Programming Principles

Lukas Panni
TINF18B5

DHBW Karlsruhe
Vorlesung Advanced Software Engineering Semester 5/6

Inhaltsverzeichnis

1 SOLID

SOLID steht für fünf Prinzipien, die bei der Softwareentwicklung eingehalten werden sollten. Im Folgenden werden diese Prinzipien kurz erklärt und deren Einhaltung beziehungsweise Verwendung in diesem Projekt analysiert und begründet.

1.1 Single Responsibility Principle (SRP)

Das **Single Responsibility Principle** besagt, dass jede Klasse genau eine Aufgabe erfüllen sollte und es deshalb auch nur einen Grund geben sollte, eine Klasse zu ändern. Das Prinzip lässt sich ebenso auf andere Struktureinheiten des Codes, wie zum Beispiel Methoden oder auch Module übertragen. Dabei gilt auch, dass die *eine Aufgabe* auf Modulebene abstrakter ist als auf Klassenebene und sich aus Aufgaben mehrerer Klassen zusammensetzen kann.
Vorteile des Single Responsibility Principle sind unter anderem eine bessere Wiederverwendbarkeit, Wartbarkeit und Testbarkeit.

1.1.1 Analyse und Begründung für SRP

Bei der Betrachtung der Projektstruktur wird bereits deutlich, dass die Aufteilung des Codes in mehrere Packages bereits der Umsetzung des SRP dient. Zum Beispiel enthält das *auth*-Package ausschließlich Code, der für die Authentifizierung (kurz: Auth) verantwortlich ist. Hier finden sich Interfaces und dazugehörige Implementierungen, die nur für die Authentifizierung eines Nutzers verwendet werden. Betrachtet man innerhalb dieses Packages die Klasse *GithubOAuthHandler* genauer, erkennt man, dass diese Klasse die Hauptaufgabe hat, einen Nutzer über GitHub zu authentifizieren. Allerdings bietet diese Klasse auch Funktionalität, eine erfolgreiche Authentifizierung für weitere Aktionen zu nutzen und eine Authentifizierung zu prüfen, was das SRP verletzt. Durch diese bewusste Verletzung des Prinzips wird in diesem Fall in Kauf genommen, da so eine Fragmentierung ähnlicher und fundamental zusammengehöriger Logik über mehrere sehr kleine Klassen vermieden werden kann.
Betrachtet man das *client*-Package erkennt man eine klare Zusammenstellung von Klassen mit dem Zweck, Daten über die GitHub-API abzurufen und Callback-Methoden mit diesen Daten aufzurufen. Zusammengefasst kümmert sich dieses Package um die Arbeit mit Daten aus der GitHub-API. Die Klasse *GithubOAuthClient* kann dabei als weiteres Beispiel für eine bewusste Verletzung des SRP angesehen werden, da hier die Funktionalität für das abrufen verschiedener Daten gesammelt ist um auch hier eine Fragmentierung über mehrere Klassen zu vermeiden. So kann diese Klasse verwendet werden um Daten zu Repositories, aber auch Daten zu Contributions eines Nutzers abzurufen. Aktuell fällt

