Programmentwurf - ASWE

Backend für eine Pflanzenpflege-Management-Plattform

Studienrichtung Medizinische Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

Studenten	Kira Isenberg, Fabian Hagenmeier
Matrikelnummern	3427985, 2207087
Kurs	TINF22B1
Abgabedatum	31.05.2025
Dozent	Daniel Lindner

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Die Idee zu diesem Projekt entstand aus einem ganz persönlichen Bedürfnis: Kira möchte langfristig ein System entwickeln, mit dem sie die Pflanzen in ihrem hydroponischen Garten verwalten kann. Statt sich alleine durch ein großes Softwareprojekt zu kämpfen, haben wir die Gelegenheit genutzt, diese Idee mit dem Programmentwurf zu verbinden – und damit die Grundlage für eine spätere, vollständige Anwendung geschaffen.

Unser Ziel war es nicht, eine fertige App abzuliefern, sondern ein robustes, erweiterbares Backend zu entwickeln, das zentrale Funktionen wie die Verwaltung von Pflanzen, Pflegeplänen und Pflegeaufgaben unterstützt. Dabei haben wir besonderen Wert auf eine saubere Strukturierung nach den Prinzipien der Clean Architecture, testbare Komponenten und sinnvolle Domain-Modelle gelegt – alles mit dem Ziel, das System später problemlos erweitern zu können.

Der Abschnitt zu Unit Tests soll doppelt gewertet werden.

Unsere Codebasis ist unter https://github.com/kira-isi/plant-care-backend öffentlich zugänglich.

Kapitel 2

Clean Architecture

Bei der Einrichtung des Projekts haben wir besonderen Wert auf die saubere Trennung der Schichten entsprechend der Clean Architecture gelegt. Jede Schicht ist als eigenes Projekt im Solution-Explorer organisiert – beginnend bei 3_Domain, über 2_Application und 1_Adapters bis hin zu Plugins. Dadurch wird bereits auf Dateiebene erzwungen, dass die Abhängigkeiten nur „nach innen“ zeigen.

Die Domain-Schicht referenziert keine anderen Projekte. Die Application-Schicht referenziert ausschließlich 3_Domain, und 1_Adapters kennt sowohl 2_Application als auch 3_Domain. Das Testprojekt und die Plugin-Schicht kennen alle anderen Projekte. Ein Rückgriff auf innere Schichten von außen ist damit hoffentlich technisch ausgeschlossen. Damit soll garantiert werden, dass die Dependency Rule konsequent eingehalten wird.

Für die Verbindung der einzelnen Schichten verwenden wir Dependency Injection. Alle Use Cases und Services erhalten ihre Abhängigkeiten wie Repositories über Interfaces, die in der Application-Schicht definiert sind. Die konkreten Implementierungen werden in der Plugin-Schicht (Program.cs) registriert. So bleiben alle inneren Schichten unabhängig von konkreten Technologien und jederzeit testbar.

2.1 Domain

Die Domain-Schicht bildet das Herzstück der Anwendung und enthält ausschließlich fachliche Modelle und Logik. Hier werden zentrale Begriffe wie Pflanze, Pflegeplan und

Pflegeaufgabe als Entitäten oder Value Objects modelliert. Ziel dieser Schicht ist es, unabhängig von Technologien und Implementierungsdetails zu bleiben.

In unserer Domain haben wir unter anderem folgende Konzepte definiert:

- Plant und PlantType
- Careplan
- CareTasks in Form von RecurringCareTask, ScheduledCareTask mit entsprechenden CareTypes und CareTaskDetails wie WateringDetails, FertilizingDetails
- Location

Zusätzlich wurden in dieser Klasse weitere grundlegende Klassen wie Exeptions und eine CareTaskFactory erstellt. Die nach näherer Betrachtung vermutlich in die Application-Schicht ausgelagert werden könnten.

Die grundlegendsten Funktionen und Logiken der hier definierten Klassen wird ebenfalls erstellt z. B. das Berechnen von IsDue() oder IsOverdue() bei wiederkehrenden Aufgaben.

Die Domain kennt weder Datenbanken noch Schnittstellen – sie ist vollständig unabhängig und dadurch leicht testbar und wiederverwendbar.

