# Programmentwurf
# Paketmanager

Name: Nitz, Franz Josef
Martrikelnummer: 2791813

Abgabedatum: 29.05.2022

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

*Allgemeine Anmerkungen:*

- *es darf nicht auf andere Kapitel als Leistungsnachweis verwiesen werden (z.B. in der Form "XY wurde schon in Kapitel 2 behandelt, daher hier keine Ausführung")*
- *alles muss in UTF-8 codiert sein (Text und Code)*
- *sollten mündliche Aussagen den schriftlichen Aufgaben widersprechen, gelten die schriftlichen Aufgaben (ggf. an Anpassung der schriftlichen Aufgaben erinnern!)*
- *alles muss ins Repository (Code, Ausarbeitung und alles was damit zusammenhängt)*
- *die Beispiele sollten wenn möglich vom aktuellen Stand genommen werden*
    - *finden sich dort keine entsprechenden Beispiele, dürfen auch ältere Commits unter Verweis auf den Commit verwendet werden*
    - *Ausnahme: beim Kapitel "Refactoring" darf von vorne herein aus allen Ständen frei gewählt werden (mit Verweis auf den entsprechenden Commit)*
- *falls verlangte Negativ-Beispiele nicht vorhanden sind, müssen entsprechend mehr Positiv-Beispiele gebracht werden*
    - *Achtung: werden im Code entsprechende Negativ-Beispiele gefunden, gibt es keine Punkte für die zusätzlichen Positiv-Beispiele*
    - *Beispiele*
        - *"Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt."*
            - *Antwort: Es gibt keine Klasse, die SRP verletzt, daher hier 2 Klassen, die SRP einhalten: [Klasse 1], [Klasse 2]*
            - *Bewertung: falls im Code tatsächlich keine Klasse das SRP verletzt: volle Punktzahl ODER falls im Code mind. eine Klasse SRP verletzt: halbe Punktzahl*
- *verlangte Positiv-Beispiele müssen gebracht werden*
- *Code-Beispiel = Code in das Dokument kopieren*

# 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>*

Sendestatus zu einem Paket erfragen:

*GET PACKAGE <Sendeverfolgungsnummer>*

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

PUT PACKAGE <Sendeverfolgungsnummer>

Hilfe:

HELP

Paketkategorien herausfinden:

HELP CATEGORY

Mit dieser Software wird das Problem der Intransparenz 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.

*[Was macht die Applikation? Wie funktioniert sie? Welches Problem löst sie/welchen Zweck hat sie?]*

## Wie startet man die Applikation?

*[Wie startet man die Applikation? Welche Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]*

*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*

## Wie testet man die Applikation?

*[Wie testet man die Applikation? Welche Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]*

*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)

*[allgemeine Beschreibung der Clean Architecture in eigenen Worten]*

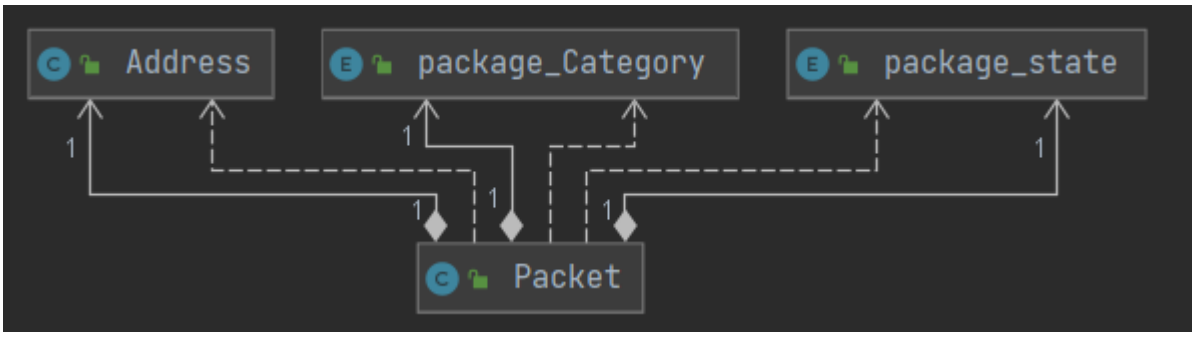
## Analyse der Dependency Rule

*[(1 Klasse, die die Dependency Rule einhält und eine Klasse, die die Dependency Rule verletzt); jeweils UML der Klasse und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]*

