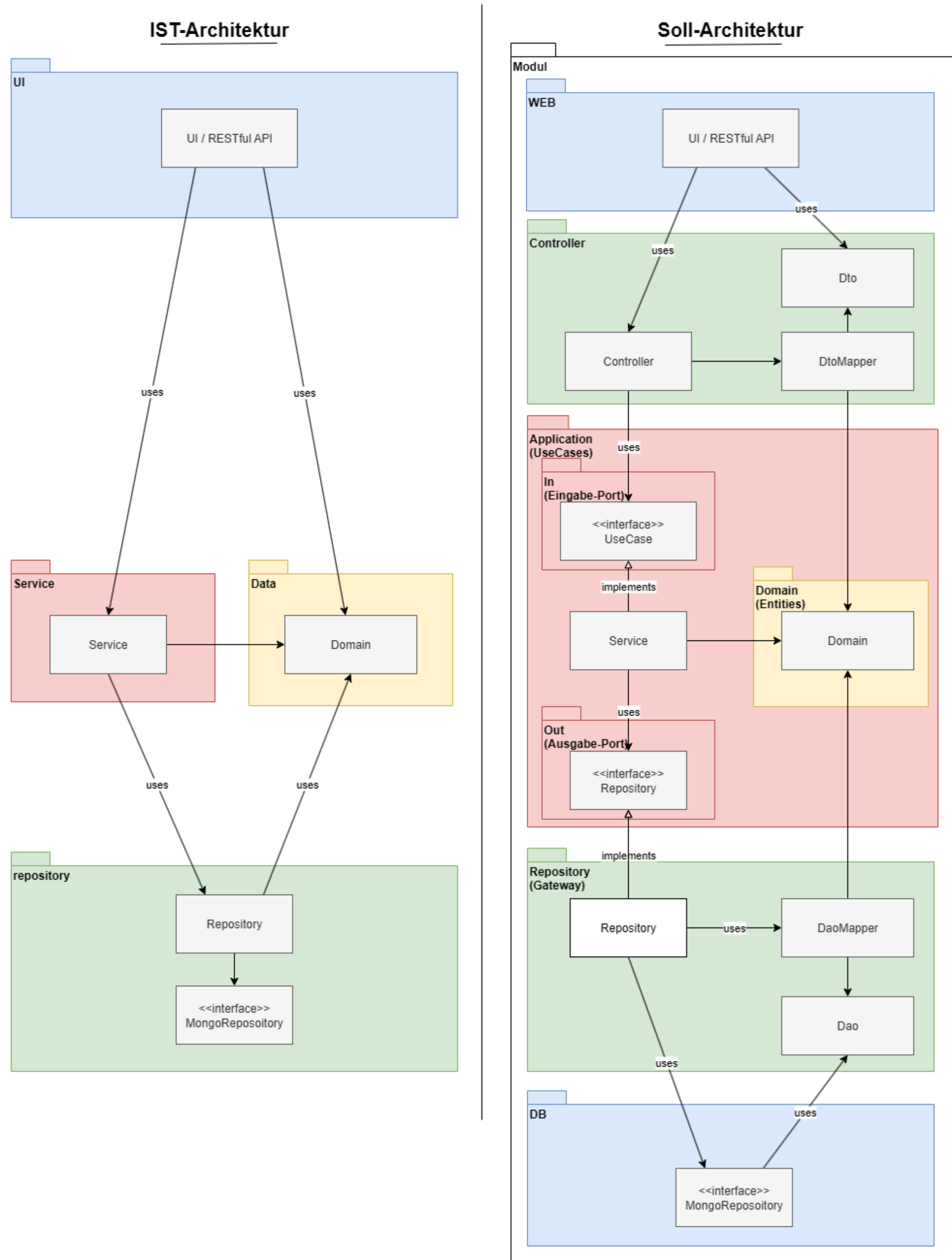

Abbildung 11: Vergleich Ist- und Soll-Architektur

3.2.3 Use-Case zentrierte Modulkommunikation

Mit der neuen Architektur rückt die fachliche Logik konsequent ins Zentrum der Anwendung und wird als klar abgegrenzte, zentrale Ressource innerhalb jedes Moduls behandelt. Anstelle der bisherigen, vorwiegend CRUD-getriebenen Servicekommunikation werden die Module künftig ausschliesslich über klar definierte Anwendungsfälle (Use Cases) angesprochen, die vollständige fachliche Abläufe abbilden. Dadurch verschiebt sich der Fokus weg von rein datenorientierten Operationen hin zu einer domänenzentrierten Arbeitsweise, bei der die Absicht („Was soll passieren?“) wichtiger ist als die konkrete Datenmanipulation.

Die Benutzeroberfläche übernimmt in diesem Modell ausschliesslich die Rolle eines Präsentations- und Interaktionslayers. Sie enthält keinerlei Geschäftslogik mehr, sondern konsumiert ausschliesslich die im Modul bereitgestellten Anwendungsfälle. Diese klare Trennung sorgt dafür, dass Änderungen an der UI nicht zu unbeabsichtigten Eingriffen in die Fachlogik führen – und umgekehrt.

Auch die Kommunikation zwischen den Modulen erfolgt über diese eindeutigen, lose gekoppelten Schnittstellen. Diese Form der Schnittstellendefinition reduziert Abhängigkeiten, erhöht die Austauschbarkeit von Implementierungen und stellt sicher, dass fachliche Details klar innerhalb ihres Moduls verbleiben. Dadurch wird verhindert, dass interne Logik unkontrolliert in andere Bereiche des Systems übergreift und dort unerwünschte Abhängigkeiten erzeugt.

Da die zentrale Fachlogik in den Use Cases innerhalb der jeweiligen fachlichen Domäne gebündelt ist, verbessert sich auch die Test- und Wartbarkeit: Use Cases lassen sich isoliert testen, unabhängig von UI, Datenbank oder externen Integrationen. So wird die langfristige Pflege und Weiterentwicklung der Anwendung erleichtert.

3.2.4 Klare Schnittstellen zu externen Systemen

Ein zentrales Ziel des Refactorings ist die Neugestaltung der Anbindung externer Systeme, insbesondere des ERP-Systems. In der bestehenden Architektur erfolgt der Zugriff synchron direkt aus der Benutzeroberfläche: Beim Öffnen bestimmter Ansichten werden ERP-Daten in Echtzeit abgefragt. Diese enge Kopplung führt zu langen Ladezeiten und macht die Anwendung anfällig für Ausfälle oder Verzögerungen im angebundenen System.

Künftig werden externe Systeme über ein eigenständiges Integrationsmodul angebunden, das unabhängig von UI und Fachlogik arbeitet. Dieses Modul übernimmt den bidirektionalen Datenaustausch mit dem ERP-System asynchron, sodass Informationen nicht mehr während der UI-Interaktion geladen werden müssen. Stattdessen werden Änderungen ereignisgesteuert synchronisiert und lokal in EVOPRO zwischengespeichert.

Durch diese Architektur entsteht eine flexible Integrationsschicht, die sowohl die Stabilität als auch die Performance verbessert: Die Kernlogik von EVOPRO bleibt von externen Systemen entkoppelt, Benutzeroberflächen reagieren schneller und das Gesamtsystem bleibt robuster gegenüber Störungen in angebundenen Diensten.

3.3 Qualitätsziele

Die Erreichung der in dieser Zieldefinition beschriebenen Architekturprinzipien wird anhand folgender messbarer Kriterien überprüft:

- **Architekturentkopplung**: Mindestens drei vollständig entkoppelte fachliche Module mit klar definierten Schnittstellen.

- **Testabdeckung**: Bei jedem umgesetzten Modul ist die Use Case Test-Abdeckung bei 100%. Dies beinhaltet sowohl die Domänen-, als auch die Applicaiton-Klassen. Die Infrastruktur- und UI-Schichten werden nicht explizit getestet, sondern lediglich über Integrations- oder Funktionstests indirekt abgesichert.
- **Code-Duplizierung:** Reduktion der Code-Duplizierung auf ≤ 1,4 % gemäss SonarQube[6].
- **Sicherheitsaspekte**: Beseitigung aller aktuell bestehenden Security Hotspots sowie aller Maintainability-Issues mit hoher Auswirkung.
- **Performance**: Ladezeiten von ERP-abhängigen Ansichten maximal gleich wie bei einer Instanz ohne ERP-Anbindung.

3.4 Abgrenzung

Nicht Teil dieser Arbeit sind:

- Funktionale Erweiterungen des Systems über den bestehenden Funktionsumfang hinaus.
- Performance-Optimierungen einzelner Ansichten, deren Ursache nicht in der Architektur liegt (z. B. komplexe Renderprozesse der Planungsübersicht).
- Einführung einer Microservice-Architektur oder vollständige Umsetzung der Clean Architecture in allen Details.

3.5 Erfolgskontrolle

Der Projekterfolg wird auf Basis objektiver Kriterien überprüft:

- **Technische Metriken:** Auswertung der SonarQube-Analyse und Testabdeckung vor und nach dem Refactoring.
- **Architekturvalidierung:** Für jedes Modul ist ein ArchUnit-Test zu definieren, der erfolgreich durchlaufen werden muss und die jeweiligen Architekturrestriktionen (z. B. Schichtentrennung, Zugriffsbeschränkungen) überprüft.
- **Leistungstests:** Vergleichsmessungen der Ladezeiten vor und nach der Umsetzung.

3.6 Fazit der Zieldefinition

Mit der beschriebenen Zieldefinition wird eine klare Ausrichtung für das Architektur-Refactoring von EVOPRO geschaffen. Die Kombination aus modularer horizontaler Struktur, strikter vertikaler Schichtung, Use-Case-zentrierter Kommunikation und klar definierten Schnittstellen zu externen Systemen legt die Grundlage für ein wartbares, erweiterbares und qualitativ hochwertiges System. Durch die festgelegten Qualitätsziele und die präzise Abgrenzung des Projektumfangs ist der Rahmen für die Umsetzung klar gesteckt. Die definierten Erfolgskriterien, einschliesslich der

[6] SonarSource: SonarQube Dokumentation, https://docs.sonarsource.com

automatisierten Architekturvalidierung per ArchUnit[7]-Tests, ermöglichen eine objektive Überprüfung der Zielerreichung. Damit ist sichergestellt, dass das Refactoring nicht nur kurzfristige Verbesserungen bringt, sondern die langfristige Stabilität und Weiterentwicklung des Systems unterstützt.

