

Programmentwurf Paketmanager

Name: Nitz, Franz Josef
Matrikelnummer: 2791813

Abgabedatum: 04.06.2022

Repository: <https://github.com/FranzJosef2000/ASE-Projekt>

Kapitel 1: Einführung

Übersicht über die Applikation

Es soll eine Software geschrieben werden, die einen transparenten Pakettransport für den Kunden ermöglicht. Mit dieser Applikation können Pakete in das System aufgegeben und der Sendestatus abgefragt werden. Zudem können die Pakete in ihrem Sendestatus verändert werden. Die Applikation ist eine CLI Anwendung und wird über die Kommandozeile gesteuert.

Paket aufgeben:

CREATE PACKAGE <Paketkategorie>

Beispiel: CREATE PACKAGE PARCEL_S

Sendestatus zu einem Paket erfragen:

GET PACKAGE <Sendeverfolgungsnummer>

Beispiel: GET PACKAGE cec70e16-5a5e-4708-8ada-c57f0dc1519e

Status des Pakets verändern (nur für Mitarbeiter der Post):

PUT PACKAGE <Sendeverfolgungsnummer>

Beispiel:

PUT PACKAGE cec70e16-5a5e-4708-8ada-c57f0dc1519e

ID: vrv81r38-6s6s-5819-9sfs-v68g1fv2620r

Passwort: test123

Hilfe:

HELP

Paketkategorien herausfinden:

HELP CATEGORY

Mit dieser Software wird das Problem der Intransparenz gelöst und dass man oft nicht genau weiß in welchem Zustand der Zustellung das Paket sich befindet. Da die aktuellen Lösungen mehr gut gedacht als gut umgesetzt sind wurde eine Applikation, die die aktuellen Standards der Softwareentwicklung einhält, entwickelt.

Wie startet man die Applikation?

Starten: Die Applikation muss in der IDE über den Startpfeil gestartet werden oder wenn dieser nicht funktioniert über einen rechtsklick auf die Mainmethode (packageTrackingManager) und auf Run packageTrackingManager.

Danach startet die Anwendung und in der CLI kann dann ein Kommando eingegeben werden.

Voraussetzungen: Für eine Reibungslose Funktion wird Java Version ≥ 12 benötigt und eine Entwicklungsumgebung benötigt

Wie testet man die Applikation?

Testen: Alle Tests können mit einem Rechtsklick auf den Ordner „test“ und dann auf „Run Tests in test“ geklickt werden. Danach werden alle Tests automatisch ausgeführt.

Voraussetzungen: Java Version > 12 , eine IDE und JUNIT 5

Kapitel 2: Clean Architecture

Was ist Clean Architecture?

Clean Architecture legt den architekturellen Aufbau einer IT-Anwendung fest. Dabei liegt im Mittelpunkt die Unabhängigkeit der fachlichen Anwendung zur Umgebungsinfrastruktur. Dadurch wird Weiterentwickelbarkeit und Wartbarkeit gewährleistet.

Clean Architecture ist in Schichten aufgebaut, wobei die innersten Schichten nie bzw. so gut wie nie verändert werden sollten und je weiter man mit den Schichten nach außen kommt desto mehr dürfen Änderungen im Code vorgenommen werden.

Die Schichten sind (von innen nach außen):

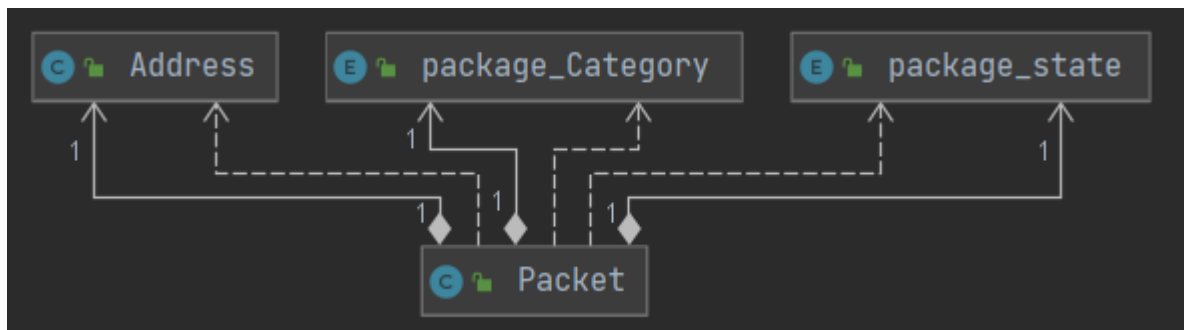
Abstraction Code (mathematische Konzepte), Domain Code (Entitäten), Application Code (Anwendungsfälle), Adapters (Controller, Schnittstellen), Plugins (Drittsysteme oder Datenbanken)

Analyse der Dependency Rule

Positiv-Beispiel: Dependency Rule

Klasse: Packet

UML:

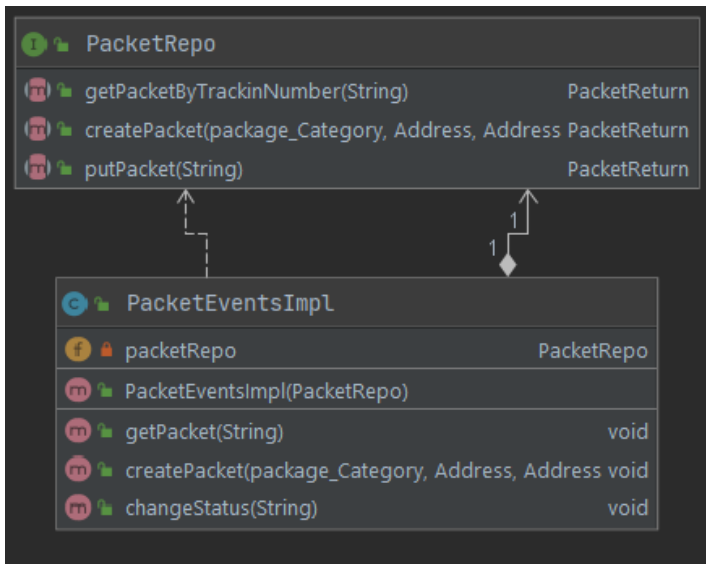


Analyse: Die Klasse Packet hat nur Abhängigkeiten innerhalb der Domain Schicht zu Address, package_Category und package_State und keine von innen nach außen. Aus diesem Grund hält Packet die Dependency Rule ein.