2.2 Application

Die Application-Schicht stellt das Bindeglied zwischen der fachlichen Logik (Domain) und der Außenwelt dar. Unser Fokus lag darauf, hier klar abgegrenzte Anwendungsfälle („Use Cases“) zu definieren, die konkrete Abläufe kapseln – wie etwa das Anlegen einer Pflanze oder das Durchführen einer wiederkehrenden Pflegeaufgabe. Alle Abläufe, die ein Benutzer typischerweise in einer Anwendung auslöst, sollten hier als eigenständige Klassen modelliert werden.

Wir haben uns bewusst gegen ein generisches Service-Konzept entschieden und stattdessen die Use-Case-Klassen als jeweils zuständig für genau einen Anwendungsfall konzipiert. Das erhöht zwar die Anzahl der Klassen, sorgt aber für eine klare Trennung der Verantwortlichkeiten. Eine Herausforderung bestand darin, die Zuständigkeiten

sauber aufzuteilen. Wir haben komplexe Logik bewusst in die Adapter-Schicht verlagert und sämtliche Ablaufentscheidungen aus den Use-Cases ferngehalten. So werden z. B. Benachrichtigungen bei überfälligen Aufgaben nicht automatisch verschickt, sondern über den Use Case GetDueTasks lediglich zur Verfügung gestellt. Das macht die Anwendung flexibler – etwa für geplante Erweiterungen wie eine mobile App oder eine Integration mit Sensoren.

Für die technische Entkopplung der Schicht setzen wir konsequent auf Interfaces für alle externen Abhängigkeiten. So verwendet z. B. CreatePlant das IPlantRepository, ohne zu wissen, ob die Daten später aus einer SQLite-Datenbank, einem Webservice oder einer JSON-Datei stammen. Diese Interfaces werden über Dependency Injection bereitgestellt, was die Testbarkeit erhöht und zukünftige Änderungen an der Persistenzschicht erleichtert. Diese Interfaces werden in der Application-Schicht definiert.

2.3 Adapter

Die Adapter-Schicht dient in unserer Architektur als technische Brücke zwischen der Application-Schicht und der Außenwelt. Hier werden beispielsweise Daten aus einer Datenbank geladen und in Form gebracht, sodass sie von der Application-Schicht genutzt werden können. Hier würde typischerweise ebenfalls die bereitstellung von Endpunkten einer API stattfinden. Allerdings lag in unserem Projekt der Fokus auf der Backend-seitigen Vorbereitung.

Die Implementierung von HTTP-Controllern – die in Clean Architecture ebenfalls in die Adapter-Schicht gehören – haben wir bewusst zurückgestellt. Zwar hätten wir mit .NET eine API-Anbindung relativ einfach realisieren können, jedoch hätten wir dafür zusätzliche Framework-Konventionen und externe Abhängigkeiten berücksichtigen müssen.

Stattdessen haben wir uns auf zwei Dinge konzentriert:

- Ressourcen und Mapper: Wir haben erste Datenstrukturen entworfen, mit denen sich z. B. Pflegeaufgaben oder Pflanzen für eine spätere API-Ausgabe strukturieren lassen. Dazu gehören einfache DTOs sowie Mapper, mit denen sich Domain-Modelle in transportfreundliche Objekte überführen lassen.
- Repository-Implementierungen: Ein zentraler Bestandteil dieser Schicht sind unse-

re konkreten Implementierungen der Repository-Interfaces. So enthält die Adapter-Schicht beispielsweise die Klassen SqlPlantRepository, SqlCarePlanRepository und SqlLocationRepository, die den Datenzugriff über Dapper und SQLite kapseln. Die Domain- und Application-Schicht bleiben dadurch vollständig unabhängig von der verwendeten Datenbanktechnologie.

Durch diese Trennung bleibt unser System modular: Die Adapter-Schicht kann in einem nächsten Schritt um Controller oder ViewModel-Adapter ergänzt werden, ohne dass Änderungen an der Fach- oder Anwendungslogik nötig sind.

2.4 Plugin

Die Plugin-Schicht enthält die konkreten externen Abhängigkeiten, die zum Starten und Ausführen der Anwendung nötig sind. In unserem Projekt umfasst sie zwei zentrale Bestandteile:

- Datenbankkonfiguration: Hier wird die SQLite-Verbindung eingerichtet. Die Adapter-Schicht nutzt Dapper, die Plugin-Schicht stellt dafür die konkrete Datenquelle bereit.
- Programmstart (Program.cs): In dieser Datei werden die Abhängigkeiten per Dependency Injection registriert und die Anwendung gestartet. Damit verbindet die Plugin-Schicht alle Komponenten zu einem lauffähigen System.