[7] ArchUnit: ArchUnit Dokumentation, https://www.archunit.org/

4 Vorgehensweise

Um die in der Zieldefinition definierten Ziele zu erreichen, wird ein strukturiertes und schrittweises Vorgehen gewählt. Dabei liegt der Fokus darauf, die bestehenden Strukturen kontrolliert zu verändern, um Risiken zu minimieren und jederzeit einen lauffähigen Systemzustand sicherzustellen. Die Umsetzung erfolgt in klar abgegrenzten Arbeitspaketen, die jeweils aufeinander aufbauen und nach jedem Schritt überprüfbare Zwischenergebnisse liefern. Neben der technischen Umsetzung werden begleitend Massnahmen zur Qualitätssicherung, Dokumentation und Architekturvalidierung durchgeführt, um sicherzustellen, dass die angestrebte Soll-Architektur konsequent umgesetzt wird.

4.1 Methodisches Vorgehen

Die Umsetzung des Architektur-Refactorings erfolgt in einem iterativen Vorgehensmodell, das sich am Grundgedanken von Scrum orientiert, jedoch an die Rahmenbedingungen des Projekts angepasst ist. Die Arbeit wird in zweiwöchige Sprints unterteilt, auf die jeweils eine zwei- bis vierwöchige Pause folgt. Dieser Vier- bis Sechswochenrhythmus ermöglicht es, in den Umsetzungsphasen fokussiert technische Anpassungen vorzunehmen und in den Pausen Feedback auszuwerten, Tests zu ergänzen und die nächsten Schritte zu planen.

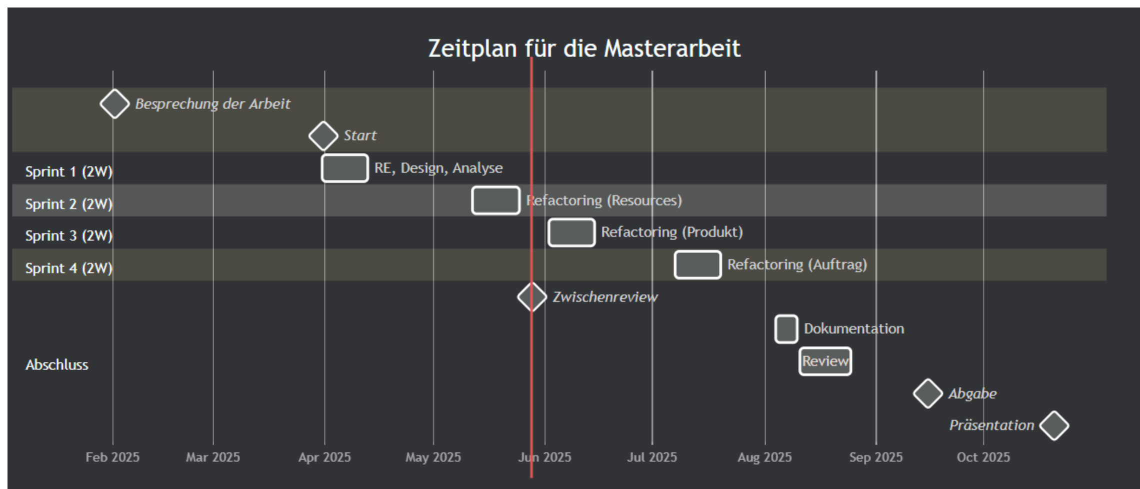

Abbildung 12: Umsetzungszeitplan Refactoring EVOPRO

Für die Planung und Nachverfolgung der Arbeitspakete wird ein Kanban-Board in Jira (Abbildung 13) eingesetzt. Der Quellcode wird versioniert und über GitHub verwaltet, während die bestehende Build- und Deployment-Pipeline in Azure DevOps integriert ist. Zur Sicherstellung der Codequalität und Einhaltung der Architekturprinzipien werden statische SonarQube-Analysen ausgeführt und die ArchUnit-Tests dienen der kontinuierlichen Überprüfung der architekturellen Restriktionen in jedem Modul.

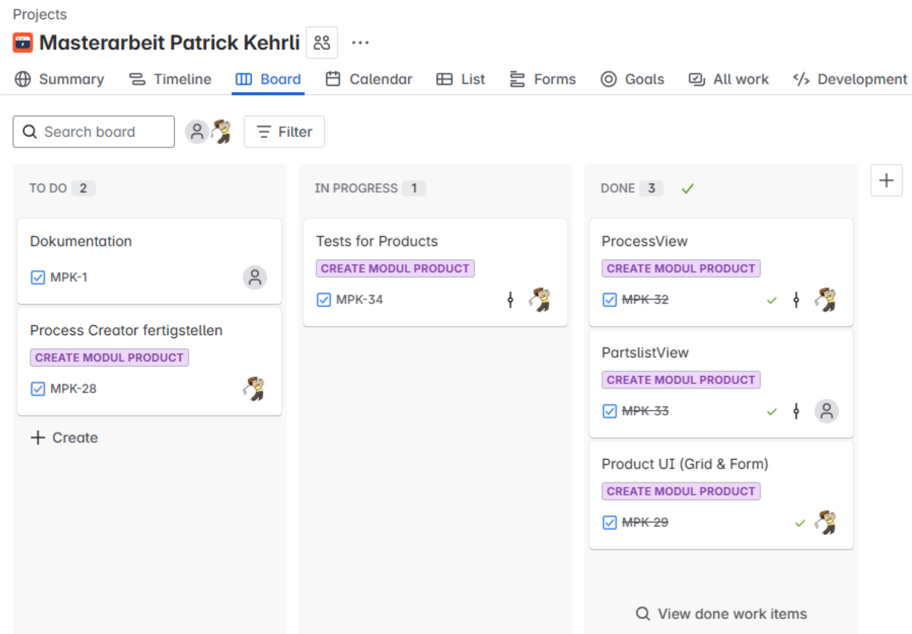

Abbildung 13: Ausschnitt des Kanban-Boards in Jira

4.2 Umsetzungsschritte

Die Umsetzung erfolgt pro Modul in mehreren aufeinander aufbauenden Phasen, die sich an den in Kapitel 3 definierten, architektonischen Zielen orientieren. Jedes Modul wird dabei vollständig durch alle Schritte geführt, bevor das nächste Modul beginnt. Dadurch lassen sich Änderungen kontrolliert umsetzen und frühzeitig Erfahrungen aus den ersten Modulen auf weitere übertragen.

1. Analysephase

Zu Beginn wird der bestehende Code hinsichtlich seiner Struktur und Abhängigkeiten untersucht. Dabei werden insbesondere Stellen identifiziert, an denen eine Entkopplung erforderlich ist. Weiter werden die fachlichen Modulgrenzen festgelegt. Parallel dazu werden die bestehenden Integrationen zu externen Systemen analysiert.

2. Modularisierung in vertikale Einheiten

Die Codebasis wird gemäss dem Soll-Komponentenmodell in fachlich abgegrenzte Module überführt.

Die Kommunikation zwischen den Modulen erfolgt ausschliesslich über deren Controller. Andere Module greifen somit nicht direkt auf Anwendungsfälle oder interne Klassen zu, sondern nutzen die im Controller bereitgestellten Schnittstellen, welche die jeweiligen Use Cases aufrufen. Dadurch bleiben die internen Strukturen gekapselt, und die Module können unabhängig voneinander weiterentwickelt werden.

Zur Entkopplung werden bestehende Objektbeziehungen angepasst. Anstelle ganzer Objekte werden nur deren IDs gespeichert. So verwaltet jedes Modul ausschliesslich die eigenen Entitäten, während zugleich zyklische Abhängigkeiten und unnötige Kopplung vermieden werden.

3. Horizontale Schichtung nach Clean Architecture

Innerhalb der Module wird die Schichtung nach den Clean-Architecture-Prinzipien eingeführt. Jedes Modul erhält eine eigene interne Struktur mit *Web, Controller, Application, Domain* und *Repository*. *Domain*- und *Application*-Layer werden von externen Bibliotheken entkoppelt, und die Kommunikation zwischen den Schichten erfolgt ausschliesslich über Interfaces.

4. Einführung Use-Case-zentrierter Modulkommunikation

CRUD-basierte Service-Aufrufe werden durch fachliche Use Cases ersetzt. Die öffentlichen Inbound-Ports eines Moduls definieren dessen API, die sowohl von internen UI-Controllern als auch von anderen Modulen verwendet wird.

5. Integration externer Systeme

Die bisherige synchrone Kopplung zu externen Systemen (z. B. ERP) wird durch ein eigenständiges Integrationsmodul ersetzt. Dieses Modul übernimmt den Datenaustausch asynchron und ist vollständig von UI und Fachlogik entkoppelt. Dadurch bleibt die Kernanwendung auch bei Verzögerungen oder Ausfällen externer Systeme stabil, während die Anbindung flexibel und erweiterbar gestaltet werden kann.

6. Qualitätssicherung und Architekturvalidierung

Parallel zu allen Schritten werden automatisierte Unit- und ArchUnit-Tests implementiert. SonarQube-Analysen begleiten jede Entwicklungsphase, um Codequalität und Einhaltung der Architekturregeln zu überprüfen.

4.3 Risiken und Massnahmen

Das Refactoring einer bestehenden, produktiv genutzten Anwendung birgt technische und organisatorische Risiken. Diese werden im Vorfeld identifiziert und durch gezielte Massnahmen minimiert.

Komplexität der Architekturänderungen
Bei der Umsetzung der geplanten Entkopplung und horizontalen Schichtung besteht die Gefahr, dass bestehende Abhängigkeiten übersehen und nicht vollständig entfernt werden.
Massnahmen: Sorgfältige Analyse des Moduls vor der Umsetzung, konsequente Anwendung der in Kapitel 3 definierten Architekturprinzipien sowie Einsatz von ArchUnit-Tests zur automatisierten Überprüfung der Einhaltung.

Unklare Modulgrenzen
Werden die fachlichen Verantwortlichkeiten nicht klar getrennt, können

Querverbindungen zwischen Modulen bestehen bleiben, was die Wartbarkeit beeinträchtigt.
Massnahmen: Verbindliche Festlegung der Modulgrenzen im Komponentenmodell sowie Review der geplanten Schnittstellen vor Beginn der Umsetzung.