Negativ-Beispiel: Dependency Rule

Klasse: PacketEventsImpl

UML:



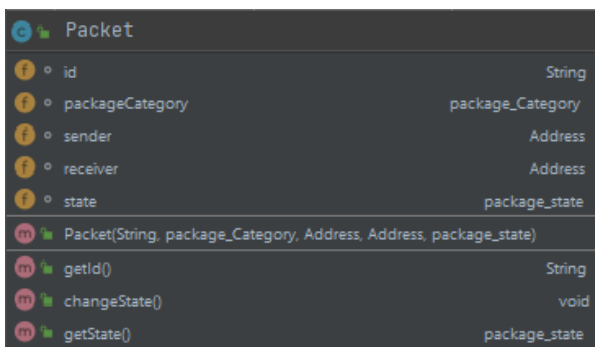
Analyse: **PacketEventsImpl** (ApplicationCode) besitzt eine Abhängigkeit zu **PacketRepo** (Adapter). Hierbei geht die Abhängigkeit von innen nach außen und verletzt damit die Dependency Rule.

Analyse der Schichten

Schicht: [Domain-Code]

Klasse: Packet

UML:



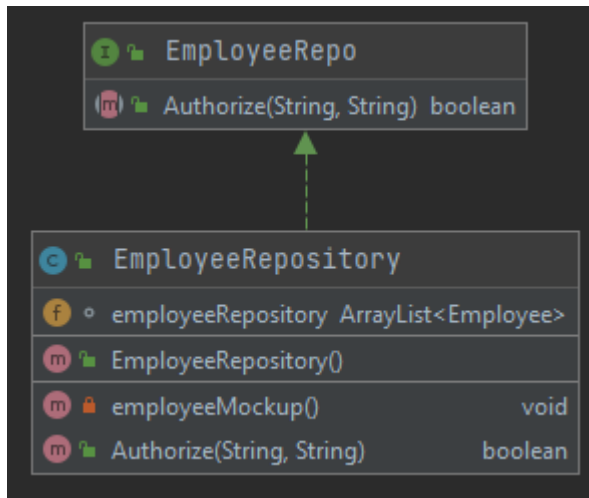
Aufgabe: Definieren der Funktionen und Variablen, die für ein Paket notwendig sind. Wird zur Paketobjekterzeugung verwendet.

Einordnung: Wichtiger Grundbaustein für das komplette Projekt, ohne diese Klasse ist die Umsetzung nicht möglich. Wird nicht bzw. sollte nicht verändert werden.

Schicht: [Adapter]

Klasse: EmployeeRepo

UML:



Aufgabe: Weitergabe der Daten des Repositorys in Richtung des Applicationcode.

Einordnung: Weil diese Klasse nur Daten an eine innere Schicht weitergibt und keine Implementation stattfindet.

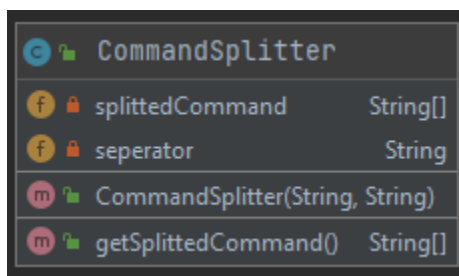
Kapitel 3: SOLID

Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel

Klasse: CommandSplitter

UML:

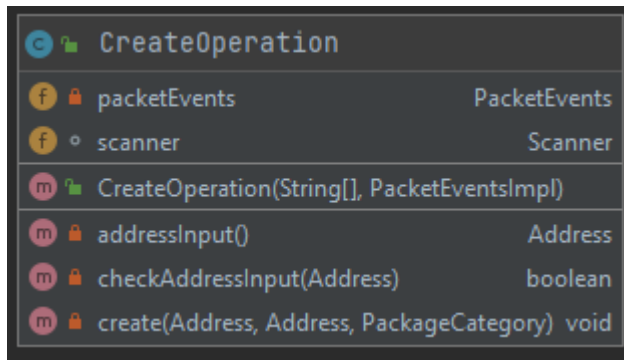


Aufgabe: Über einen Seperator (Leerzeichen) wird der Eingabestring, der über die Kommandozeile eingegeben wird, in einzelne Wörter aufgeteilt. Diese Wörter werden dann als Array zurückgegeben

Negativ-Beispiel

Klasse: CreateOperation

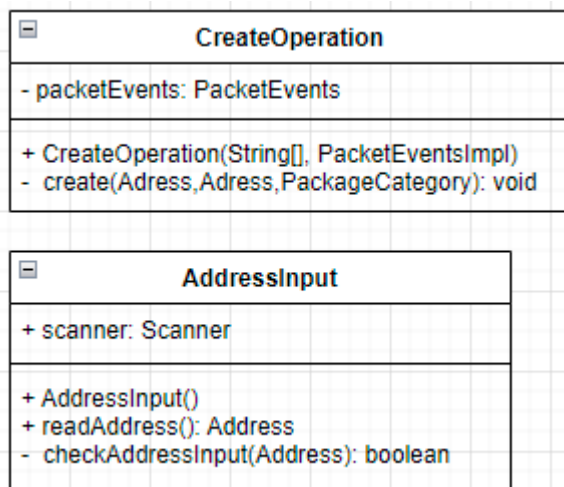
UML (vorher):



Aufgabe: Soll am Ende den Aufruf zur Paketerstellung triggern. Dafür werden die Eingaben zum Empfänger und Absender eingelesen und überprüft. Außerdem wird die Paketkategorie überprüft.