Wir haben darauf geachtet, alle externen Technologien klar auf diese Schicht zu beschränken. So bleibt der Rest der Anwendung unabhängig und leicht erweiterbar – z. B. für andere Datenbanken, Logging oder eine API-Anbindung.

Kapitel 3

Unit-Tests

3.1 Ziel und Aufbau der Tests

Ziel unserer Tests war nicht die vollständige Abdeckung aller Programmteile, sondern das exemplarische Aufzeigen, wie Unit-Tests im Rahmen einer sauberen Architektur sinnvoll eingesetzt werden können. Im Mittelpunkt standen daher ausgewählte, fachlich relevante Methoden und Use Cases.

Wir haben uns darauf konzentriert, die Logik in der Domain- und Application-Schicht zu testen. Die Tests wurden mit xUnit geschrieben und folgen durchgehend dem Arrange-Act-Assert-Muster.

Insgesamt wurden 11 Unit-Tests erstellt. Diese decken zentrale Bereiche der Domain- und Application-Schicht ab:

- CareTaskTests: Fälligkeitsprüfung bei wiederkehrenden Aufgaben
- LocationTests: Validierung und Anlage von Standorten
- PlantTests: Erstellung und Aktualisierung von Pflanzen

Von 11 Tests waren 8 auf anhieb erfolgreich. Debei viel auf das besonders fehlerhafte Eingaben wie leere Strings in usecases und Konstrucktoren nicht beachtet wurden.

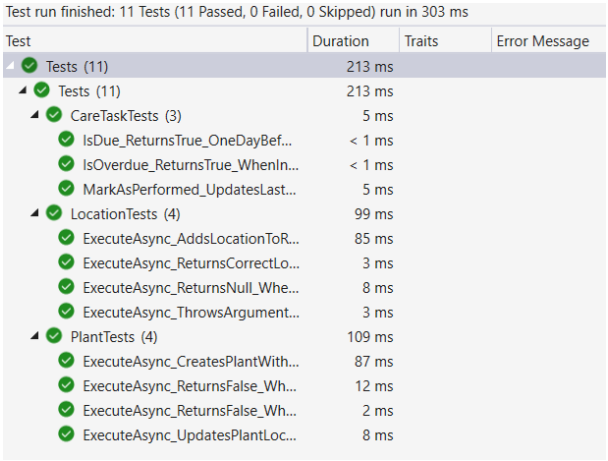

Abbildung 3.1: Erfolgreicher Test-Durchlauf

3.2 Einsatz von Mocks

Um Use Cases isoliert zu testen, kamen Mocks zum Einsatz – erstellt mit dem Framework Moq. So konnten Repository-Interfaces ersetzt und typische Szenarien simuliert werden wie etwa eine Abfrage, die ein bestimmtes Domain-Objekt zurückliefert oder die Prüfung, ob ein AddAsync-Aufruf mit den erwarteten Daten erfolgt. Die Mocks ermöglichen saubere Unit-Tests ohne externe Infrastruktur.

Die Implementierungen der Repositories selbst wurden nicht getestet, da diese als Integrationstests zu werten wären und nicht zum Fokus eines Programmentwurfs im Sinne der Clean Architecture gehören.

3.3 Teststrategie und Abdeckung

Die Testabdeckung wurde mit Coverlet gemessen. Der Gesamtwert beträgt 15,3 %, was angesichts der gezielten Auswahl und des prototypischen Charakters des Projekts erwartbar war. Es wurden nur sehr wenige kleine Aspekte der Anwendung getestet. Komplexe Controller sind nicht implementiert und konnten daher auch nicht getestet werden.

Viele Zeilen entfallen zudem auf DTOs, Konfigurationen oder technische Infrastruktur (z. B. Datenbankadapter).