Unzureichende Testabdeckung
Neue Strukturen ohne ausreichende Testabdeckung gefährden die Überprüfbarkeit und Qualität der Umsetzung.
Massnahmen: Klare Teststrategie mit Fokus auf Unit-, Integrations- und ArchUnit-Tests, verbindliche Zielwerte für die Testabdeckung gemäss den in Kapitel 3.3 definierten Qualitätszielen.

4.4 Erfolgskontrolle

Die Überprüfung des Umsetzungserfolgs erfolgt anhand der in Kapitel 3.3 und 3.5 definierten Qualitätsziele und Metriken.
Für jedes umgesetzte Modul werden folgende Punkte kontrolliert:

- **Einhaltung der Architekturprinzipien**
  Überprüfung der Modulstruktur und Schichtentrennung anhand der ArchUnit-Tests. Diese müssen für jedes Modul erfolgreich durchlaufen und die definierten Architekturrestriktionen abbilden.
- **Testabdeckung**
  Messung der automatisierten Testabdeckung auf *Domain*- und *Application*-Ebene gemäss den festgelegten Zielwerten.
- **Codequalität**
  Analyse der Codebasis mit SonarQube, um die Einhaltung der Vorgaben zu Code-Duplizierung, Sicherheitsaspekten und Maintainability sicherzustellen.

Die Ergebnisse der Erfolgskontrolle werden am Ende der Masterarbeit in einer Gesamtbewertung zusammengefasst, um den Grad der Zielerreichung transparent darzustellen.

4.5 Fazit Vorgehensweise

Das gewählte Vorgehen kombiniert eine modulweise Umsetzung mit klar definierten Arbeitsschritten, kontinuierlicher Qualitätssicherung und automatisierter Architekturvalidierung. Durch diese strukturierte Herangehensweise wird sichergestellt, dass die in Kapitel 3 definierten Ziele schrittweise, überprüfbar und nachhaltig erreicht werden können. Die modulare Vorgehensweise erlaubt es zudem, Erfahrungen aus den ersten Umsetzungen direkt in die weiteren Module zu übertragen und so Effizienz und Qualität im Projektverlauf kontinuierlich zu steigern.

5 Umsetzung

Aufbauend auf der in Kapitel 3 definierten Zielarchitektur und dem in Kapitel 0 beschriebenen Vorgehensmodell wird in diesem Kapitel die konkrete Umsetzung des Architektur-Refactorings beschrieben. Die Umsetzung erfolgt pro Modul und orientiert sich an den definierten Arbeitsschritten, um eine schrittweise und überprüfbare Transformation der bestehenden Codebasis zu gewährleisten.

Für jedes bearbeitete Modul werden zunächst die relevante Ausgangslage und die identifizierten Schwachstellen dargestellt. Anschliessend wird erläutert, welche Änderungen vorgenommen wurden, um die horizontalen und vertikalen Architekturprinzipien umzusetzen, die Use-Case-zentrierte Kommunikation einzuführen und externe Schnittstellen über Adapter zu entkoppeln. Besonderes Augenmerk liegt dabei auf der konsequenten Einhaltung der Architekturregeln, die durch automatisierte Tests wie ArchUnit validiert werden.

5.1 Modul Ressourcen

Das Modul Ressourcen bildet in EVOPRO die Grundlage für die Verwaltung von Maschinen, Personal und Werkzeugen, die in der Produktionsplanung eingesetzt werden. In der bestehenden Architektur war dieses Modul stark mit anderen Bereichen verknüpft und wies keine klaren Modulgrenzen auf, was die Wartung und Weiterentwicklung erschwerte.

In den folgenden Abschnitten werden zunächst die Ausgangslage und die identifizierten Probleme beschrieben. Anschliessend wird dargestellt, welche konkreten Massnahmen zur Umsetzung der Zielarchitektur ergriffen wurden, welche technischen Herausforderungen dabei zu bewältigen waren und welche Ergebnisse erzielt werden konnten. Abschliessend wird das Fazit für dieses Modul gezogen.

5.1.1 Ausgangslage

Das Modul Ressourcen bildet den Grundstein für die Produktionsplanung. Ohne die darin verwalteten Maschinen, Werkzeuge, Mitarbeiter und weiteren Kapazitäten können keine Produkte hergestellt werden. In der bisherigen Architektur war dieses Modul stark mit den Prozessen und Produkten verknüpft, die wiederum direkt mit den Bestellungen verbunden waren. Änderungen an Ressourcen hatten dadurch häufig ungewollte Seiteneffekte bis in die Planungsansicht und Auswertungen hinein, was die Pflege und Weiterentwicklung erschwerte.

Das Domänenmodell in Abbildung 14 zeigt die Verknüpfung der Ressourcen vor dem Refactoring. Die meisten Entitäten entsprechen auch nach der Umstrukturierung den ursprünglichen Klassen, jedoch wurden die direkten Verbindungen zu Produkten und Prozessen inzwischen entfernt. In der Ist-Situation werden die Entitäten unverändert direkt in der Datenbank gespeichert und enthalten persistenzspezifische Annotationen wie @Id. Eine saubere Trennung zwischen Domänen-Logik und Persistenzschicht ist

somit nicht gegeben. Zudem existiert für dieses Modul keinerlei automatisierte Testabdeckung, wie bereits in Kapitel 3 beschrieben, was die Verlässlichkeit von Änderungen zusätzlich beeinträchtigt.

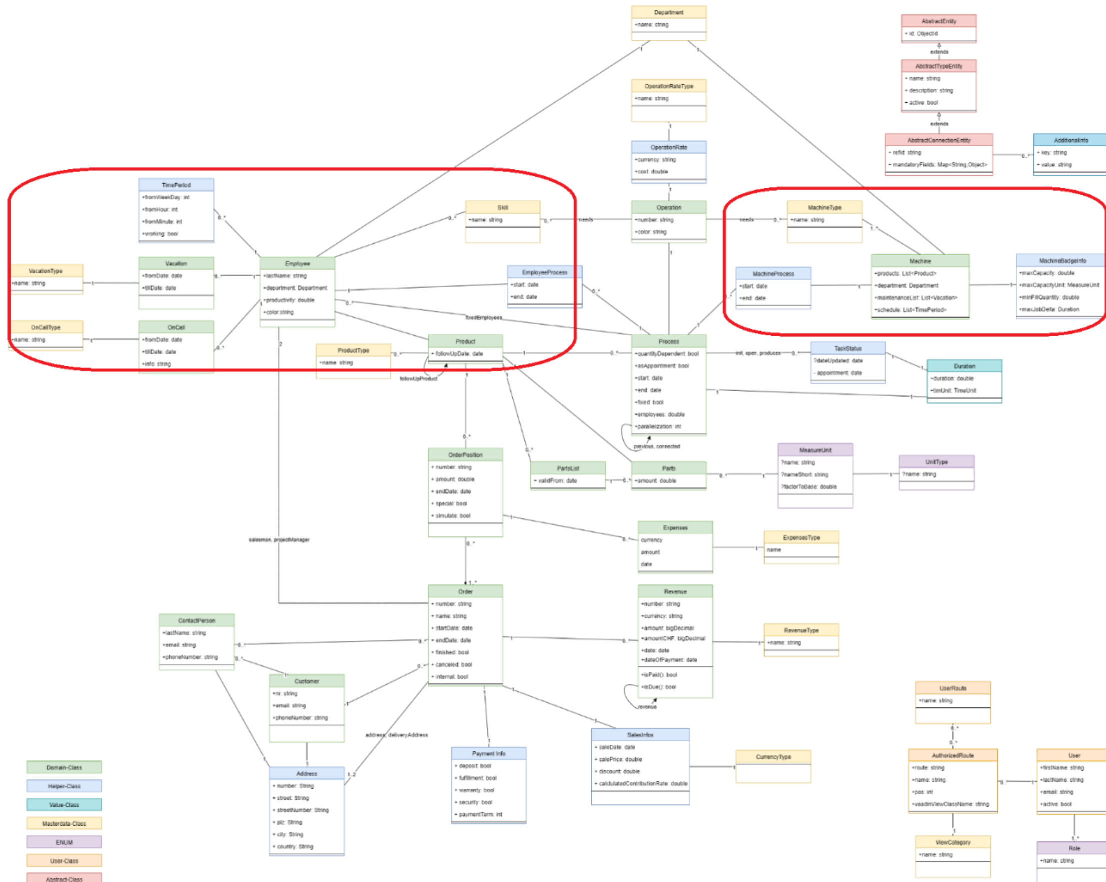

Abbildung 14: Ressourcen im IST-Domänenmodell

5.1.2 Durchgeführte Schritte

Zu Beginn wurde das Modul Ressourcen als eigenständige Einheit angelegt und mit den vorgesehenen Schichten *Domain, Application, Controller, Repository* und *Web* strukturiert. Anschliessend erfolgte die Migration der bestehenden Klassen aus der bisherigen Struktur: Die Entitäten wurden aus dem alten *Data*- bzw. *Domain*-Ordner in die neue *Domain*-Schicht verschoben, *Repository*-Klassen in die *Repository*-Schicht überführt und die bisherigen *Service*-Klassen der *Application*-Schicht zugeordnet.

Nachdem alle relevanten Klassen im Modul verortet waren, wurde die *Domain*-Schicht bereinigt: Persistenzspezifische Annotationen wie @Id wurden entfernt und die Klassen, soweit sinnvoll, als Records oder immutable Klassen umgesetzt. Ziel war eine vollständige Unabhängigkeit der *Domain* von externen Bibliotheken ausserhalb des Java-Basisumfangs.

Darauf aufbauend wurden aus den bisherigen Service-Klassen fachlich abgegrenzte Use Cases definiert. Die alten Service-Klassen wurden so angepasst, dass auch sie keine Abhängigkeiten zu Spring Boot oder anderen Frameworks mehr besitzen und ausschliesslich mit den Use-Case-Interfaces arbeiten.

Im nächsten Schritt folgte die Neugestaltung der Datenbankanbindung. In der *application.out*-Schicht wurden *Repository*-Interfaces definiert, die in der *Repository*-Schicht implementiert werden. Hierfür wurden DAO-Klassen erstellt sowie entsprechende Mapper-Funktionen zur Transformation zwischen *Domain*-Objekten und Datenbankmodellen entwickelt.