Lösungsweg: Erster Lösungsschritt Einlesen und Überprüfen der Daten für Absender und Empfänger in eine eigene Klasse auslagern.

UML (nachher):

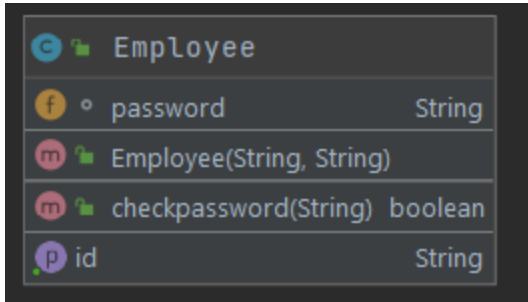


Analyse Open-Closed-Principle (OCP)

Positiv-Beispiel

Klasse: Employee

UML:

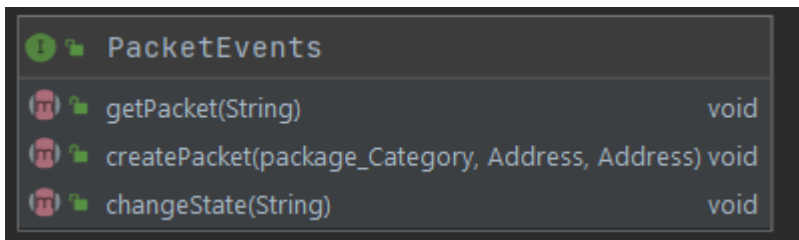


Begründung: Ein Mitarbeiter kann um zusätzliche Attribute erweitert werden wie z.B. Vorname, Nachname, Standort, Dabei wird nur die Klasse Employee erweitert und keine weiteren Klassen verändert.

Negativ-Beispiel

Klasse: PacketEvents

UML:



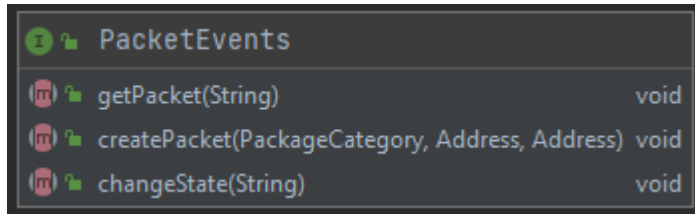
Begründung: Das Interface könnte über eine Methode deletePacket erweitert werden. Diese müsste dann sowohl in der Klasse PacketEventsImpl als auch in den Repos PacketRepo und PackerRepository verändert werden.

Analyse Liskov-Substitution- (LSP), Interface-Segregation- (ISP), Dependency-Inversion-Principle (DIP)

Positives Beispiel ISP:

Klasse: PacketEvents

UML:



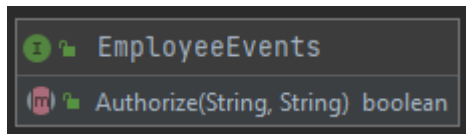
Begründung:

ISP wurde hier erfüllt, weil das Interface möglichst schmal gehalten wurde. Da für ein Paket diese Methoden minimal benötigt werden, um diese Aufgaben zu erfüllen.

Positives Beispiel ISP:

Klasse: EmployeeEvents

UML:



Begründung:

ISP wurde hier erfüllt, weil das Interface minimal gestaltet ist. Es wird nur Authorize implementiert, da ein Angestellter in der Anwendung stand jetzt keine weiteren Funktionalitäten benötigt, damit die Anwendung funktioniert. Andere Funktionalitäten können weiter hinzugefügt werden, wenn dies von Nöten ist, damit nicht sinnlos Methoden im Interface stehen die gar nicht verwendet werden.

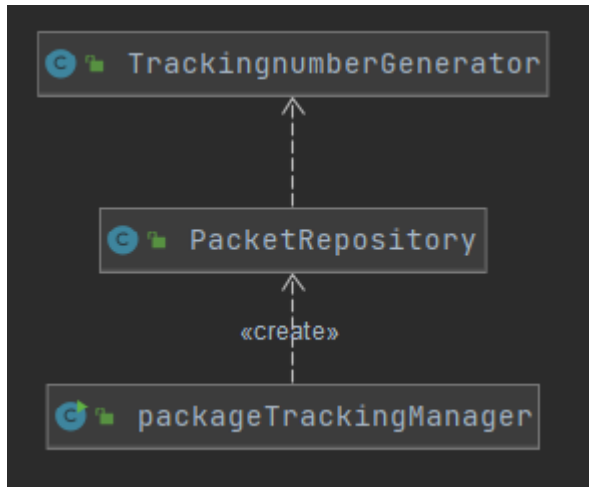
Kapitel 4: Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

Positiv-Beispiel

Klasse: PackageTrackinManager zu TrackingNumbergenerator

UML:

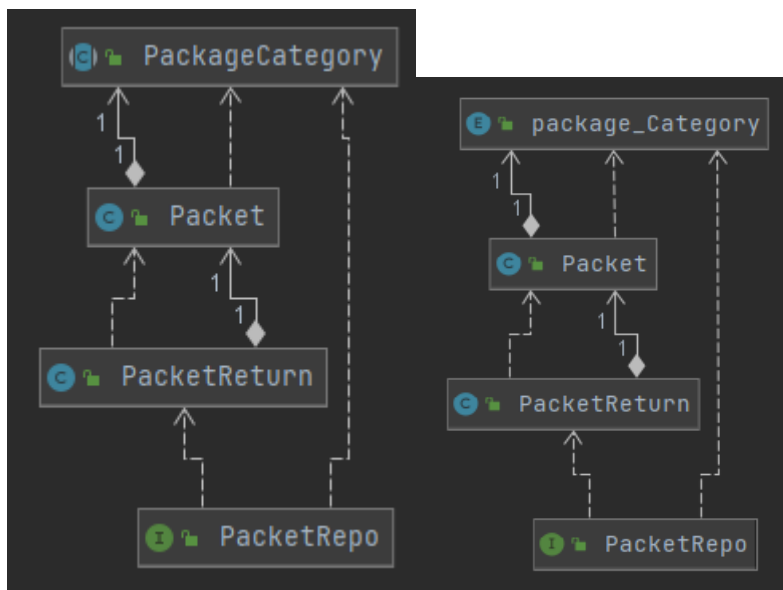


Begründung: zwischen diesen beiden Klassen besteht nur eine Abhängigkeit über die Klasse Package Repository.

Negativ-Beispiel

Klasse: PacketRepo zu PackageCategory

UML:



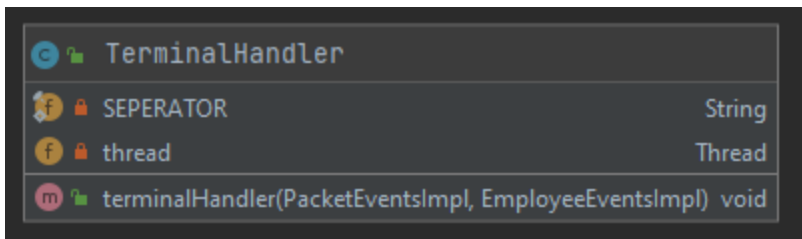
Begründung: PacketRepo hat sowohl eine direkte Abhängigkeit zu PackageCategory als auch indirekt über PacketReturn und Packet.

Lösung: Diese Abhängigkeit kann auch wie folgt modelliert werden: die Klasse PacketRepo hat eine Abhängigkeit zu PacketReturn diese wiederum zu Packet und die Klasse Paket hat dann die Abhängigkeit zu PackageCategory.

Analyse GRASP: Hohe Kohäsion

Klasse: TerminalHandler

UML:



Begründung: Alle Instanzvariablen von TerminalHandler werden in der einzigen Methode der Klasse, dem überladenen Konstruktor verwendet.

Don't Repeat Yourself (DRY)

Commit: Removed duplicated Code (DRY): 6d728e420356195cf284ac3931a9e5b6b0a2a672

Vorher:

```
if (command[POSITION].equalsIgnoreCase("CREATE") && command.length==3){
    new CreateOperation(command, packetEventsImpl);
}
else if (command[POSITION].equalsIgnoreCase("PUT") && command.length==3){
    new PutOperation(command, packetEventsImpl, employeeEventsImpl);
}
else if (command[POSITION].equalsIgnoreCase("GET") && command.length==3){
    new GetOperation(command, packetEventsImpl);
}
```

Nachher:

```
if (checkOperator(command, POSITION, "CREATE", 3)) {
    new CreateOperation(command, packetEventsImpl);
}
else if (checkOperator(command, POSITION, "PUT", 3)) {
    new PutOperation(command, packetEventsImpl, employeeEventsImpl);
}
else if (checkOperator(command, POSITION, "GET", 3)) {
    new GetOperation(command, packetEventsImpl);
}
```

```
private boolean checkOperator(String[] commands,int commandPosition, String commandCheck, int length){
    return commands[commandPosition].equalsIgnoreCase(commandCheck) && commands.length==length;
}
```

Begründung: Es wurde 3-mal fast das gleiche überprüft, lediglich ein Wort war unterschiedlich, dafür wurde eine Funktion geschrieben, die nur noch aufgerufen werden muss. Damit wird ein ungewolltes Verhalten was den anderen Funktionen abweicht vorgebeugt.

Kapitel 5: Unit Tests

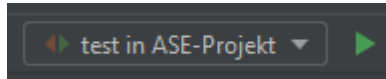
10 Unit Tests

Unit Test	Beschreibung
PacketRepositoryTest # getElementByIdTest	Überprüft ob mit dem gewählten Originalverfahren ein Element über die ID aus der ArrayList herausgefiltert werden kann.
PacketEventsImplTest # getPacketByTrackingNumberTest	Prüft das Paketsuchverhalten nach der Trackingnummer. Dabei soll ein Paket mit der richtigen Trackingnummer zurückgegeben werden.
PacketRepositoryTest # ChangeValueFromElementTest	Prüfen ob nach dem Aufruf der Methode auch sich der Wert des package_State verändert hat.
PacketClassTest#getIdTest	Prüfen ob getId die richtige ID zurückgibt,
EmployeeClassTest#checkPasswordTest	Prüft ob die Methode checkPassword mit richtigen Eingaben funktioniert.
EmployeeClassTest#checkpasswordTestNegative	Prüft ob falsche Passwörter erkannt werden und ein false zurückgegeben wird
PacketEventsImplTest# getPacketByTrackingNumberByUnsetPacketTest	Prüft das Paketsuchverhalten nach der Trackingnummer, wenn es kein Paket gibt. In so einem Fall soll ein null-Objekt zurückgegeben werden.
PacketEventsImplTest # changePacketByUnsetPacketTest	Prüft das Verhalten der Paketstatusänderung, wenn es kein Paket gibt. In so einem Fall soll ein null-Objekt zurückgegeben werden.
PacketEventsImplTest # createPacketTest	Prüft, ob ein Paketobjekt durch diese Methode erzeugt wurde.
PacketEventsImplTest # changeStateTest	Prüft das Verhalten der Paketstatusänderung im Normalfall.

ATRIP: Automatic

Automatic wurde realisiert, indem die Tests automatisch ablaufen, also ohne manuellen Input. Außerdem überprüfen sich die Tests selbst mit der Assert-Funktion.