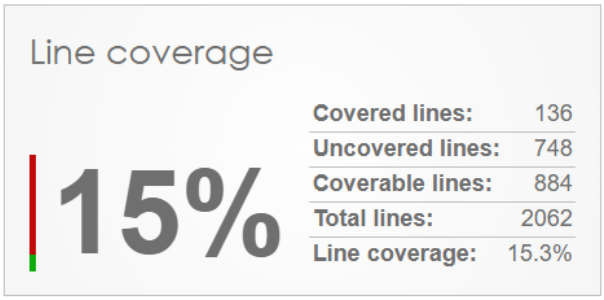

Abbildung 3.2: Coverrage durch Unit-Tests

Die Tests dienten dabei auch zur Reflektion der Architektur: Wiederverwendbarkeit, Testbarkeit und Trennung von Zuständigkeiten. Sie helfen den Entwicklungsprozess aktiv zu hinterfragten und zu verbessern.

3.4 ATRIP-Analyse

Unsere Tests erfüllen die ATRIP-Kriterien im Rahmen des Programmentwurfs in sinnvoller Weise.

- Automatic: Alle Tests sind vollständig automatisiert mit xUnit eingebunden. Sie lassen sich ohne manuelle Schritte per dotnet test ausführen. Auch die Code Coverage wurde über Coverlet integriert, was den automatisierten Testprozess sinnvoll ergänzt.
- Thorough: Wir haben nicht alle relevanten Funktionen getestet – insbesondere Randfälle, Validierungslogik und Fehlerbehandlung wurden größtenteils ausgespart. Das war allerdings eine bewusste Entscheidung. Stattdessen haben wir exemplarisch zentrale fachliche Methoden getestet, z. B. die Fälligkeitsprüfung bei Pflegeaufgaben oder das Erstellen und Aktualisieren von Pflanzen. Die Tests zeigen, dass unser Code testbar aufgebaut ist, auch wenn nicht alle Fälle abgedeckt sind.
- Repeatable: Alle Tests laufen stabil und liefern reproduzierbare Ergebnisse. Es gibt keine externe Abhängigkeit (z. B. Datenbank), weil alle Infrastrukturkomponenten gemockt wurden. Ein Beispiel ist CreatePlant, bei dem das Repository vollständig isoliert und das Verhalten überprüfbar ist.

- Independent: Unsere Tests sind voneinander unabhängig. Jeder Test initialisiert seine eigene Datenbasis, es gibt keine gemeinsame Setup-Logik oder globale Zustände. Das gilt sowohl für einfache Domain-Methoden als auch für die Use Cases.
- Professional: Die Tests sind klar benannt und strukturiert. Wir folgen durchgängig dem AAA-Muster.

```
[Fact]
public async Task ExecuteAsync_UpdatesPlantLocation()
{
    // Arrange
    var originalLocation = Guid.NewGuid();
    var newLocation = new Location("Balkon", "Ostseite");
    var plant = new Plant(Guid.NewGuid(), originalLocation, "Umzugspflanze");

    var repo = new Mock<IPlantRepository>();
    repo.Setup(r => r.GetByIdAsync(plant.Id)).ReturnsAsync(plant);
    var useCase = new RelocatePlant(repo.Object);

    // Act
    var result = await useCase.ExecuteAsync(plant.Id, newLocation);

    // Assert
    Assert.True(result);
    Assert.Equal(newLocation.Id, plant.LocationId);
    repo.Verify(r => r.UpdateAsync(plant), Times.Once);
}
```

Abbildung 3.3: Beispiel für einen Test

Kapitel 4

Domain Driven Design

Unsere Domäne ist die Pflanzenpflege, ein Bereich, der nicht unbedingt gut professionell durchorganisiert ist. Und trotzdem enthält diese Domäne Begriffe und Aktionen, die sich durch die Gesamtheit der Tätigkeit ziehen. Für dieses Projekt wurde es vorgezogen, die Domänensprache im Code auf Englisch anzuwenden, deshalb würd für jeden deutschen Domänenbegriff die im code verwendete englische Übersetzung in Klammern zugefügt.

4.1 Entitäten

Ganz im Vordergrund stehen hier 3 Dinge, die Pflanze selbst, die Pflegeaktionen und die Organisation dieser in einem Plan.

Pflanzen sind das Hauptobjekt der Domäne. Pflanzen sind meist durch Ihren Artennamen spezifiziert. Arten teilen bestimmte Merkmale miteinander, wie zum Beispiel wann und wie viel Wasser oder Sonne sie benötigen. Bewässerung kann manuell als Aktion durchgeführt werden. Das nötige Sonnenlicht erhalten die Pflanzen meist durch ihre Platzierung an sonnigeren oder schattigeren Bereichen.