Anschliessend wurde die *Controller*-Schicht implementiert, die die definierten Use Cases aufruft und die notwendigen DTOs sowie zugehörige Mapper bereitstellt. Zum Abschluss wurde die *Web*-Schicht mit den Controllern verbunden und das Zusammenspiel aller Schichten getestet.

Da die *Application*-Schicht keine Framework-Annotationen mehr verwendet, wurde eine eigene Konfigurationsklasse (*RessourceConfig*) erstellt, die die notwendigen UseCases instanziiert und deren Abhängigkeiten zu den Repository-Implementierungen bereitstellt. Diese manuelle Orchestrierung stellt sicher, dass die Abhängigkeitsrichtung gewahrt bleibt und die *Application*-Schicht frameworkunabhängig bleibt.

Nach erfolgreicher Funktionsprüfung erfolgte die Implementierung von Unit-Tests für die *Domain*- und *Application*-Schicht unter Verwendung von JUnit in IntelliJ. Damit wurde eine erste automatisierte Testbasis für das Modul geschaffen.

Neben der grundsätzlichen Umstrukturierung wurde das Modul Ressourcen in mehrere Submodule gegliedert. Dadurch konnten fachliche Verantwortlichkeiten noch klarer abgegrenzt werden. Die Submodule umfassen *Employee, Machine* und *Tool* sowie ein gemeinsames *Common*-Submodul, in dem übergreifende Konzepte wie Abteilungen, Arbeitszeiten, Instandhaltung und das Ressourcentracking zusammengefasst sind. Diese Aufteilung trägt zur besseren Verständlichkeit bei und erleichtert die Weiterentwicklung einzelner Bereiche, ohne dass Änderungen ungewollte Auswirkungen auf andere Teile des Moduls haben.

```
@Test   Patrick Kehli *
void domain_should_not_depend_on_other_layers() {
    ArchRuleDefinition.classes() GivenClasses
            .that().resideInAPackage( s: ".." + MODULE_NAME + ".domain..") GivenClassesConjunction
            .should().onlyDependOnClassesThat() ClassesThat<ClassesShouldConjunction>
            .resideInAnyPackage(
                    ...strings: ".." + MODULE_NAME + ".domain..",
                    "..resources.common.domain..",
                    "ch.prosim.evopro.common.domain..",
                    "java..") ClassesShouldConjunction
            .check(classes);
}
```

Abbildung 15: ArchUnit-Test das die Domäne keine Abhängigkeiten kennt

Zur Sicherstellung der Einhaltung der Architekturprinzipien wurden drei ArchUnit-Tests eingeführt. Abbildung 15 zeigt den ArchUnit-Test der sicherstellt, dass die *Domain*-Schicht nur Abhängigkeiten zu sich selbst, zur Java-Basis-Bibliothek und zur *common.domain* und im Fall von Ressourcen zum *resources.common.domian* hat. Analog dazu gibt es einen Test für die *Application*-Schicht (Abbildung 16), der sicherstellt, dass keine Abhängigkeiten zu Klassen ausserhalb der *Application*-Schicht besteht ausser zur *Domain*-Schicht und den entsprechenden Common- und Sub-Common-Modulen.

```
@Test  Patrick Kehli *
void application_should_only_depend_on_domain() {
    noClasses() GivenClasses
            .that().resideInAPackage( s: ".." + MODULE_NAME + ".application..") GivenClassesConjunction
            .should().dependOnClassesThat() ClassesThat<ClassesShouldConjunction>
            .resideOutsideOfPackages(
                    ...strings: ".." + MODULE_NAME + ".application..",
                    "..resources.common.application..",
                    ".." + MODULE_NAME + ".domain..",
                    "..resources.common.domain..",
                    "ch.prosim.evopro.common..",
                    "java..") ClassesShouldConjunction
            .check(classes);
}
```

Abbildung 16: ArchUnit-Test für den Application-Layer

Der dritte ArchUnit-Test stellt sicher, das nur über die Interfaces im In- und Out-Ordner auf die Application-Schicht zugegriffen wird. Abbildung 17 zeigt das für das Resourcen-Modul. Hier ist auch die Ausnahme ersichtlich, dass die ResourcenConfig-Klasse für die Orchstrierung Zugriff auf die Application-Schicht benötigt.

```
@Test  Patrick Kehli *
void no_layer_should_access_application_except_in_and_out() {
    DescribedPredicate<JavaClass> forbiddenApplicationAccess =
            resideInAPackage( packageIdentifier: ".." + MODULE_NAME + ".application..")
                    .and(resideOutsideOfPackage( packageIdentifier: ".." + MODULE_NAME + ".application..in.."))
                    .and(resideOutsideOfPackage( packageIdentifier: ".." + MODULE_NAME + ".application..out.."));

    noClasses() GivenClasses
            .that().resideOutsideOfPackage( s: ".." + MODULE_NAME + ".application..") GivenClassesConjunction
            .and().doNotHaveSimpleName( s: "ResourceConfig")
            .should().dependOnClassesThat(forbiddenApplicationAccess) ClassesShouldConjunction
            .check(classes);
}
```

Abbildung 17: ArchUnit-Test für den Zugriff auf die Application-Schicht

Durch diese automatisierte Architekturvalidierung wird gewährleistet, dass die aufgestellten Regeln auch bei zukünftigen Erweiterungen eingehalten werden und die Struktur des Systems konsistent bleibt.

5.1.3 Technische Herausforderungen

Eine der grössten Herausforderungen bestand darin, sämtliche bestehenden Abhängigkeiten zu Frameworks wie Spring aus der *Domain*- und *Application*-Schicht

zu entfernen. Viele Klassen waren ursprünglich stark mit Infrastrukturkomponenten und anderen Modulen verflochten, sodass diese Abhängigkeiten zunächst identifiziert, aufgelöst und durch Schnittstellen ersetzt werden mussten. Um die Funktionalität während des Refactorings aufrechtzuerhalten, waren an mehreren Stellen temporäre Workarounds erforderlich, beispielsweise die schrittweise Entkopplung von Services und deren Übergang in klar definierte UseCases.

Auch die Einführung der *ResourceConfig*-Klasse zur manuellen Instanziierung der UseCases brachte zusätzlichen Aufwand mit sich, da alle benötigten Abhängigkeiten explizit verdrahtet werden mussten. Trotz des Mehraufwands bietet dieser Ansatz langfristig eine höhere Flexibilität und Unabhängigkeit von spezifischen Frameworks.

Ein weiterer signifikanter Aufwand entstand durch das wiederholte Mapping zwischen den verschiedenen Repräsentationen der Daten: vom DAO zur Domain, von der Domain zu DTOs und in umgekehrter Richtung. Dieser Prozess stellte sich als zeitintensiv und fehleranfällig heraus und führte während der Umsetzung mehrfach zu der Frage, ob dieser zusätzliche Aufwand durch die gewonnene Schichtentrennung tatsächlich gerechtfertigt ist. Hier zeigte sich deutlich, dass eine saubere Architektur zwar strukturelle Vorteile bringt, diese aber mit einem erhöhten Implementierungs- und Pflegeaufwand verbunden sind.

5.1.4 Ergebnis

Durch die Umstrukturierung des Moduls Ressourcen konnte eine klare Trennung der Schichten gemäss der in Kapitel 3 definierten Zielarchitektur umgesetzt werden. Die Domäne ist nun vollständig von Frameworks und externen Bibliotheken entkoppelt, und alle fachlichen Operationen werden über klar definierte UseCases abgewickelt. Die Datenbankanbindung erfolgt ausschliesslich über Repository-Interfaces in der *application.out*-Schicht, deren Implementierungen in der Repository-Schicht gekapselt sind.

Die zuvor bestehenden direkten Abhängigkeiten zu Produkten und Prozessen wurden entfernt, indem nur noch deren IDs referenziert werden. Dadurch sind Ressourcen nicht länger direkt mit Objekten anderer Module verknüpft, sondern lediglich über stabile Identifikatoren. Auf diese Weise verursachen Änderungen an den Ressourcen keine unbeabsichtigten Seiteneffekte mehr in der Planungsansicht oder in Auswertungen. Die neue Architektur, wie in Abbildung 18 ersichtlich, erleichtert es, Änderungen an der fachlichen Logik isoliert vorzunehmen und gezielt zu testen.

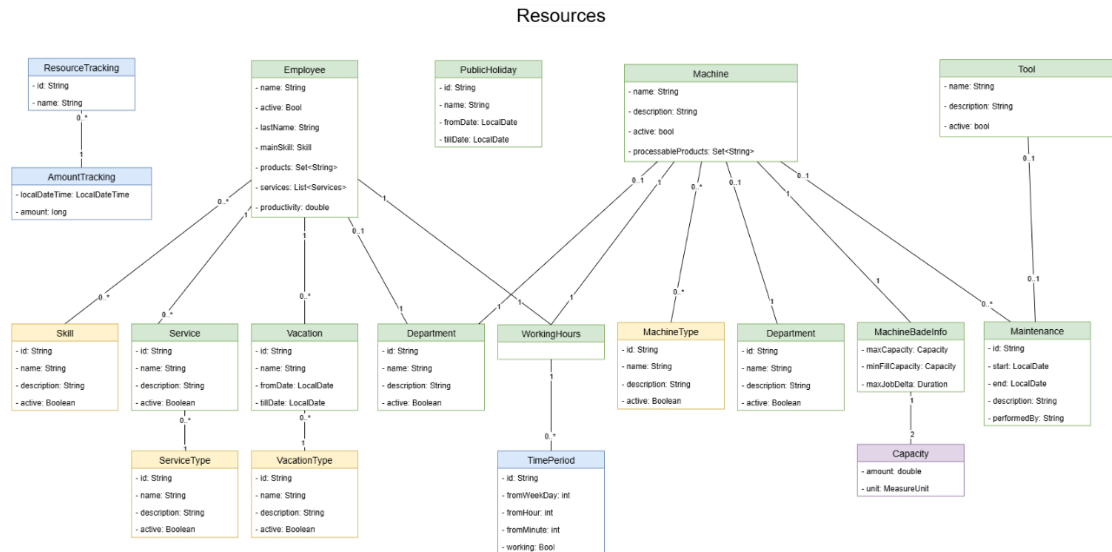

Abbildung 18: Neues Ressourcen Domain-Modell

Mit der Einführung von ArchUnit-Tests ist eine automatisierte Prüfung der Architekturprinzipien etabliert, die langfristig sicherstellt, dass die Schichtentrennung eingehalten wird. Zusätzlich wurden erstmals Unit-Tests für die *Domain*- und *Application*-Schicht erstellt, wodurch eine grundlegende Testbasis für künftige Anpassungen vorhanden ist.