Die Tests können mittels eines Klicks gestartet werden:



ATRIP: Thorough

Positiv: PacketEventsImplTest, ist Thorough, da in dieser Testklasse alle notwendigen Dinge getestet wurden inkl. Negativtest. Bei PacketEventsImpl werden die Funktionen getPacketByTrackingnumber, createPacket und changeState vollständig getestet.

Negativ: PacketRepositoryTest, ist nicht Thorough, da in diesem Fall keine sinnhaften Tests durchgeführt wurden, die alles abdecken, was die Klasse zu bieten hat.

ATRIP: Professional

Positiv: PacketEventsImplTest ist professional, weil hier die gleichen Qualitätsstandards wie im Produktivcode eingehalten wurden. Die Methode createMockPacket lagert mehrfach benutzten Code in eine Methode aus.

Negativ: getIDTest: ist nicht professionell, da hier ein getter getestet wurde welches unnötig ist.

Code Coverage

Die Testabdeckung der Klassen liegt bei 40%, 33% der Methoden wurden getestet und 26% der Zeilen sind getestet. Das ist ein ausbaubarer Zustand jedoch ist eine 100% Testabdeckung nicht erstrebenswert, da hier der Aufwand im Bezug auf Benefits und Sinnhaftigkeit nicht lohnend ist.

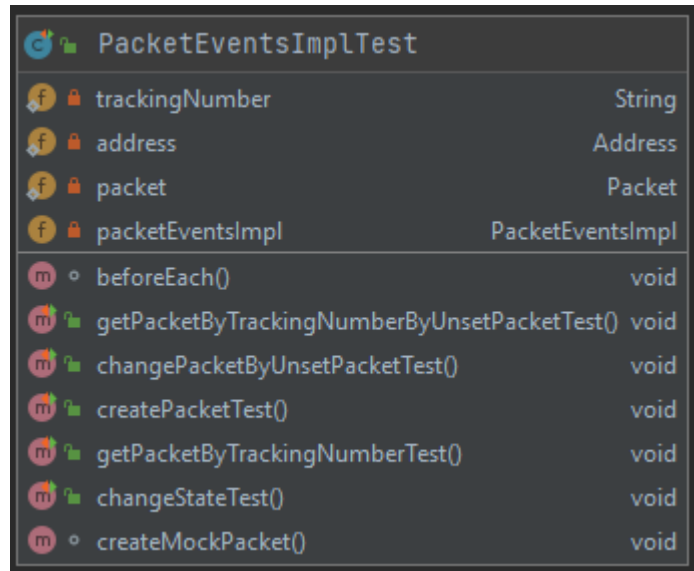
Element	Class, %	Method, %	Line, %
main	40% (11/27)	33% (23/69)	26% (81/31...

Fakes und Mocks

Moks und Fakes werden benötigt, um im Testfall immer die gleichen Ergebnisse zu bekommen und um sich von externen Services loszueisen.

1. Mock-Objekt: PacketEventsImplTest

a. UML:

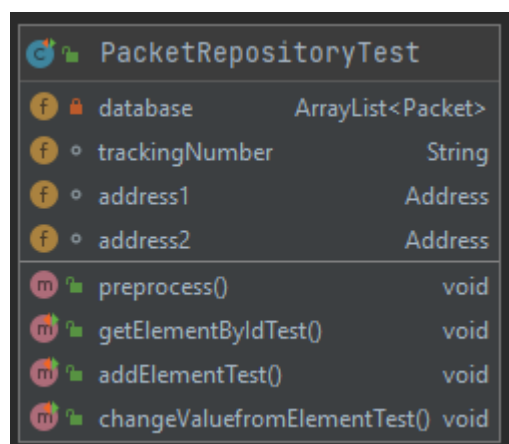


PacketEventsImplTest		
f	trackingNumber	String
f	address	Address
f	packet	Packet
f	packetEventsImpl	PacketEventsImpl
<hr/>		
m	beforeEach()	void
m	getPacketByTrackingNumberByUnsetPacketTest()	void
m	changePacketByUnsetPacketTest()	void
m	createPacketTest()	void
m	getPacketByTrackingNumberTest()	void
m	changeStateTest()	void
m	createMockPacket()	void

- b. Analyse: Die Tests in PacketEventsImplTest überprüfen die Aufrufe von PacketEvents. Dafür werden Daten benötigt, die auf einer Datenbank abgelegt sind.
- c. Begründung: Da UnitTests nur das jeweilige Modul ohne Abhängigkeiten testen sollen, sowie immer die gleichen Ergebnisse rauskommen, wurde ein Mock-Objekt verwendet. Hier in diesem Fall ist es keine Liste von Objekten, sondern nur ein Fake-Paket welches als Basis genommen wurde. Dadurch muss auf keine externe Datenhaltung (außerhalb der Klasse) zugegriffen werden.

2. Mock-Objekt

a. UML:



PacketRepositoryTest		
f	database	ArrayList<Packet>
f	trackingNumber	String
f	address1	Address
f	address2	Address
<hr/>		
m	preprocess()	void
m	getElementByIdTest()	void
m	addElementTest()	void
m	changeValuefromElementTest()	void

- b. Analyse: Die Tests in PacketRepositoryTest prüfen das Verhalten rund um den Speicher wo die Pakete abgelegt werden (ArrayList). Dafür wird diese Datenhaltungsform benötigt.
- c. Begründung: Damit die Tests immer den gleichen Ausgang bei gleichbleibenden Objekten haben, wurde eine eigene ArrayList für die Tests verwendet.