### *Positiv-Beispiel: Dependency Rule*

Klasse: Packet

UML:

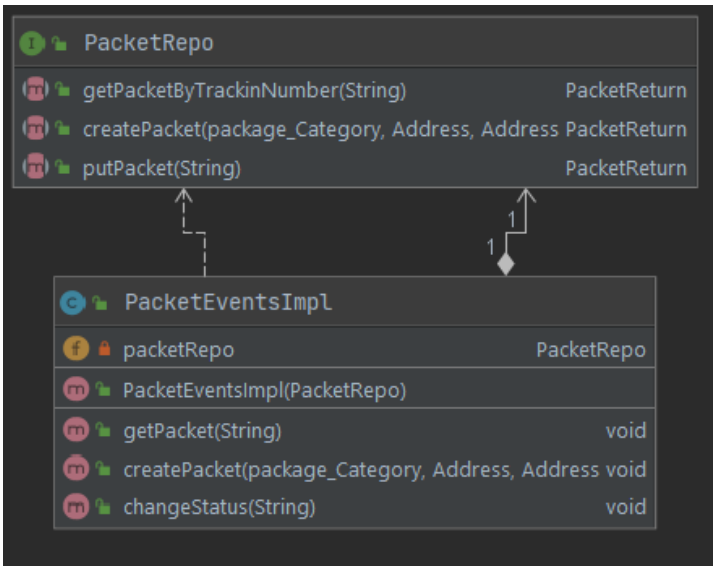



Analyse: Die Klasse Packet hat nur Abhängigkeiten innerhalb der Domain Schicht zu Address, package_Category und package_State und keine von innen nach außen. Aus diesem Grund hält Packet die Dependency Rule ein.

### *Negativ-Beispiel: Dependency Rule*

Klasse: PacketEventsImpl

UML:




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

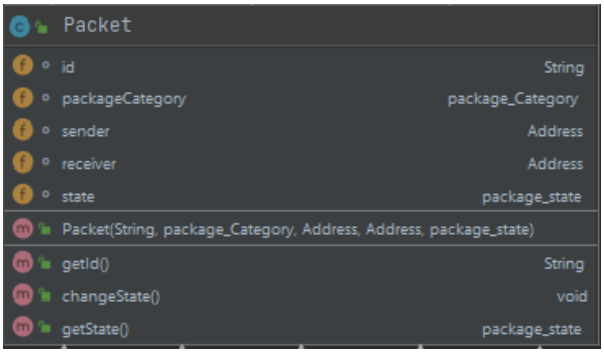
## Analyse der Schichten

*[jeweils 1 Klasse zu 2 unterschiedlichen Schichten der Clean-Architecture: jeweils UML der Klasse (ggf. auch zusammenspielenden Klassen), Beschreibung der Aufgabe, Einordnung mit Begründung in die Clean-Architecture]*

### *Schicht: [Domain-Code]*

Klasse: Packet

UML:



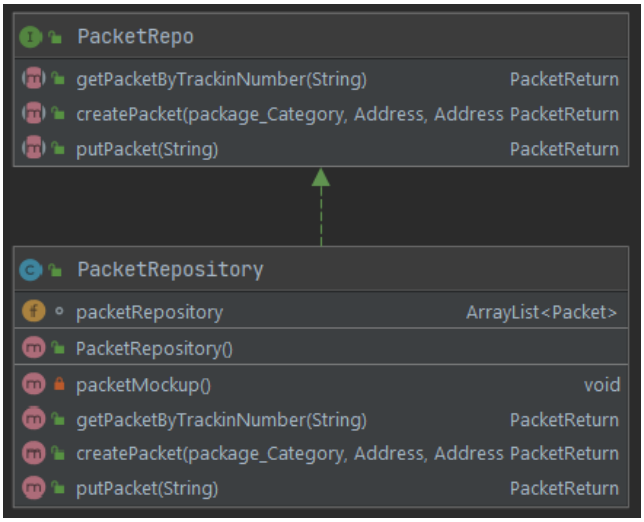

Aufgabe: Definieren der Funktionen und Variablen, die für ein Paket notwendig sind. Wird zur Paketobjekterzeugung verwendet.