5.1.5 Fazit

Mit der Umsetzung im Modul Ressourcen wurde ein wesentlicher Schritt in Richtung der angestrebten Soll-Architektur erreicht. Die klare Schichtentrennung, die Entkopplung von externen Bibliotheken und die Einführung von UseCases als zentrale fachliche Schnittstellen haben die Struktur deutlich verbessert und die Grundlage für eine nachhaltige Weiterentwicklung geschaffen.

Die Auflösung der direkten Abhängigkeiten zu Produkten und Prozessen reduziert potenzielle Seiteneffekte und erhöht die Stabilität bei Änderungen. Durch die etablierten ArchUnit-Tests und die neu eingeführte Testabdeckung ist eine dauerhafte Sicherung der Architektur- und Qualitätsziele gewährleistet.

Trotz des zusätzlichen Aufwands, insbesondere durch das umfangreiche Mapping zwischen DAO, Domain und DTO, überwiegen die Vorteile in Bezug auf Wartbarkeit, Testbarkeit und langfristige Flexibilität des Systems. Die in diesem Modul gesammelten Erfahrungen bilden eine wertvolle Grundlage für die Umsetzung der weiteren Module.

5.2 Modul Produkt

5.2.1 Ausgangslage

Das Modul Produkt bildet gemeinsam mit den Prozessen den fachlichen Kern von EVOPRO. Es verwaltet nicht nur Produktstammdaten, sondern definiert auch den zugrunde liegenden Produktionsprozess, der massgeblich die Ablaufplanung und

Auswertungen beeinflusst. Im bestehenden System war dieses Bereich stark mit anderen Klassen verknüpft, wie Abbildung 19 zeigt, und wurde in diversen UI-Ansichten verwendet, wie beispielweise der Produktionsplanung und den Auswertungen.

Abbildung 19: Produkt im IST-Domain-Modell

Zudem war die Entität Process, die die zentralen fachlichen Abläufe modellieren sollte, in der bisherigen Implementierung nur unzureichend strukturiert. Ihre Modellierung war weder fachlich klar abgegrenzt noch technisch sauber umgesetzt, was zu einer Vielzahl an Abhängigkeiten innerhalb und ausserhalb des Bereichs führte. Änderungen an diesen Klassen hatten daher oft weitreichende Auswirkungen auf andere Module und UI-Komponenten.

Durch die hohe Zahl an Querverbindungen und die fehlende Trennung der Verantwortlichkeiten lag die Komplexität deutlich höher als ursprünglich erwartet, was insbesondere die Umgestaltung der Process-Entität zu einer zeitintensiven Aufgabe machte.

5.2.2 Durchgeführte Schritte

Das Modul Produkt wurde nach dem gleichen Schema wie zuvor das Modul Ressourcen aufgebaut. Dazu wurde zunächst die Modulstruktur mit den Schichten *Domain, Application, Controller, Repository* und *Web* angelegt und die relevanten Klassen aus der bisherigen Codebasis in die entsprechenden Schichten verschoben.

Die Domain wurde neu strukturiert und um die fachlichen Entitäten wie Product, Process, ProcessStep, ProcessDetails, ProcessConnection und ProcessResources ergänzt. Dabei wurden die Beziehungen zwischen den Entitäten neu definiert und die Modellierung an die Zielarchitektur angepasst. Das neue Domain Model ist in Abbildung 20 dargestellt.

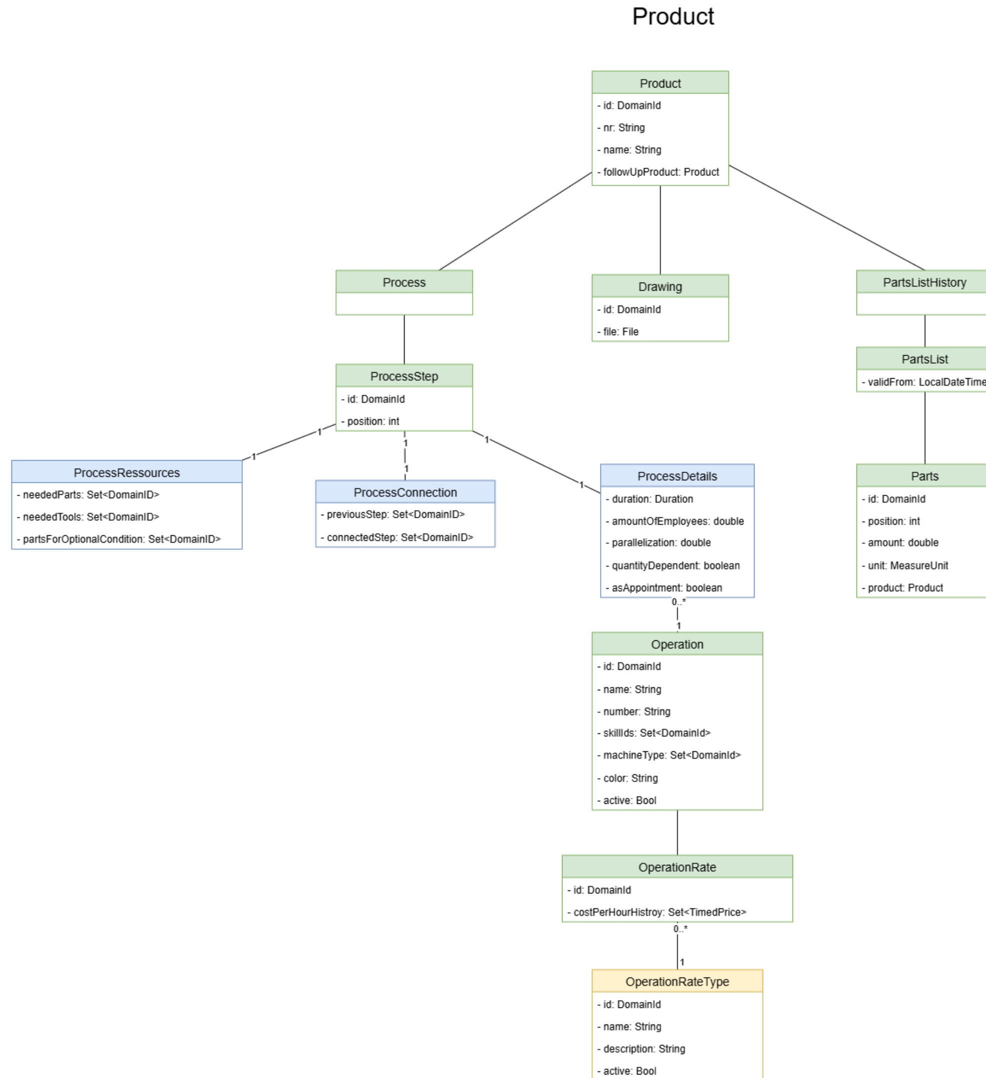

Abbildung 20: Neues Produkt Domain-Modell

Das im *Product*-Modul neu eingesetzte Architekturpattern wird in Abbildung 21 am Beispiel der ProductType-Entität dargestellt. Anstatt für jede Domänenklasse ein eigenes Set von UseCase-, Repository- und Controller-Implementierungen bereitzustellen, wurde ein generischer Ansatz gewählt, der sowohl die Datenbankanbindung als auch die Webschnittstelle umfasst.

Dazu wurden die generischen Interfaces CrudUseCase<T> und GenericRepository<T> definiert. Sie bilden die zentrale Schnittstelle für die Geschäftslogik bzw. die Datenpersistenz. Die abstrakte Klasse AbstractCrudService<T> implementiert das

Interface CrudUseCase<T> und nutzt gleichzeitig das GenericRepository<T>, wodurch eine einheitliche Grundlage für CRUD-Operationen geschaffen wird.

Da auch die konkrete Umsetzung dieser Standardoperationen häufig identisch ist, wurden weitere Abstraktionen eingeführt:

- **Persistenzschicht**: AbstractRepositoryImpl<DOMAIN, DAO> bündelt die generische Implementierung für den Datenbankzugriff.
- **Webschicht**: AbstractController<DOMAIN, DTO> stellt die generische Anbindung an die Webschnittstelle bereit.

Über spezialisierte Mapper (DtoMapperInterface, DaoMapperInterface) werden Domain-Objekte, DTOs und DAOs in Controller und Repository-Implementierungen zusammengeführt. Damit reduziert sich der Implementierungsaufwand für neue CRUD-basierte Entitäten erheblich: Im Regelfall müssen lediglich die Domain-, DTO- und DAO-Klassen sowie die zugehörigen Mapper erstellt werden. Service-, Repository- und Controller-Klassen enthalten in solchen Fällen keine eigene Logik mehr, sondern lediglich die Instanziierung der generischen Infrastruktur.

Die gewählte Lösung verfolgt zwei zentrale Ziele:

1. **Wiederverwendbarkeit durch Generik** – die CRUD-Funktionalität wird nur einmal bereitgestellt und kann über Typparameter beliebig auf neue Domänenobjekte angewendet werden.
2. **Reduktion von Abhängigkeiten** – die fachliche Logik bleibt klar von Infrastruktur- und Frameworkcode getrennt, sodass die Domäne langfristig robust, modular und erweiterbar bleibt.

Dieses Muster schafft eine Balance zwischen Generik und Entkopplung und bildet damit eine nachhaltige Grundlage für die Weiterentwicklung. Es erleichtert sowohl die Einführung neuer Entitäten als auch die Evolution bestehender Module und trägt wesentlich zur Wartbarkeit und Konsistenz des Gesamtsystems bei.