Kapitel 6: Domain Driven Design

Ubiquitous Language

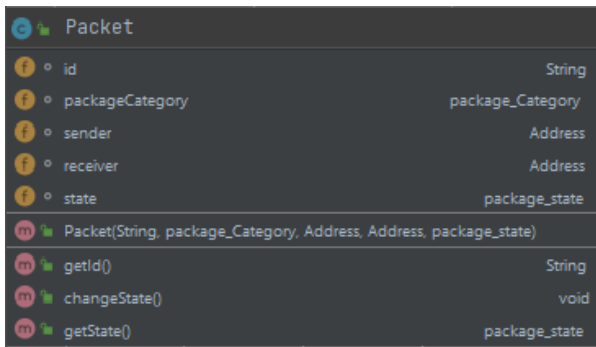
Es wurde sich für eine englische Domänensprache entschieden, damit mehr Personen die Bezeichnungen und Funktionalitäten verstehen.

Bezeichnung	Bedeutung	Begründung
createPacket	Methode zum Erzeugen eines Paketes	Durch die Bezeichnung wird klar, dass es sich um den Vorgang des Erstellens eines Paketes handelt. Damit ist sowohl den Entwickler als auch den Domänenexperten diese Funktionalität klar
generateTrackingNumber	Damit soll eine Trackingnummer für ein Paket generiert werden	Durch die Methodenbezeichnung wird die Funktionalität schnell klar, dass es sich hierbei um die Generierung einer Trackingnummer handelt.
checkpassword	Damit soll das Passwort eines Mitarbeiters überprüft werden	Durch den sprechenden Namen wird für alle Beteiligten klar um was es sich bei dieser Methode handelt. Alle Beteiligten kennen diese Bezeichnungen auch aus anderen Bereichen, da Passwörter oft verwendet und überprüft werden.
getPacketByTrackingNumber	Soll ein Paket zu einer bestimmten Trackingnummer ausgegeben	Damit wird für Entwickler und Domänenexperten klar, dass ein Paket durch seine Trackingnummer angezeigt wird. Die Methodenbezeichnung ist hier klar formuliert.

Entities

Entität: Packet

UML:



UML class diagram for the Packet entity. It shows a class named Packet with six attributes: id (String), packageCategory (package_Category), sender (Address), receiver (Address), and state (package_state). The class has three methods: Packet(String, package_Category, Address, Address, package_state), getId() (String), and changeState() (void). The state attribute is of type package_state.

Attribute	Type
id	String
packageCategory	package_Category
sender	Address
receiver	Address
state	package_state

Method	Return Type
Packet(String, package_Category, Address, Address, package_state)	
getId()	String
changeState()	void
getState()	package_state

Beschreibung: Definieren der Funktionen und Variablen, die für ein Paket notwendig sind. Wird zur Paketobjekterzeugung verwendet.

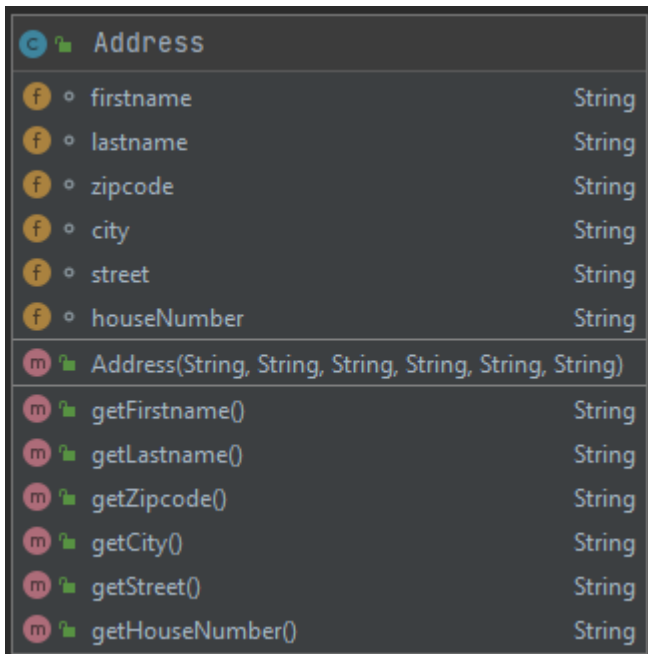
Begründung:

Weil ein Paket eine eindeutige Id besitzt und zwei Pakete nur gleich sind, wenn ihre Id's übereinstimmen.

Value Objects

Value Object: Address

UML:



UML class diagram for the Address value object. It shows a class named Address with six attributes: firstname (String), lastname (String), zipcode (String), city (String), street (String), and houseNumber (String). The class has seven methods: Address(String, String, String, String, String, String), getFirstname() (String), getLastname() (String), getZipcode() (String), getCity() (String), getStreet() (String), and getHouseNumber() (String).

Attribute	Type
firstname	String
lastname	String
zipcode	String
city	String
street	String
houseNumber	String

Method	Return Type
Address(String, String, String, String, String, String)	
getFirstname()	String
getLastname()	String
getZipcode()	String
getCity()	String
getStreet()	String
getHouseNumber()	String

Beschreibung: Definiert wie eine Adresse erzeugt wird, und implementiert die Methoden, die mit einem Address-Objekt genutzt werden können.

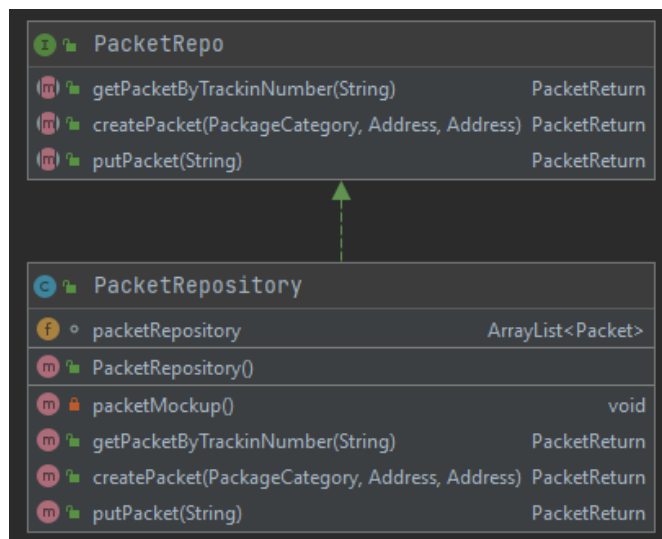
Begründung:

Eine Adresse ist unveränderlich, da z.B. durch eine alleinige Änderung der Postleitzahl eine ungültige Adresse entsteht, wo Postleitzahl und Stadt nicht mehr zusammenpassen. Aus diesem Grund darf und kann keine Änderung vorgenommen werden. Zwei Adressen sind nur gleich, wenn die Werte Vorname, Nachname, Postleitzahl, Stadt, Straße und Hausnummer identisch sind.