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: PacketRepo

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)

*[jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML der Klasse und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]*

### *Positiv-Beispiel*

Klasse: CommandSplitter

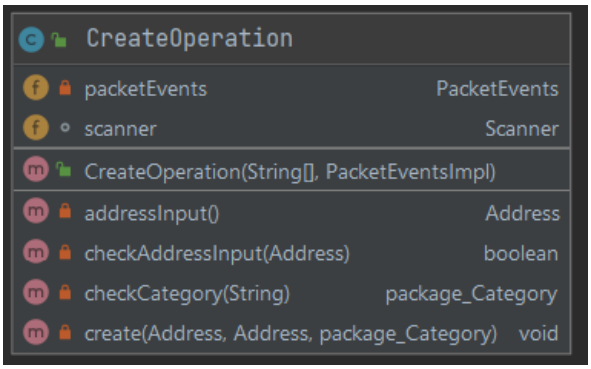
UML:


Aufgabe: Teilt den Eingabestring, der über die Kommandozeile eingegeben wurde, in einzelne Wörter über einen Separator (Leerzeichen) auf und gibt diese als Array zurück.

### *Negativ-Beispiel*

Klasse: CreateOperation

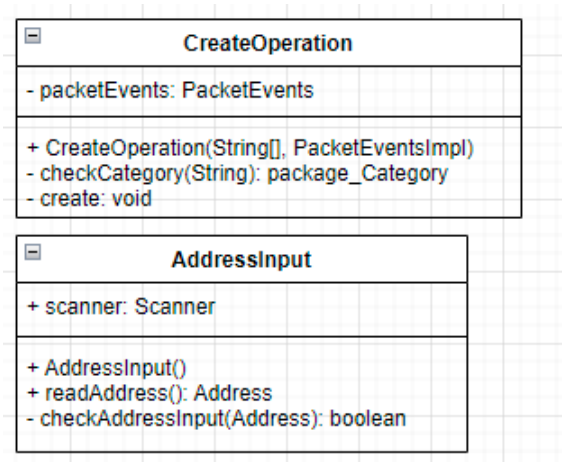
UML (vorher):




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 eigene Klasse Auslagern.

UML (nachher):



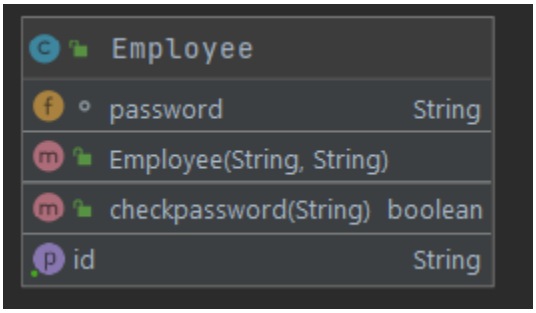
## Analyse Open-Closed-Principle (OCP)

*[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML der Klasse und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?]*

### *Positiv-Beispiel*

Klasse: Employee

UML:

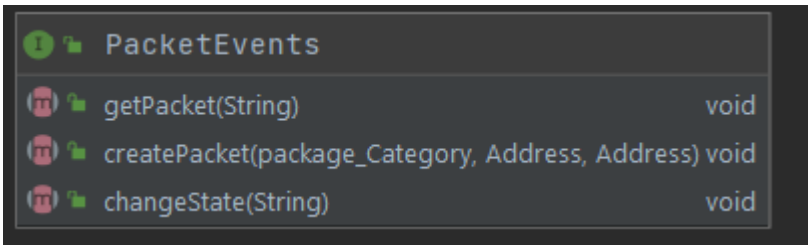



Begründung: Ein Mitarbeiter kann um weiter Attribute erweitert werden wie z.B. Vorname, Nachname, Standort, … . Dabei wird nur die Klasse Employee erweitert und keine weiteren Klassen verändert.