Es gibt verschiedene Typen von Aktionen. Einmalige Tasks werden einmalig geplant und durchgeführt, wiederkehrende Aufgaben werden in einem gegebenen Zeitintervall wiederholt. Daraus ergeben sich zwei weitere Entitäten für die Domäne, die die Pflegeaktion spezifizieren.

Daraus Ergeben sich die Hauptsächlichen Entitäten, die in der Domäne abgebildet werden sollen:

- Pflanze und Pflanzentyp (Plant und PlantType)
- Platzierung (Location)
- Pflegeaktion (CareTask)
- Einmalige, geplante Aktion (ScheduledTask)
- Wiederkehrende Aufgabe (RecurringTask)

4.2 Value Objects

Pflegeaktionen kommen in unterschiedlichen Typen, die sich allerdings in ihrer Art auch bei Anwendung auf verschiedene Pflanzentypen eigentlich nicht ändern. Deshalb werden die einzelnen Pflegeaktionen als ValueObjects in der Domäne angesehen. Sie besitzen durch ihren Typen eine inhärente Wertigkeit für den Anwender, sind allerdings im Code alle gleich zu behandeln. Sie benötigen keine besonderen Methoden oder Attribute, ihr größtes Unterscheidungsmerkmal ist der Typ selbst. Pflegeaktionen, die als Value Objects angelegt wurden, beinhalten:

- Bewässerung (Watering)
- Düngen (Fertilizing)
- Umtopfen (Repotting)
- Zurückschneiden oder Stutzen (Pruning)

Des weiteren existieren in der Domäne noch die Möglichkeit bestimmten Value Objects noch Details anzuhaften. So muss zum Beispiel beim Bewässern klar sein, wie viel Wasser gegeben werden soll, oder welcher Dünger in welcher Menge angewandt werden soll. Dafür wurden die careTaskDetails eingeführt.

4.3 Aggregate

Ein klassisches Beispiel für Aggregate in dieser Domain Bildes der Pflegeplan. Nach der Initialisierung ist es im Pflegeplan möglich, sowohl Pflanzen als auch Aktionen hinzuzu-

fügen und zu entfernen, alle zugewiesenen Aktionen oder Pflanzen zurückzusetzen, sowie alle auszuführenden Aktionen auf einen Blick erkennen zu können. Damit erfüllt der Pflegeplan auch die Anforderungen von CRUD.

4.4 Repositories

In dieser Domäne stellen Repositories eine Sammlung aus entweder Aggregaten oder deren Komponenten da, aus denen über IDs Zugriff gewährt wird. Auch für Locations, die nicht direkt Teil eines Aggregates sind, gibt es ein Interface. Daher gibt es im Code vorbereitete Repositories für Pflanzen, Aktionen und Pflegepläne. Sie werden hauptsächlich verwendet, um Domain Services eine Sammlung an Objekten zu geben, mit denen diese Weiterarbeiten können. Dabei wurden sie nur als Interfaces definiert, um nicht die Generierung und Speicherung nach Außen geben zu können (Stichwort Dependency Injection).

Die Repositories erben alle von der Parent Interface IGenericRepository, die die Methoden zum Besorgen von Objekten per ID, das Hinzufügen, Updaten und Löschen vorschreiben.

Für den CarePlan werden zursätzlich noch Methoden vorgeschrieben, die es ermöglichen alle Plane mit Aktionen zu erhalten, die entweder anstehen oder bereits überfällig sind. Für Pflanzen wird vorgeschrieben eine Methode zu implementieren, die es ermöglicht Pflanzen auch per Typ zu finden. Da es sich dabei um keine eindeutige Zuweisung handelt, wird der Return Type der Methode als IEnumerable vorgeschrieben.

4.5 Domain Services

Domain Services bilden Aktionen ab, die im ausführenden Code später die Lesbarkeit erhöhen und diese Aktionen standardisieren. In der Domäne der Pflanzenpflege gibt es einige Domain Services, die abgebildet werden müssen. Diese lassen sich dabei in 3 Grundarten unterteilen: Pflegeplan-Management (carePlanManagement), Platzierungs-Management (locationManagement) und Pflanzen-Management (plantManagement).