Repositories

Repository: PaketRepository

UML:



Beschreibung:

Hält die Pakete, die im System sind, im Speicher und bietet Schnittstellen zur Weiterverarbeitung der Daten an.

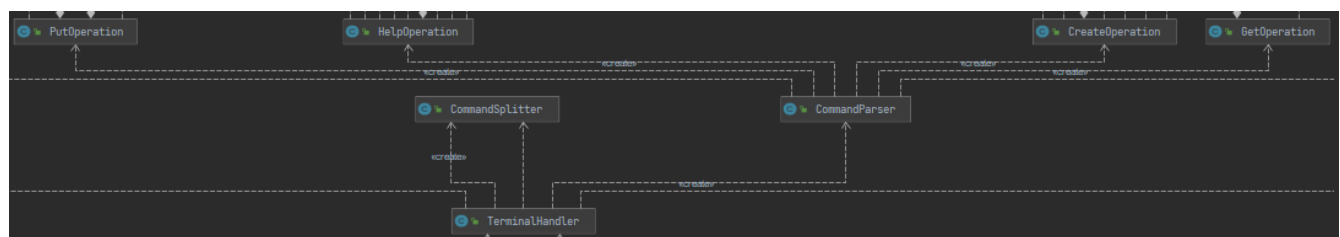
Begründung:

Durch den Einsatz von Repositories kann der Domain Code auf persistenten Speicher zugreifen, wo die Speichertechnologie dadurch vollständig verborgen ist. Außerdem bieten sie für die Domänen Abfragemöglichkeiten auf den Datenbestand.

Aggregates

Aggregate: TerminalHandling

UML:



Beschreibung:

Dieses Aggregate sorgt für das richtige behandeln einer Eingabe. Des Weiteren ruft es weitere Aggregates auf die der initialen Eingabe entsprechen.

Begründung:

Durch das Bilden einer Einheit werden Objektbeziehungen entkoppelt und bilden eine natürliche Transaktionsgrenze über die Aggregate-Root. Dadurch wird sichergestellt, dass der Zustand immer den Domänenregeln entspricht.

Kapitel 7: Refactoring

Code Smells

Duplicated Code:

Vorher:

```
if (command[POSITION].equalsIgnoreCase("CREATE") && command.length==3){
    new CreateOperation(command, packetEventsImpl);
}
else if (command[POSITION].equalsIgnoreCase("PUT") && command.length==3){
    new PutOperation(command, packetEventsImpl, employeeEventsImpl);
}
else if (command[POSITION].equalsIgnoreCase("GET") && command.length==3){
    new GetOperation(command, packetEventsImpl);
}
```

Nachher:

```
if (checkOperator(command, POSITION, "CREATE", 3)) {
    new CreateOperation(command, packetEventsImpl);
}
else if (checkOperator(command, POSITION, "PUT", 3)) {
    new PutOperation(command, packetEventsImpl, employeeEventsImpl);
}
else if (checkOperator(command, POSITION, "GET", 3)) {
    new GetOperation(command, packetEventsImpl);
}
```

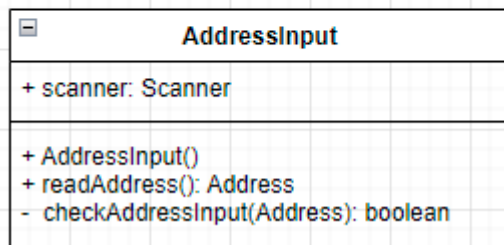
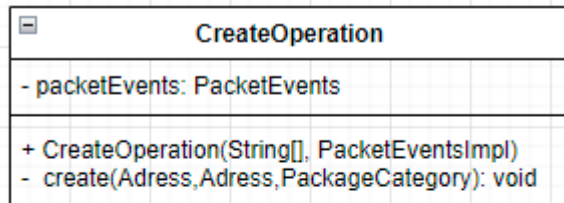
```
private boolean checkOperator(String[] commands, int commandPosition, String commandCheck, int length) {
    return commands[commandPosition].equalsIgnoreCase(commandCheck) && commands.length==length;
}
```

Um der Duplizierung entgegenzuwirken, wurde eine Funktion geschrieben die nur noch Aufgerufen werden muss. Damit wird ein ungewolltes Verhalten was den anderen Funktionen abweicht, vorgebeugt.

Long Methode:

CreateOperation

Erster Lösungsschritt Einlesen und Überprüfen der Daten für Absender und Empfänger in eigene Klasse Auslagern.