### *Negativ-Beispiel*

Klasse: PacketEvents

UML:




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

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

*[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP); jeweils UML der Klasse und Begründung, warum man hier das Prinzip erfüllt/nicht erfüllt wird]*

*[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]*

### *Positiv-Beispiel*

### *Negativ-Beispiel*

## Kapitel 4: Weitere Prinzipien

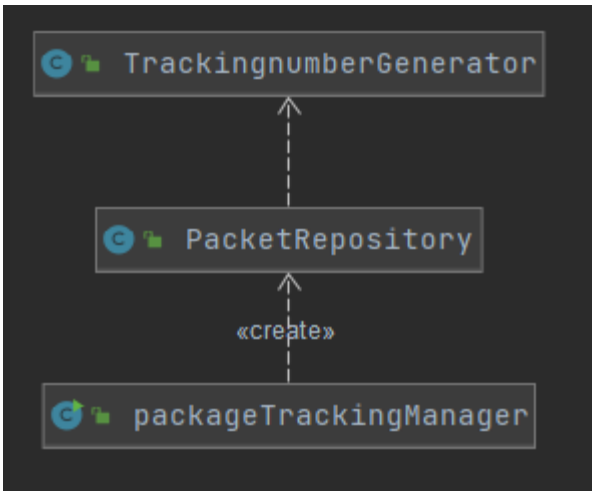
## Analyse GRASP: Geringe Kopplung

*[jeweils eine bis jetzt noch nicht behandelte Klasse als positives und negatives Beispiel geringer Kopplung; jeweils UML Diagramm mit zusammenspielenden Klassen, Aufgabenbeschreibung und Begründung für die Umsetzung der geringen Kopplung bzw. Beschreibung, wie die Kopplung aufgelöst werden kann]*

### *Positiv-Beispiel*

Klasse: PackageTrackinManager zu TrackingNumbergenerator

UML:

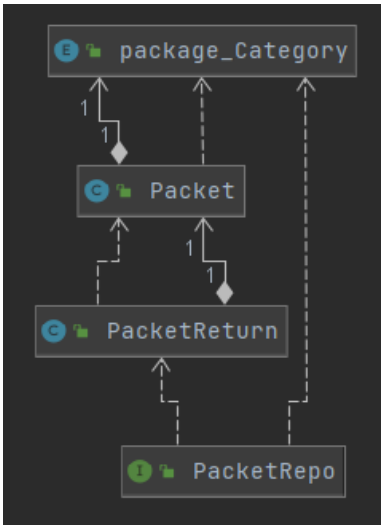



Begründung: zwischen diesen beiden Klassen besteht nur eine Abhängigkeit über die Klasse Package Repository.

### *Negativ-Beispiel*

Klasse: PacketRepo zu package_Category

UML:




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

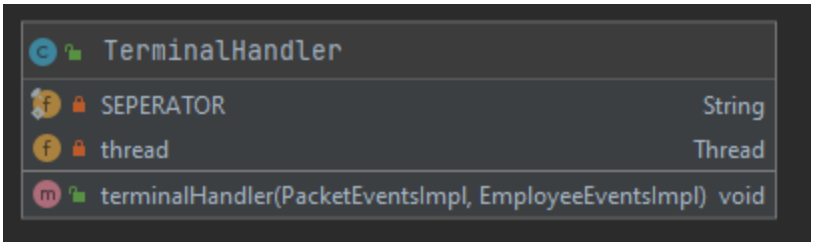
Lösung: Diese Abhängigkeit kann auch wie folgt modelliert werden: die Klasse PacketRepo hat eine Abhängigkeit zu PacketReturn diese wiederum zu Packet und die Klasse Paket hat dann die Abhängigkeit zu package_Category.

## Analyse GRASP: Hohe Kohäsion

*[eine Klasse als positives Beispiel hoher Kohäsion; UML Diagramm und Begründung, warum die Kohäsion hoch ist]*

*Klasse: TerminalHandler*

*UML:*




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

## Don’t Repeat Yourself (DRY)

*[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher); begründen und Auswirkung beschreiben]*

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 com-
mandCheck, int length){
    return commands[commandPosition].equalsIgnoreCase(commandCheck) && com-
mands.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

*[Nennung von 10 Unit-Tests und Beschreibung, was getestet wird]*

| Unit Test | Beschreibung |
|---|---|
| *PacketRepositoryTest # getElementByIdTest* | Überprüft ob mit dem gewählten Originalverfahren ein Element über die ID aus der ArrayList herausgefiltert werden kann, |
| *PacketRepositoryTest # addElementTest* | Prüfen ob Elemente in eine Liste hinzugefügt werden können, |
| *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#checkPass wordTest* | Prüft ob die Methode checkPassword mit richtigen eingaben funktioniert. |
| *EmployeeClassTes#checkpassw ordTestNegative* | Prüft ob falsche Passwörter erkannt werden und ein false zurückgegeben wird |

| | |
|---|---|
| *Klasse#Methode* | |
| *Klasse#Methode* | |
| *Klasse#Methode* | |
| *Klasse#Methode* | |