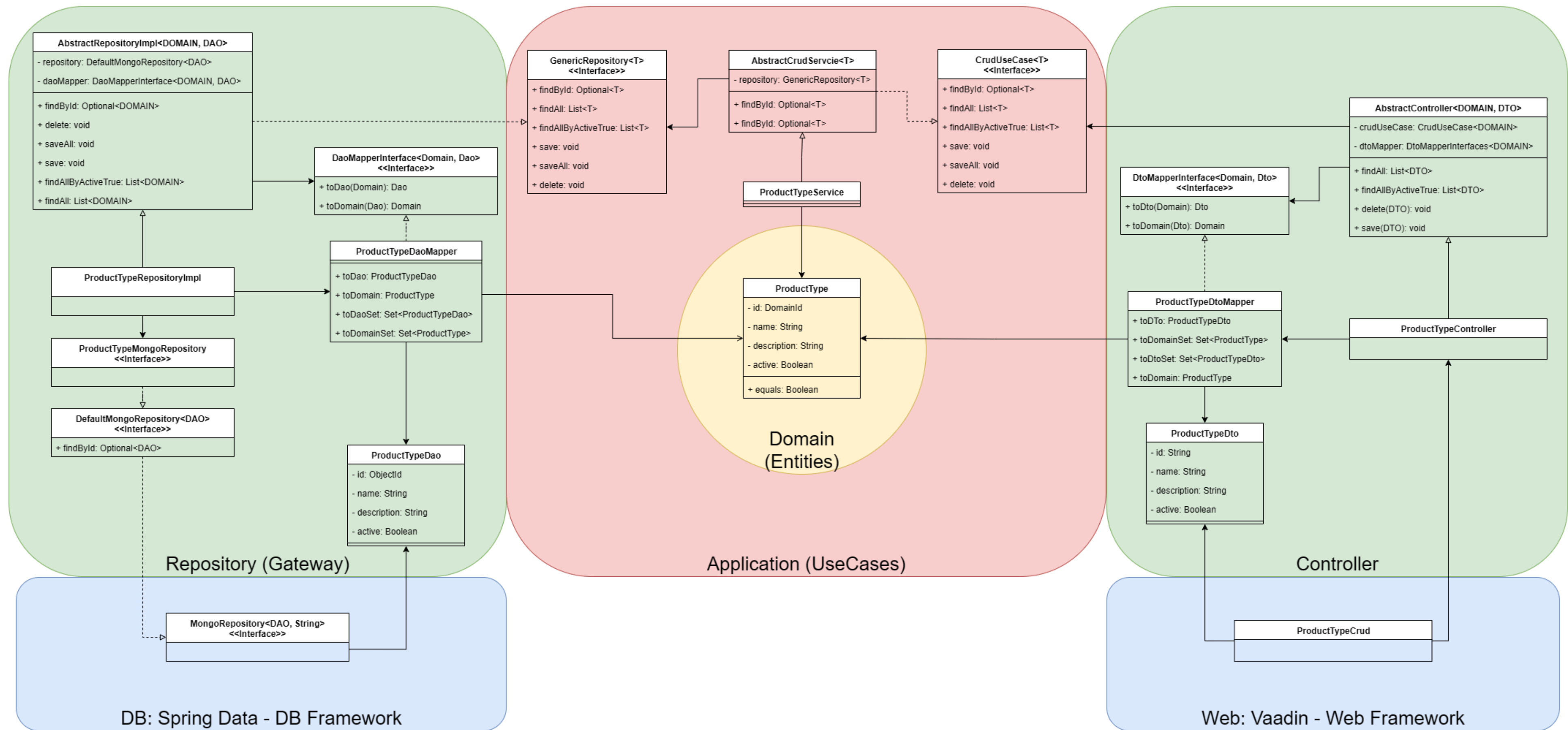

Abbildung 21: Generische Clean-Code-Architektur am Beispiel ProductType

Wie im Modul Ressourcen wurden auch hier nebst dem CrudUseCase weitere UseCases definiert, die die fachliche Logik kapseln und als öffentliche API des Moduls dienen. Ergänzend wurden Unit-Tests sowie die drei ArchUnit-Tests implementiert, um die Einhaltung der Architekturprinzipien automatisiert zu prüfen.

5.2.3 Technische Herausforderungen

Eine wesentliche Herausforderung im Modul Produkt war die Neugestaltung der Entität *Process* und die damit verbundene Auflösung zahlreicher Abhängigkeiten zu anderen Teilen des Systems. Die ursprüngliche Struktur war stark vernetzt und erforderte eine präzise Analyse, um fachliche Verantwortlichkeiten klar abzugrenzen und Beziehungen zwischen den beteiligten Entitäten neu zu ordnen.

Hinzu kam die Einführung des neu eingesetzten Architekturpatterns mit einem generischen Ansatz. Gerade in der Anfangsphase erwies sich dies als anspruchsvoll, da die abstraktere Modellierung mehr Konzeptionsarbeit erfordert als eine direkte, spezifische Implementierung. Es mussten geeignete Abstraktionen für UseCases, Services, Repositories und Controller definiert werden, die einerseits breit genug für die Wiederverwendung, andererseits schlank genug für die konkrete Umsetzung blieben. Dieser anfängliche Mehraufwand zahlt sich jedoch langfristig aus: Neue Entitäten können mit deutlich geringerem Implementierungsaufwand integriert werden, und die fachliche Logik bleibt klar von technischen Details entkoppelt.

5.2.4 Ergebnis

Mit der Umsetzung des Moduls Produkt konnte die Domäne klar strukturiert und von unnötigen Abhängigkeiten befreit werden. Die neu gestaltete Entität *Process* und ihre zugehörigen Strukturen sind nun fachlich und technisch sauber abgegrenzt, wodurch die Wartung und Weiterentwicklung deutlich vereinfacht wird.

Die Einführung des generischen Architekturpatterns ermöglicht eine einheitliche Handhabung von CRUD-Operationen und reduziert Code-Duplizierungen. Gleichzeitig bleibt die Implementierung flexibel und erweiterbar für verschiedene Produkttypen.

Besonders deutlich traten in diesem Modul die Vorteile der klar definierten UseCases zutage. Komplexe fachliche Abläufe wie die Erstellung eines Prozesses, das Kopieren eines Produkts oder die Übernahme eines bestehenden Prozesses lassen sich nun vollständig in UseCases kapseln. Dadurch konnte die Logik zentral gebündelt, von der UI entkoppelt und ein konsistentes Vorgehen über verschiedene Anwendungsfälle hinweg etabliert werden.

5.2.5 Fazit

Die Umsetzung des Moduls Produkt hat gezeigt, dass selbst hochvernetzte und komplexe fachliche Bereiche erfolgreich in die Zielarchitektur überführt werden können. Durch die Neugestaltung der Process-Struktur, die Einführung klar definierter

Modulgrenzen und den Einsatz generischer Repository-Implementierungen konnte eine deutlich höhere Wartbarkeit und Flexibilität erreicht werden.

Die konsequente Kapselung komplexer Abläufe in UseCases hat nicht nur die Trennung von Fachlogik und Präsentationsebene gestärkt, sondern auch eine klare, wiederverwendbare und testbare Struktur geschaffen. Die in diesem Modul gewonnenen Erkenntnisse – insbesondere im Umgang mit komplexen Abhängigkeiten – bilden eine wertvolle Grundlage für die Bearbeitung weiterer zentraler Module im System.

5.3 Modul Auftrag

5.3.1 Ausgangslage

Das Modul Order bildet mit dem Product das Herzstück der Anwendung, da es die Kundenaufträge verwaltet und damit Ausgangspunkt für nahezu alle weiteren Prozesse ist – von der Ressourcenplanung über die Produktionsprozesse bis hin zu Auswertungen. In der ursprünglichen Architektur war Order stark mit zahlreichen anderen Entitäten verknüpft, darunter Product, Process, Customer, PartsList und Employee wie in Abbildung 22 dargestellt ist. Diese enge Vernetzung führte dazu, dass Änderungen an der Auftragslogik häufig unbeabsichtigte Auswirkungen in anderen Bereichen nach sich zogen.

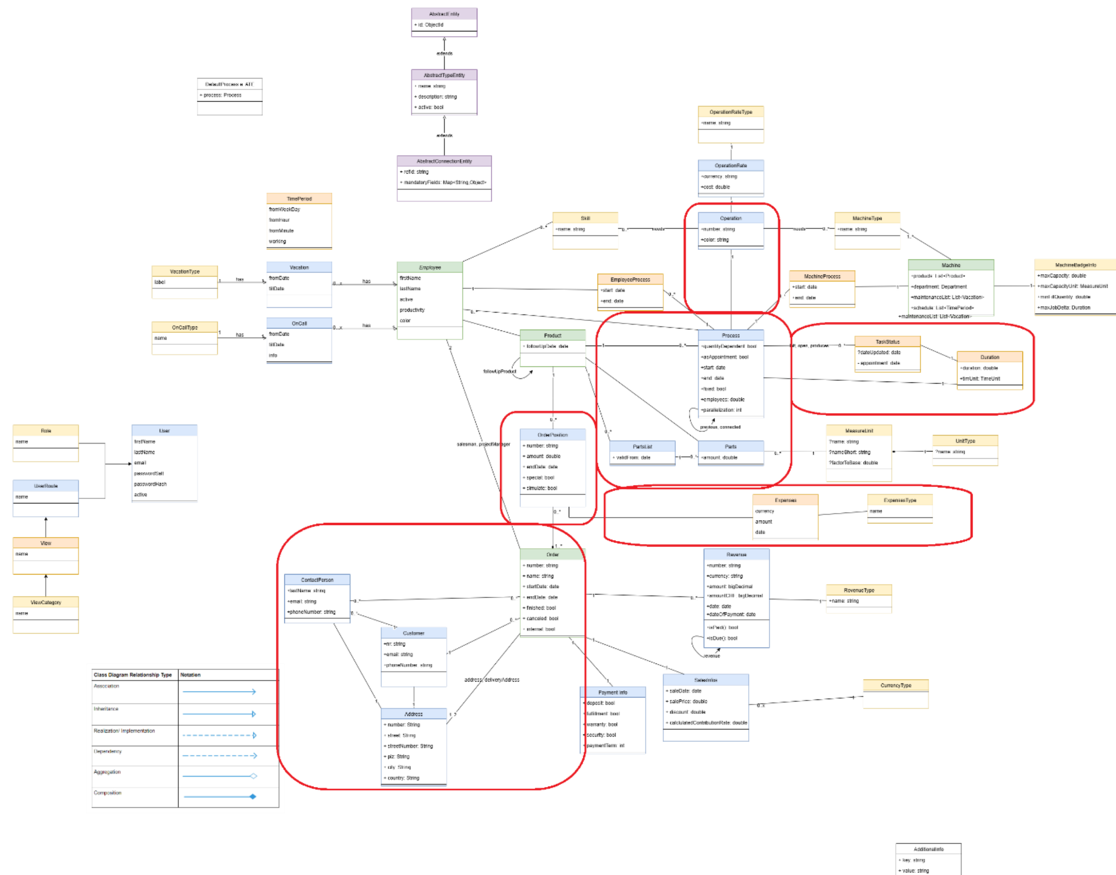

Abbildung 22: Order im IST-Domain-Modell

5.3.2 Durchgeführte Schritte

Bei der Umsetzung des Moduls Order wurde zunächst die Modulstruktur nach dem in den vorangehenden Kapiteln etablierten Muster aufgebaut. Die bisherigen Klassen und Entitäten wurden den Schichten *Domain, Application, Repository, Controller* und *Web* zugeordnet.