In den Grundarten bilden sich verschiedene Aktionen ab, die alle später durch Verwendung der Domain Services automatisiert durchgeführt werden können, wie zum

Beispiel die Domain Services AddTaskToCarePlan oder GetDueTasksForCarePlan. Die lassen den späteren Ausführungscode verboser, im Sinne des Domain Driven Designs, erscheinen.

Innerhalb des Codes überscheiden sich die Domain Services mit dem Prinzip der Use Cases der Clean Architecture und werden deshalb auch als solche betitelt.

Kapitel 5

Refactoring

Im Zuge des Refactoring wurde zuerst einmal ein code review durchgeführt. Dabei wurde der gesamte erstellte Code im Repository einmal „mit dem Auge“ auf Auffälligkeiten geprüft, danach ein zweites Mal mit einer Checklist, die Anhand der Vorlesung erstellt wurde. Diese Checkliste beinhaltete die besprochenen Code Smell Beispiele, konkret Duplicated Code, Long Method, large Class, Shotgun Surgery, Switch Statements, Code Comments, geprüft. Hierbei wurden mehrere, zumeist kleinere Stellen identifiziert, bei denen Ausbesserungsbedarf bestand. Bei diesen Findings handelte es sich um Code Smell Fälle, die wir bereits konkret in der Vorlesung behandelt hatten:

1. Es wurde ein Switch-Statement für die Identifizierung eines Enums verwendet
2. In einem Konkreten Fall eine Methode mit auslagerbarem Code identifiziert („Long Method“)
3. Execute Methoden der Use Cases beinhalteten Error Return Werte
4. Viel ähnlicher Code in Use Case Definitionen

Bei weiteren waren minimale Optimierungen anderer Aspekte gefragt, um den Code etwas ausbessern und lesbarer zu machen:

a. Manche Methodennamen hielten sich nicht an die gängigen C# Benennungs-konventionen (hauptsächlich Großschreibung) oder waren im Sinne des Domain Driven Designs nicht optimal benannt

b. Manche Variablen konnten den Wert NULL annehmen, waren aber nicht entsprechend konfiguriert

c. Inkonsistente Variableninitialisierung in Use Cases

Diese Findings nach der Identifizierung in einem neuen Zweig des Repositories jeweils einzeln bearbeitet.

Im ersten Finding wurde im Konstruktor der Klasse CareTask eine statische Methode MatchesDetailsType zum identifizieren des Task Typen eines Objektes einer Klasse, die aus dem Interface ICareTaskDetails, also eine detaillierte Pflegeaufgabe. Die Statische Methode verwendete dabei einen Switch Case und iterierte über den Enum-Typen CareType und den passenden Typen der Variable type zu finden. Dabei handelt es sich um ein gutes Besipiel des Code Smells Switch Statements, und damit im einen Shotgun Surgery Smell im erweiterten Sinne. Wird das Enum erweitert, so muss der Entwickler auch wissen, dass dieses Switch Statement exisitert und das er erweitert werden muss, sonst baut ein unwissender Entwickler eine Fehlerquelle in den Code. Um eben nicht bei jeder Erweiterung des Enums CareType auch das Switch Statement der MatchesDetailsType erweitern zu müssen, muss hier deshalb eine Änderung vorgenommen werden. Der Klassische Lösungsansatz solcher Problematiken ist dabei das Nutzen von Polymorphie um das Switch Statement als Conditional zu ersetzen. Dies wurde auch hier angewandt.

Dafür wurde das Enum aufgelöst, und jedes Element des Enum zu einer eigenen Klasse umgeschrieben, die alle von einer abstakten ParentKlasse CareType erben. Diese bekommt eine abstrakte Methode Matches zugeordnet, die alle Children implementieren. Dabei wird der CareType des übergebenen Arguments von Typ ICareTaskDetails auf Gleichheit mit dem jeweils eigenen Typen geprüft. So muss in der Kontruktor von CareTask nurnoch vom übergebenen Argument type des Typen CareType die matches methode auf das andere Argument details aufrufen und erhält dasselbe Ergebnis wie zuvor. Wird ein neuer CareType hinzugefügt, muss dies aber nur noch an einer Stelle gemacht werden und der Entwickler braucht keine Kenntnis mehr über die Existens des Switch Cases in der Klasse CareTask.