diese Verletzung nicht weiter negativ auf. Das Potential für negative Effekte, zum Beispiel durch Erweiterung um weitere Funktionalität, ist allerdings gegeben, weshalb in einem zukünftigen Refactoring diese Klasse aufgeteilt werden könnte.
Das *data*-Package ist vergleichsweise trivial, da es nur Klassen sammelt, die als Nutzdaten innerhalb der Anwendung verarbeitet werden. Alle Klassen in diesem Package haben als einzige Aufgabe die Repräsentation von Daten, die von der GitHub-API geladen wurden.
Im *repository*-Package ist Funktionalität zur Verwaltung von Daten realisiert. Dazu gehört auch das Sub-Package *cache*, dessen einziger Zweck das Caching von Daten ist. Diese Funktionalität ist hauptsächlich in der Klasse *ResponseCache* realisiert, die auch sonst keine weiteren Aufgaben erfüllt. Die abstrakte *Repository* Klasse und ihre Ableitungen stellen den Rest des *repository*-Packages dar und haben die einzige Aufgabe, das Caching der Daten für Verwender transparent zu gestalten, sodass ein Verwender Daten aus einem Cache beziehen kann anstatt sie jedes mal über die GitHub-API abzurufen. Weitere Verantwortlichkeiten lassen sich weder innerhalb der einzelnen Klassen, noch insgesamt auf das Package bezogen, erkennen.
Als Letztes Package kann das *ui*-Package genauer betrachret werden. Hier sind alle Klassen gesammelt, die für die Implementierung des User Interfaces benötigt werden. Damit ist die Aufgabe dieses Packages, UI-Logik zu realisieren. Da das User-Interface in mehrere Teile mit spezifischeren Funktionen aufgeteilt ist, ist auch das gesamte Package in mehrere Sub-Packages zerlegt. Aber auch diese Sub-Packages erfüllen jeweils einen klar abgegrenzten Zweck. Als Beispiel hat das Sub-Packate *dashboard* die einzige Aufgabe, die Dashboard-Unterseite des User-Interfaces darzustellen.

1.2 Open / Closed Principle (OCP)

Nach dem **Open / Closed Principle** sollten Module immer offen für Erweiterungen und geschlossen für Änderungen sein. Dadurch sollen Erweiterungen erleichtert werden, ohne dass für eine Erweiterung bestehender Code angepasst werden muss. So wird der Code unter anderem auch Modularer.

1.2.1 Analyse und Begründung für OCP

Insgesamt wurde beim Design der Klassen immer darauf geachtet, Erweiterungen mit möglichst wenig Aufwand zu ermöglichen. Im Laufe der Entwicklung konnten die Vorteile davon bereits mehrfach ausgenutzt werden um Erweiterungen der bestehenden Funktionalität zu realisieren. Da hier nicht für alle enthaltenen Klassen eine ausführliche Betrachtung erfolgen kann werden stattdessen einige Beispiele angeführt.
Ein gutes Beispiel für die Umsetzung des OCP stellt die Klasse *ResponseCache* dar. Diese Klasse wurde dafür konzipiert die Verwendung mit neuen Datentypen möglichst einfach zu gestalten. Anstatt eine Cache-Klasse für einen festen Datentyp zu erstellen wurde diese Klasse durch Verwendung von Generics so ausgelegt, dass alle Datentypen, die ein bestimmtes Interface implementieren, im Cache gespeichert werden können. Damit erfordert eine Erweiterung um Caching anderer Datentypen keinen weiteren Aufwand als die Implementierung der benötigten Interfaces.

1.3 Liskov Substitution Principle (LSP)

Im **Liskov Substitution Principle** ist beschrieben, dass eine Instanz einer abgeleiteten Klasse jederzeit so verwendbar sein soll wie eine Instanz der Basisklasse. Dabei wird nicht nur gefordert, dass die Verwendung technisch möglich ist, sondern dass diese Art der Verwendung auch logisch sinnvoll und frei von unerwünschten Nebeneffekten ist. Das führt schließlich dazu, dass jede Vererbung genauer betrachtet werden muss um zu prüfen, ob abgeleitete Klassen sinnvoll wie die Basisklasse eingesetzt werden können. Dadurch wird eine bessere, verständlichere Abstraktion gefördert und auch Fehler durch falsche Annahmen bezüglich Polymorphie werden verringert.

1.3.1 Analyse und Begründung für LSP

Bei Betrachtung des Codes fällt auf, dass nur an vergleichsweise wenigen Stellen Vererbung eingesetzt wurde, was die Betrachtung in Bezug auf das LSP erleichtert. Das vermutlich beste Beispiel für die Verwendung des LSP findet sich im *ui*-Package mit der Klasse *DataAccessFragment* und ihren Ableitungen. Alle von dieser Klasse abgeleiteten Klassen können auf die gleich Art verwendet werden um Daten mithilfe eines *DataAccessViewModel*s zu laden. Dabei sind die Ableitungen dafür verantwortlich, ein passendes ViewModel zu laden. Ansonsten kann auf jeder Instanz einer dieser Ableitungen die *loadData*-Methode aufgerufen werden um verschiedene Daten zu laden und im User-Interface anzuzeigen.