Ein wesentlicher Schwerpunkt lag auf der Neugestaltung der Prozesslogik. Anstelle der direkten Verknüpfung von Order und Process wurde eine neue Task-Struktur eingeführt. Auf Basis eines bestehenden Process lässt sich nun ein Ablauf (TaskList) erzeugen, der die für einen Auftrag relevanten Arbeitsschritte kapselt. Diese Entkopplung ermöglicht eine klare Trennung zwischen der produktbezogenen Prozessdefinition und der auftragsspezifischen Ausführung.

Zur Umsetzung wurden spezifische UseCases definiert, welche die Erstellung und Verwaltung von Aufträgen übernehmen. Ein zentraler UseCase erzeugt beispielsweise eine TaskList aus einem über den ProductController bereitgestellten ProcessDto. Die Geschäftslogik liegt damit vollständig im Application-Layer und ist unabhängig von UI oder Datenhaltung.

Das neue Domain-Modell (Abbildung 23) zeigt die überarbeitete Struktur mit den Entitäten Order, OrderPosition, TaskList und Task sowie deren Unterstrukturen wie TaskDetails, TaskConnection oder TaskStatus. Damit ist die bisherige enge Kopplung mit Process aufgelöst und durch eine eigenständige, auftragszentrierte Modellierung ersetzt.

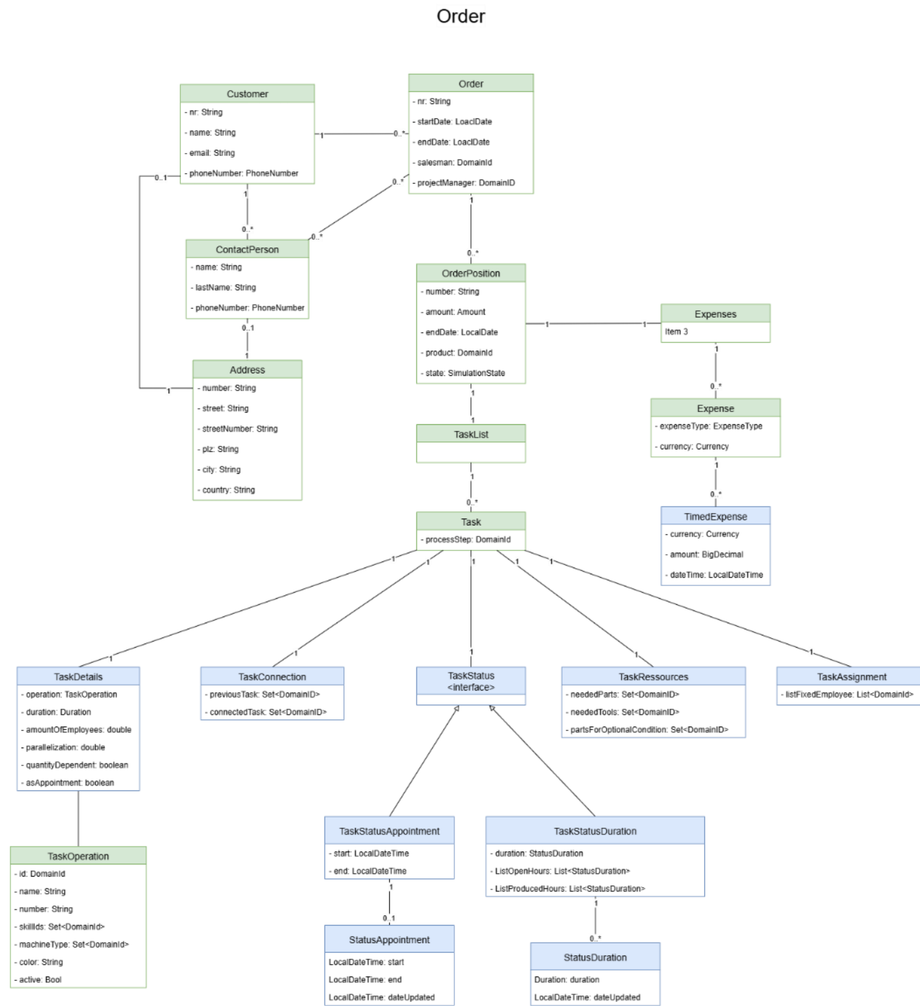

Abbildung 23: Neues Order Domain-Modell

5.3.3 Technische Herausforderungen

Die grösste Herausforderung bei der Umsetzung des Moduls Order bestand in der extremen Verflechtung mit anderen Teilen der Anwendung. In der ursprünglichen Architektur war die Auftragslogik eng an Produkte, Prozesse, Ressourcen und Auswertungen gekoppelt. Jede Änderung an der Order-Entität hatte daher potenziell weitreichende Seiteneffekte, was eine saubere Entkopplung besonders anspruchsvoll machte.

Ein weiterer schwieriger Punkt war die Neugestaltung der Prozesslogik. Da Process im ursprünglichen Modell zentral mit Aufträgen verknüpft war, musste eine Lösung gefunden werden, die einerseits die Verbindung zu produktdefinierten Prozessen erhält, andererseits aber eine eigenständige Auftragsabwicklung erlaubt. Dies wurde durch die Einführung der Task-Struktur erreicht. Die Definition, wie Tasks erzeugt, verbunden und mit Statusinformationen versehen werden, erforderte eine präzise Modellierung und eine klare Trennung der Verantwortlichkeiten.

5.3.4 Ergebnis

Mit der Umsetzung des Moduls Order konnte eine klare Entkopplung der Auftragslogik von den übrigen Bereichen der Anwendung erreicht werden. Durch die Einführung der neuen Task-Struktur wurde die bisher direkte Abhängigkeit von Process aufgelöst und in eine eigenständige Modellierung überführt. Damit lässt sich die auftragsbezogene Abwicklung unabhängig von der produktdefinierten Prozessbeschreibung steuern.

Die Abbildung zentraler Geschäftsabläufe in UseCases hat sich besonders in diesem Modul als vorteilhaft erwiesen. Komplexe Vorgänge wie die Generierung eines Auftragsablaufs aus einem Produktprozess oder die Verwaltung von Auftragspositionen lassen sich nun in klar abgegrenzten Schnittstellen abbilden. Dadurch ist die Fachlogik konsistent gebündelt, besser testbar und vom UI vollständig entkoppelt.

Die neue Struktur hat die Wartbarkeit deutlich verbessert und ermöglicht eine flexiblere Weiterentwicklung des Order-Moduls. Änderungen an Prozessen oder Produkten wirken sich nicht mehr unmittelbar auf die Auftragsebene aus, wodurch die Risiken von Seiteneffekten erheblich reduziert wurden.

5.3.5 Fazit

Das Modul Order stellt durch seine zentrale Rolle in der Anwendung besonders hohe Anforderungen an Konsistenz und Stabilität. Mit der Neugestaltung konnte gezeigt werden, dass selbst stark verflochtene Strukturen durch eine klare Domänenmodellierung und UseCase-zentrierte Architektur erfolgreich entkoppelt werden können. Die neu eingeführte Task-Logik bildet dabei eine robuste Grundlage für die Auftragsabwicklung und stärkt zugleich die Modularität des Gesamtsystems.

6 Ergebnisse und Fazit

Im Rahmen des Architektur-Refactorings von EVOPRO wurden verschiedene Module exemplarisch überarbeitet und anhand definierter Kriterien bewertet. Ziel war es, die in Kapitel 3 beschriebenen architektonischen Leitlinien und Qualitätsziele praktisch umzusetzen und deren Wirkung messbar nachzuweisen.

Die Ergebnisse stützen sich dabei sowohl auf automatisierte Analysen (SonarQube, ArchUnit-Tests, Testabdeckung) als auch auf funktionale Validierungen der refaktorierten Module. Neben der rein technischen Bewertung wurde besonderes Augenmerk daraufgelegt, wie gut die gesetzten architektonischen Ziele – Modularisierung, saubere Schichtentrennung und Use-Case-zentrierte Logik – in der Praxis umgesetzt werden konnten.

Im Folgenden werden die Ergebnisse in vier Schritten dargestellt: zunächst die Beurteilung der architektonischen Zielerreichung, anschliessend die Bewertung der Qualitätsziele, gefolgt von der Erfolgskontrolle anhand messbarer Kriterien und schliesslich einer Gesamtbewertung der Wirksamkeit des Refactorings.

6.1 Erreichung der Zielarchitektur

Ein zentrales Ziel des Refactorings war es, die ursprünglich stark gekoppelte monolithische Struktur von EVOPRO in eine klar gegliederte, modular aufgebaute Architektur zu überführen. Grundlage dafür bildeten die Prinzipien des Modularen Monolithen sowie der Clean Architecture.

Mit der Umsetzung der Module Ressourcen, Produkt und Auftrag konnte die geplante Trennung von Verantwortlichkeiten weitgehend erreicht werden. Jedes Modul verfügt nun über eine eigene Struktur mit Domain-, Application-, Repository-, Controller- und UI-Schicht, wodurch die fachliche Logik eindeutig vom technischen Rahmen abgegrenzt ist. Die Einführung von UseCases als zentrale Schnittstellen zur Fachlogik hat dabei wesentlich zur Entkopplung beigetragen: Fachliche Abläufe sind nun in klar definierten Operationen gebündelt, während UI und externe Systeme lediglich über diese Schnittstellen interagieren.