Im zweiten Finding wurde im Use Case CreateCarePlan, in der Execute Methode über eine übergebene Liste von PflanzenIDs iteriert, die korrespondierenden Objekte aus einem Repository-Objekt entnommen und dem CarePlan hinzugefügt. Hierbei handelt es sich um einen Long Method Smell (im engeren Sinne auch um ein Code Comments

Smell, da der Block logisch getrennt vom Rest des Methodencodes ist; allerdings gab es hier keine Abtrennung per Kommentar, also trifft der Smell eventuell nicht 100% zu). Zur Lösung dieses Smells wurde die sogenannte „Extract Method“-Lösung angewandt.

Dazu wurde der Code aus der Methode heraus separiert und in eine eigene Methode ausgelagert, die mit dem Keyword protected versehen wird, um nur klasseninterne Zugriffe zuzulassen. Der Name der neuen Methode wurde so gewählt, dass das Lesen der Methode Execute im Sinne des Domain Driven Designs möglichst umkompliziert und in der ubiquitous language bleibt. So entstand die Methode FillCarePlanWithPlants, die als Argumente einen CarePlan und eine Liste von PlantIDs erwartet, das Heraussuchen der Pflanzen aus dem Repository übernimmt und den CarePlan damit auffüllt. Anschließend wird der aufgefüllte CarePlan zurückgegeben.

Beim dritten Finding wurde identifiziert, dass viele UseCases, die entweder etwas Löschen oder Ändern sollen, bei einer fehlgeschlagenden Suche des jeweiligen Objektes aus einem Repository als Return-Wert false übergeben, was äquivalent zu einem Fehler angesehen werden kann. Diese können allerdings auch direkt als Fehler identifiziert und über eine Exception handgehabt werden. Dafür wurde beispielsweise in der ExecuteAsync Methode der use case Klasse AssignPlantToCarePlan eine solche sogenannte „Replace Error Code with Exception“ Lösung umgesetzt. Hierfür wurden neue Exceptions im Domain Code erstellt. Dafür wurde zuerst die Parent Exception NotFoundException kreeirt, und dann die beien Children Exceptions CarePlanNotFoundException und PlantNotFoundException. Diese können dann anstelle einer Wiedergabe von false implmenetiert werden und können durch Ausgabe einer Fehlermeldung über das Argument message verboser das eigentliche Problem beim Ausführen der Methode ansprechen. Diese Implementation könnte auch in Use Case Klassen wie DeleteCarePlan und DeleteTaskFromCarePlan implementiert werden, da auch dort der Fall auftritt, bei nicht-Finden des zu Entfernenden Objektes der Wert false zurückgegeben wird.

Im vierten Finding wurde Identifiziert, dass sich der Code, gerade in den Execute bzw ExecuteAsync Methoden vieler Use Cases ähneln. Bei näherer Betrachtung sind diese Methoden allerdings immer leicht unterschiedlich, und sich überschneidende Teile sind minimal. Deshalb wurde von einer Optimierung beispielsweise im Sinne einer Extraktion von Code abgesehen. Da diese wahrscheinlich nur wenige Zeilen code beinhalten und auch beim Leseverständnis nicht unbedingt weiterhelfen würden. Ein weiterer Gedanke zur Optimierung wäre eventuell eine Vererbung, die die Überlappenden Teile in einer Parent Klasse übernimmt. Aber auch hier konnte im Zuge des Refactorings

keine zufriedenstellende Umsetzung gefunden werden, die den Code optimiert und das Leseverständnis nicht beeinträchtigt.

Bezüglich der kleineren Findings wurde in Finding a bemerkt, dass manche Methoden der CarePlan Klasse mit Kleinbuchstaben beginnen. Konvention bei C# ist allerdings, Methoden mit Großbuchstaben beginnen zu lassen, was ein kleines Renaming Refactoring zur Folge hatte. Dies gilt dabei auch für Attribute von Klassen, die Getter und Setter definieren.

Zu Finding b lässt sich sagen, dass es in C# best practice ist, eine Variable, die auch den Wert NULL annehmen können soll, entsprechend zu konfigurieren. Entweder, wenn die Variable gleich mit Inhalt befüllt wird, kann das Keyword var anstatt eines Variablentypen eingesetzt werden. Der Compiler prüft dann auf den Typen der befüllt werden soll, und konfiguriert automatisch die Nullbarkeit der Variable. Dies kann aber auch manuell umgesetzt werden, indem hinter das Keyword des Variablentypen ein Fragezeichen gesetzt wird. So weiß der Compiler, dass für diese Variable auch NULL ein gültiger Wert sein kann. An vereinzelten Stellen wurde diese Konfiguration vergessen, und im Zuge des Refactorings entsprechend angepasst.