## ATRIP: Automatic

*[Begründung/Erläuterung, wie 'Automatic' realisiert wurde]*

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

## ATRIP: Thorough

*[jeweils 1 positives und negatives Beispiel zu 'Thorough'; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell ist]*

*Positiv:*

*Negativ: addElementTest ist nicht Thorough, da dies nicht notwendig ist zu testen, da es sich um eine vorgegebene Klasse (ArrayList) handelt und da klar ist dass es funktioniert.*

## ATRIP: Professional

*[jeweils 1 positives und negatives Beispiel zu 'Professional'; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell ist]*

*Positiv:*

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

## Code Coverage

Die Testabdeckung des Projektes ist sehr gering, da für eine hohe eine Vielzahl mehr Teste geschrieben hätten werden müssen.

## Fakes und Mocks

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

*[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten; zusätzlich jeweils UML Diagramm der Klasse]*

# Kapitel 6: Domain Driven Design

## Ubiquitous Language

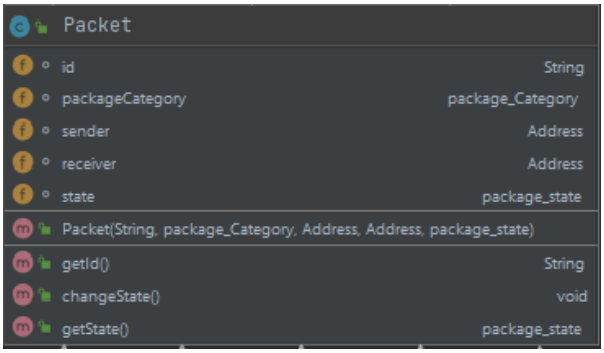
*[4 Beispiele für die Ubiquitous Language; jeweils Bezeichung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört]*

| Bezeichung | 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 für Entwickler als auch den Domänexperten diese Funktionalität klar |
| generateTrackingNumber | Damit soll eine Trackingnumber für ein Paket erstellt werden | Durch die Methodenbezeichnung wird die Funktionalität schnell klar, dass es sich hierbei um die Erzeugung 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. |
| getPacketByTrackingNumber | Soll ein Paket zu einer bestimmten Trackingnummer ausgeben | Damit wird für Entwickler und Domänexperten klar, dass ein Paket durch seine Trackingnummer angezeigt wird. Die Methodenbezeichnung ist hier klar formuliert. |

## Entities

Entität: Packet

UML:




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

Begründung:

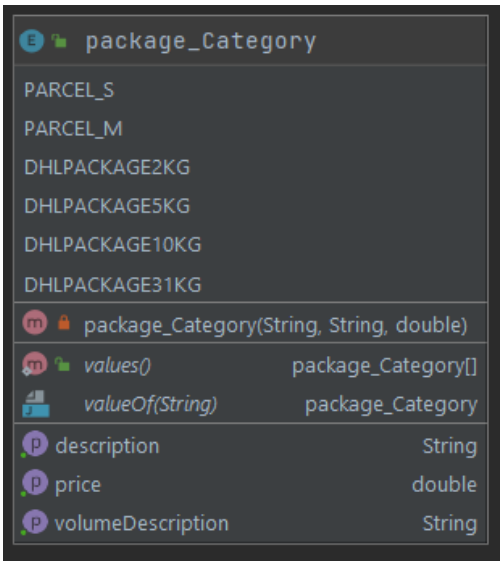
Weil ein Paket eine eindeutige Id besitzt und zwei Pakete nur gleich sind, wenn ihre Ids übereinstimmen.

*[UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]*

## Value Objects

Value Object: package_Category

UML:




Beschreibung: Definiert die Kategorien die es für die Pakete gibt.