Besonders hervorzuheben ist die durchgehende Entkopplung der Domain- und Application-Schicht von externen Frameworks. Persistenz-Annotationen und Spring-Abhängigkeiten wurden aus der Fachlogik entfernt, wodurch eine langfristige Unabhängigkeit und Testbarkeit gewährleistet ist. Die Anwendung von ArchUnit-Tests stellt sicher, dass diese Trennung auch bei zukünftigen Anpassungen eingehalten wird.

Trotz der erzielten Fortschritte ist das architektonische Ziel noch nicht vollständig erreicht. Zentrale Bereiche wie die Planungsansicht, das Reporting sowie die ERP-Integration verbleiben bislang in der alten Struktur. Diese Module wurden im Rahmen dieser Arbeit grösstenteils durch Auskommentieren entkoppelt und sind daher aktuell nicht funktionsfähig. Diese Massnahme war notwendig, da der zeitliche Rahmen der

Masterarbeit begrenzt ist und die umfassende Neugestaltung der Module Order und Product einen erheblichen Aufwand beanspruchte.

Insgesamt zeigt sich jedoch, dass die gewählte Vorgehensweise tragfähig ist: Die exemplarisch refaktorierten Module belegen, dass das definierte Architekturziel umsetzbar ist und eine robuste Basis für die sukzessive Überführung des gesamten Systems in die Soll-Architektur bietet.

6.2 Erfüllung der Qualitätsziele

Die in Kapitel 3.3 definierten Qualitätsziele bildeten die Grundlage für die Bewertung des Refactorings. Ihre Erfüllung wurde anhand objektiver Metriken mittels SonarQube Test in Abbildung 24, automatisierter Architekturvalidierungen (ArchUnit) sowie gezielter Funktionstests überprüft. Im Folgenden werden die Ergebnisse pro Zielbereich dargestellt.

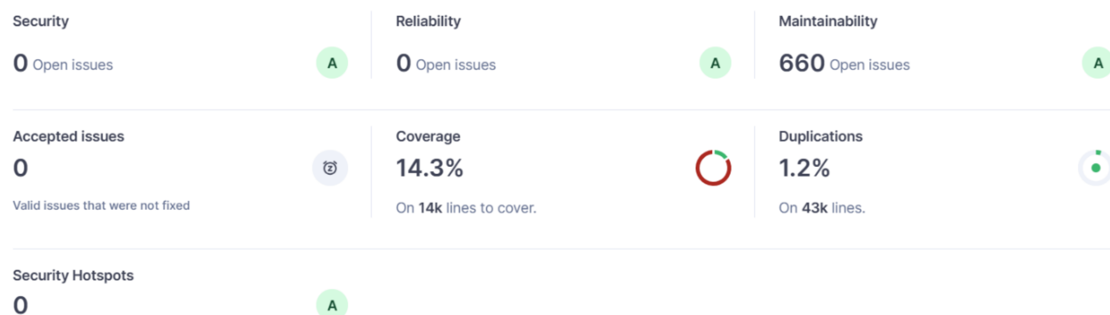

Abbildung 24: SonarQube-Test nach dem Refactoring der 3 Modulen

Architekturentkopplung

Durch die Einführung klar abgegrenzter Module (Ressourcen, Produkt, Auftrag) konnte die gewünschte Entkopplung erreicht werden. Direkte Querverbindungen zwischen den Modulen wurden aufgelöst, sodass die Fachlogik nur noch über UseCases angesprochen wird. ArchUnit-Tests validieren die Einhaltung der Modulgrenzen und bestätigen die Umsetzung der vorgesehenen Schichtentrennung.

Testabdeckung

Die Kernlogik der refaktorierten Module wurde vollständig durch Unit-Tests abgesichert (100 % Abdeckung der UseCases). Auf Gesamtsystemebene beträgt die Abdeckung nun 14.3 % – ein deutlicher Fortschritt gegenüber dem Ausgangszustand von 0 %, wenngleich der Ausbau auf weitere Module noch aussteht.

Code-Duplizierung

Die Duplizierungsrate konnte durch den Einsatz generischer Repository-Implementierungen und eine klare Trennung von Domain, DTOs und DAOs von 2,8 % auf 1,2 % reduziert werden. Damit wurde der definierte Zielwert von ≤ 1,4 % sogar unterschritten und eine deutliche Verbesserung erzielt.

Sicherheit und Zuverlässigkeit

Alle ursprünglich vorhandenen 11 Security Hotspots sowie 13 Reliability Issues wurden eliminiert. Die verbesserte Schichtentrennung und der Einsatz klarer Schnittstellen erleichtern zudem die langfristige Absicherung sicherheitskritischer Funktionen. SonarQube bestätigt den stabilen Zustand mit den Bewertungen A in Sicherheit und Maintainability.

Maintainability

Die Anzahl der Maintainability-Issues wurde von 757 auf 660 reduziert. Besonders wirksam erwies sich die klare Trennung von DTOs (für Serialisierung) und DAOs (für Persistenz), wodurch systematische Konflikte – etwa zwischen UI und Datenbanklogik – langfristig vermieden werden.

Performance

Die Ladezeiten konnten nur eingeschränkt bewertet werden. Da die Planungsansicht noch nicht refaktoriert wurde, blieben die ursprünglich bestehenden Performanceprobleme bestehen. Eine signifikante Verbesserung der ERP-abhängigen Ansichten steht damit noch aus und bleibt ein Thema für weitere Projektphasen.

Gesamtbewertung

Die Erfolgskontrolle zeigt, dass die wesentlichen Qualitätsziele – insbesondere Modularität, Testbarkeit und Wartbarkeit – nachweislich erreicht wurden. Teilerfolge wurden bei Code-Duplizierung und Maintainability erzielt, während die Performance-Optimierung noch aussteht. Insgesamt konnte damit eine solide Grundlage geschaffen werden, auf der die weitere Transformation des Gesamtsystems in die Soll-Architektur erfolgen kann.

7 Schlusswort

Mit dem in dieser Arbeit durchgeführten Architektur-Refactoring konnte gezeigt werden, dass selbst eine über Jahre gewachsene, stark gekoppelte Software erfolgreich in eine klar strukturierte und langfristig wartbare Architektur überführt werden kann. Am Beispiel der Module Ressourcen, Produkt und Order wurde demonstriert, wie durch Modularisierung, horizontale Schichtung und die konsequente Abbildung fachlicher Abläufe in Use Cases eine robuste Basis für die Weiterentwicklung geschaffen werden kann.

Die mit SonarQube durchgeführten Analysen bestätigen die erzielten Verbesserungen: Code-Duplizierungen und Maintainability-Issues wurden reduziert, Security-Hotspots beseitigt und in den bearbeiteten Modulen eine vollständige Testabdeckung der Use Cases erzielt. Damit sind die in Kapitel 3 definierten Qualitätsziele weitgehend erfüllt und die Wirksamkeit der Massnahmen objektiv nachweisbar.

Zugleich wurde deutlich, dass ein Refactoring dieser Grössenordnung im laufenden Betrieb eine besondere Herausforderung darstellt. Der notwendige Fokus auf die Architekturmodernisierung führte zeitweise dazu, dass neue Features nicht parallel integriert werden konnten. Diese Erfahrung unterstreicht die Schwierigkeit, Produktweiterentwicklung und strukturelle Verbesserungen gleichzeitig umzusetzen.

Insgesamt bietet das Refactoring eine stabile Grundlage für die Weiterentwicklung von EVOPRO. Künftige Arbeiten können darauf aufbauen, indem die neuen Architekturprinzipien sukzessive auf weitere Module übertragen, die Testabdeckung systematisch ausgebaut und ergänzende Massnahmen wie Monitoring und Performance-Optimierung vorangetrieben werden. Damit ist ein wichtiger Schritt in Richtung einer nachhaltigen, erweiterbaren und qualitativ hochwertigen Softwarearchitektur getan.

8 Literaturverzeichnis

Brown, S. (2016). *Software Architecture for Developers Vol. 2.*

Brown, S. (2018). *Modular Monoliths.* GOTO 2018.

Evans, E. (2003). *Domain-Driven Design.*

Martin, R. C. (2018). *Clean Architecture. A Craftsman's Guide to Software Structure and Design.*

9 Abbildungsverzeichnis

Abbildung 1: Interesse-Einfluss Matrix der Stakeholdergruppen 12
Abbildung 2: EVOPRO Systemkontext 14
Abbildung 3: EVOPRO Container-Diagramm 15
Abbildung 4: EVOPRO Ist-Komponentenmodell 17
Abbildung 5: EVOPRO Ist-Klassenmodell 19
Abbildung 6: Server Response Time der aktuelle EVOPRO Applikation 21
Abbildung 7: Übersicht der SonarQube Qualitäts Analyse 22
Abbildung 8: Maintainability Overview der SonarQube Qualitätsanalyse 22
Abbildung 9: Reliability Overview der Sonar Qube Qualitäts Analyse 23
Abbildung 10: EVOPRO Soll-Komponentenmodell 25
Abbildung 11: Vergleich Ist- und Soll-Architektur 27
Abbildung 12: Umsetzungszeitplan Refactoring EVOPRO 31
Abbildung 13: Ausschnitt des Kanban-Boards in Jira 32
Abbildung 14: Ressourcen im IST-Domänenmodell 36
Abbildung 15: ArchUnit-Test das die Domäne keine Abhängigkeiten kennt 37
Abbildung 16: ArchUnit-Test für den Application-Layer 38
Abbildung 17: ArchUnit-Test für den Zugriff auf die Application-Schicht 38
Abbildung 18: Neues Ressourcen Domain-Modell 40
Abbildung 19: Produkt im IST-Domain-Modell 41
Abbildung 20: Neues Produkt Domain-Modell 42
Abbildung 21: Generische Clean-Code-Architektur am Beispiel ProductType 44
Abbildung 22: Order im IST-Domain-Modell 46
Abbildung 23: Neues Order Domain-Modell 48
Abbildung 24: SonarQube-Test nach dem Refactoring der 3 Modulen 51

10 Tabellenverzeichnis

Tabelle 1: Ziele nach Stakeholdergruppe 12
Tabelle 2:Vergleich der zwei Systeme 21