Finding c bezieht sich inhaltlich auf Finding b. So wurden in Use Cases fast durchgängig var Keywords in den Execute Methoden verwendet, vereinzelt aber auch die Typen. Um hier konsistent zu bleiben, und auch um die Nullbarkeit von Finding b zu erhalten, wurde hier auf konstanten Nutzen von var umgestellt. Nachdem alle Findings bearbeitet, und kleiner Fehler bei der Bearbeitung, wie zum Beispiel eine vergessene Negierung bei der Gleichheitsprüfung von Finding 2, behoben wurden, wurde ein Merge Request vom Nebenzweig auf den Hauptzweig erstellt, nochmals reviewed und gemergt.

Kapitel 6

Programming Principles

Im Sinne der Programming Principles wird hauptsäch Wert auf die Bewertung der SOLID und DRY Prinzipien, sowie die Kopplung und Kohäsion des GRASP Prinzips gelegt.

6.1 SOLID Prinzipien

Das SOLID Prinzip schreibt durch sein Akronym folge Prinzipien vor:

6.1.1 Single Responsibility Principle (SRP)

Das SOLID Prinzip schreibt durch sein Akronym folge Prinzipien vor:

6.1.2 Open/Closed Principle (OCP)

Alle Klassen legen Wert darauf, möglichst offen einzelne Typenprüfungen zu gestalten, sodass nirgends starre Gebilde (wie zum beispiel über Enums iterierende If-Schlangen) aufkommen (siehe auch Refactoring des Switch Statements).

6.1.3 Liskov Substitution Principle (LSP)

Der Code stützt sich viel auf Vererbung von Klassen als Erweiterung von Eigenschaften. Dabei wurde nicht immer darauf geachtet, ob das Liskov Substitution Prinzip verletzt wird. Im Zuge des Refactoring ist allerdings kein solcher Fall erkannt worden. Grundsätzlich wurde aber im Sinne des LSP entwickelt, sodass keine Spezialisierung durch Vererbung eine Generalisierung invalidiert. Übertypen, also Parent Klassen, sollten immer genau so anwendbar sein, wie ihre Children, sollten sie dort gefragt sein.

6.1.4 Interface Segregation Principle (ISP)

Dieses Prinzip wurde bei den Repositories konkret umgesetzt, da dort durch Vererbung spezialisierte Typen von Interfaces entstehen.

6.1.5 Dependency Inversion Principle (DIP)

Auch hier kommen die Repository-Interfaces ins Spiel. Sie dienen dazu, die Adapterschicht mit der Applikationsschicht interargieren zu lassen, ohne direkten Zugriff zu erhalten. Adapter können die Interfaces benutzen um ihre generierten Daten in die Applikationsschicht, spezifisch in Use Cases, zu übergeben und deren spezialisierte Arbeitsweise zu nutzen.

6.2 DRY Prinzip

Es wurde bei der Entwicklung bereits stark darauf geachtet, Code möglichst nur einmal zu verfassen, und bei Wiederholungen zu schauen, ob nicht eventuell der bereits bestehende Code ausgelagert werden kann. Zusätzlich wurde während des Refactoring nochmal in den Use Cases ein solcher Code Smell identifiziert und auch beschrieben, allerdings konnte keine gute Lösung zur Behebung gefunden werden, da die ich wiederholenden Stellen gespickt mit kleinen unterscheiden sind. Dies macht eine Extraktion in ausgelagerte Methoden kompliziert.

6.3 Kopplung und Kohäsion

6.3.1 Kopplung

Eine relativ geringe Kopplung erreicht wurde bereits durch das Anwenden der Clean Architecture, die die Einzelnen Schichten voneinander trennt und Möglichkeiten zur Interaktion bietet, ohne die Kopplung zu erhöhen. Auch die Nutzung der Dependency Inversion trägt hierzu bei.

6.3.2 Kohäsion

Hier besteht bestimmt noch Optimierungsbedarf, da beim Entwickeln der Klassen, besonders der Entitäten darauf nicht besonders geachtet wurde. Jedoch sind alle Entitäten in ihrer Größre relativ gering, was eigentlich ein Zeichen guter Kohäsion ist.