1.4 Interface Segregation Principle (ISP)

Das **Interface Segregation Principle** sagt aus, dass mehrere spezifische Interfaces besser sind als ein einzelnes unspezifisches Interface. Dadurch soll der Code modularer, wartbarer und besser strukturiert werden. Allerdings muss dabei darauf geachtet werden, dass nicht zu viele Interfaces erstellt werden.

1.4.1 Analyse und Begründung für ISP

Es wird schnell deutlich, dass innerhalb der Anwendung viele Interfaces eingesetzt werden. Das kann als Zeichen dafür gesehen werden, dass das ISP konsequent umgesetzt wurde und große generische Interfaces weitgehend vermieden wurden. Allerdings kann es auch als Indiz dafür betrachtet werden, dass die Interfaces zu klein gewählt wurden. Da aber nur wenige Klassen mehr als ein Interface implementieren, kann davon ausgegangen werden, dass die Aufteilung nicht zu kleinteilig erfolgt ist. Ein Beispiel für eine Klasse, die mehrere Interfaces implementiert ist die Klasse *GithubOAuthClient*, die die Interfaces *RepositoryDataClient* und *UserContributionsClient* implementiert. Wie bereits in 1.1.1 beschrieben, realisiert diese Klasse die Funktionalität für das Abrufen von verschiedenen Arten von Daten. Ebenfalls bereits beschrieben wurde, dass diese Klasse zu groß ausfällt und eventuell weiter zerlegt werden muss. Deshalb lässt sich diese Klasse auch nicht als echtes Negativbeispiel für eine zu feingranulare Interface-Definition anführen.
Zusammenfassend kann man davon ausgehen, dass das ISP in der gesamten Anwendung umgesetzt wurde.

1.5 Dependency Inversion Principle (DIP)

Das **Dependency Inversion Principle** besagt, dass High-Level Module nicht von Low-Level Modulen direkt abhängen sollten, sondern beides möglichst von Abstraktionen abhängen sollte. Weiterhin sollen Abstraktionen nicht von Details, sondern Details von Abstraktionen abhängen. Dadurch soll der Code insgesamt wartbarer, modularer und besser wiederverwendbar gestaltet werden.

1.5.1 Analyse und Begründung für DIP

Bei der Entwicklung wurde versucht, das DIP wo möglich umzusetzen. Auch hier lässt sich die Klasse *ResponseCache* als Beispiel für die Umsetzung verwenden. Wenn man diese Klasse als Modul höherer Ebene ansieht und die Klassen, deren Objekte im Cache gespeichert werden können, einer niedrigeren Ebene zuordnet, dann erfüllt diese Klasse das DIP. Anstatt von konkreten Klassen abhängig zu sein, bestehen im ResponseCache nur Abhängigkeiten zu Abstraktionen, wie zum Beispiel der abstrakten Klasse *ResponseData*. So wird die Wiederverwendbarkeit dieser Klasse enorm erhöht.
Ein weiteres Beispiel findet sich in der Klasse *GithubOAuthClient*, die anstatt direkt von der einzigen Implementierung des Interfaces *AuthenticationHandler* abhängt, sondern nur von diesem Interface. In diesem Fall wird so hauptsächlich die Testbarkeit der Klasse erhöht, da eine spezifische Test-Implementierung des Interfaces für den Test verwendet werden kann.
Mit den Klassen *UserContributionsRepository* und *RepositoryDataRepository* gibt es noch zwei weitere Beispiele für eine gute Umsetzung dieses Prinzips. Anstatt direkt von der Klasse *GithubOAuthClient* abzuhängen, gibt es nur Abhängigkeiten zu den Interfaces *UserContributionsClient* beziehungsweise *RepositoryDataClient*. Die Erweiterbarkeit wird so deutlich verbessert, da jede beliebige Implementierung des jeweiligen Interfaces verwendet werden kann anstatt ein GithubOAuthClient-Objekt zu verwenden. Dadurch wird auch die bereits angesprochene mögliche Aufteilung der Klasse *GithubOAuthClient* deutlich einfacher möglich.

2 GRASP