Begründung:

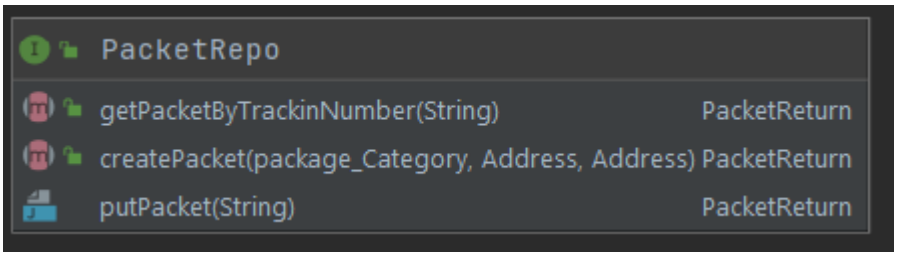
Eine Packetkategorie ist unveränderlich. Sie besteht aus einer Bezeichnung mit Gewichtsangabe, den Maßen und den Preis des Pakets. Dabei sind zwei Paketkategorien gleich wenn sie die gleichen Bezeichnungen, Maße und Preise besitzen.

*[UML, Beschreibung und Begründung des Einsatzes eines Value Objects; falls kein Value Object vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]*

## Repositories

Repository: PaketRepo

UML:




Beschreibung:

Begründung:

*[UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]*

## Aggregates

*[UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]*

# 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);
}
```

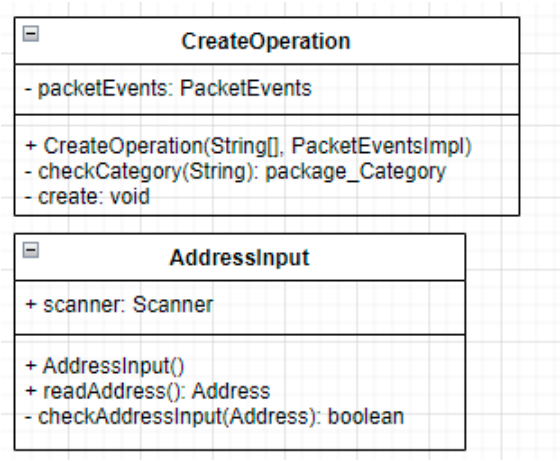
```
private boolean checkOperator(String[] commands,int commandPosition, String com-
mandCheck, int length){
    return commands[commandPosition].equalsIgnoreCase(commandCheck) && com-
mands.length==length;
}
```

Um die 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:

Createopertion(String[], PackeetEventsImpl):

Input = new AddressInput()

Address address = Input.readAddress()

Create(address)

AddressInput:

readAddress():

Address Inputdata = FromCommandline

If(checkAddressInput(Inputdata)) return Inputdata

*[jeweils 1 Code-Beispiel zu 2 Code Smells aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]*

## 2 Refactorings

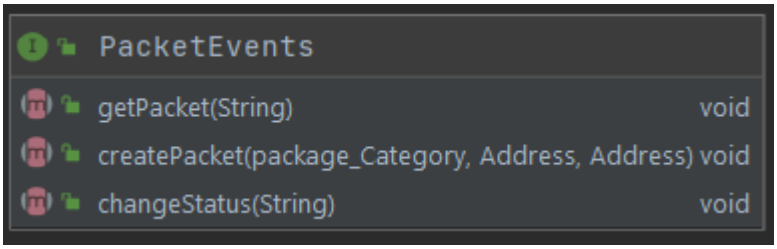
*[2 unterschiedliche Refactorings aus der Vorlesung anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen]*

*Refactoring 1:*

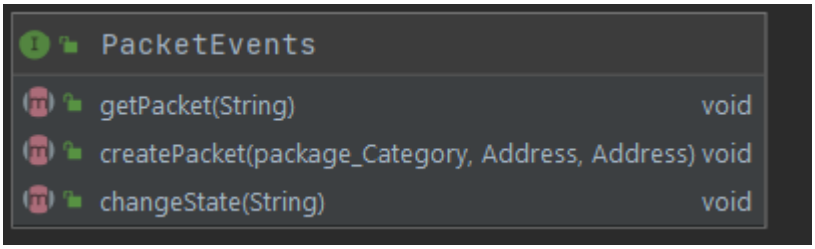
*Rename Methode:*

*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:

## Kapitel 8: Entwurfsmuster

*[2 unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils sinnvoll einsetzen, begründen und UML-Diagramm]*

### Entwurfsmuster: [Name]

### Entwurfsmuster: [Name]