CreateOperation:

```
Createoperation(String[], PackeetEventsImpl):
```

```
    Input = new AddressInput()
```

```
    Address address = Input.readAddress()
```

```
    Create(address)
```

AddressInput:

```
    readAddress():
```

```
        Address Inputdata = FromCommandline
```

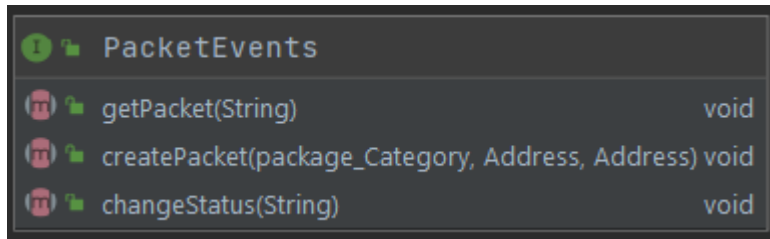
```
        If(checkAddressInput(Inputdata)) return Inputdata
```

2 Refactorings

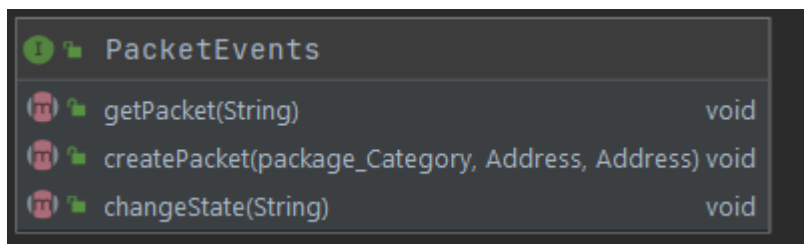
Refactoring 1: Rename Methode:

Begründung: In PacketEvents wurde changeStatus zu changeState geändert da es inkonsistent mit der Sprache war.

UML vorher:



UML nachher:



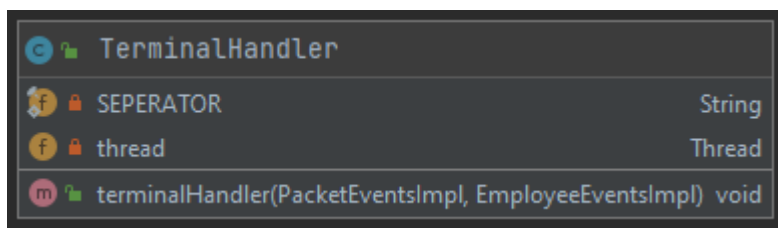
Commit:” refactor 1 renamed changeStatus to changeState”

Refactoring 2: Extract Methode:

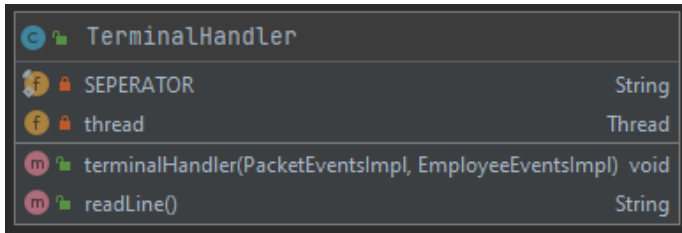
Klasse: TerminalHandler

Begründung: In TerminalHandler wurde das Einlesen einer Zeile mit BufferedReader in eine externe Methode ausgelagert damit zukünftige Erweiterungen diese Methode auch Aufrufen können und somit kein Code doppelt geschrieben werden muss.

UML vorher:



UML nachher:



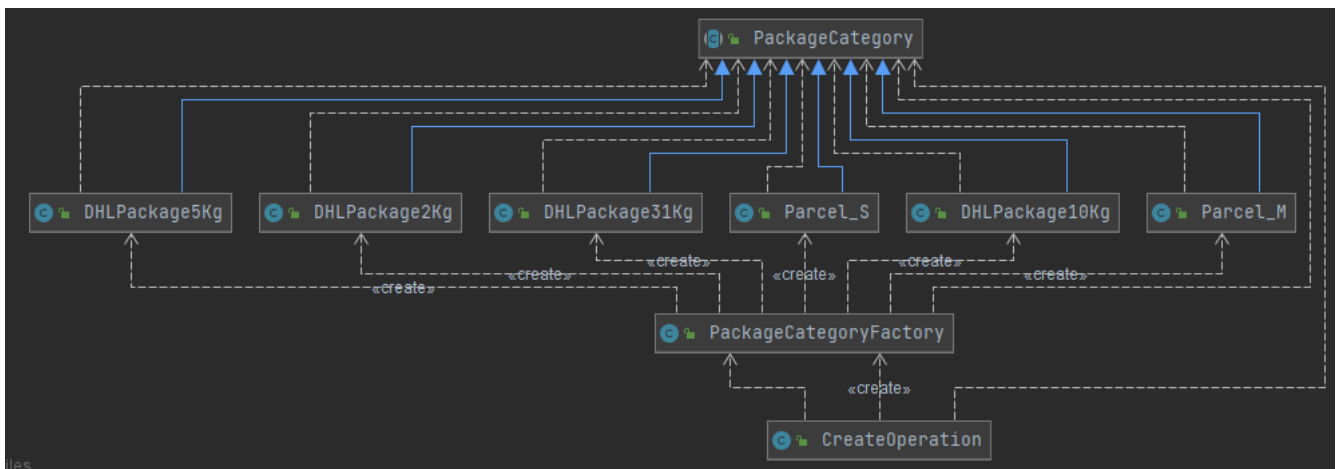
Commit: „Refactoring 2 extract methode“

Kapitel 8: Entwurfsmuster

Entwurfsmuster: [Factory]

Begründung: Mittels einer Factory Method erfolgt die Objekterzeugung über eine Methode und nicht über einen Konstruktor. Dadurch kann der Quellcode der normalerweise an eine Paketkategorie wie `Parcel_S` gekoppelt wäre unabhängig gestaltet werden. Dadurch muss die aufrufende Methode nicht wissen welchen Untertyp einer Instanz geliefert wird. Somit wird erst zur Laufzeit der konkrete Untertyp erzeugt und ist nicht vordefiniert.

UML:



Entwurfsmuster: [Einzelstück]

Begründung:

Durch dieses Pattern wird sichergestellt, dass es nur ein Objekt der Main-Klasse (packageTrackingManager) im gesamten Programm gibt.

UML:

