Programmentwurf
Animal Crossing: New Horizons Hilfsprogramm

Name: Vollmer, Marina
Martrikelnummer: 9686417

Name: Nguyen, Vanessa
Martrikelnummer: 2884655

Abgabedatum: 29. Mai 2022

GitHub Repository

Kapitel 1: Einführung

Übersicht über die Applikation

Das Videospiel Animal Crossing: New Horizons ist eine Echtzeit-Lebenssimulation, bei der man mit seiner Spielfigur auf eine einsame Insel zieht und dort ein neues Leben aufbaut. Zu Beginn hat man die Wahl zwischen verschiedenen Inseln. Dabei ist entscheidend, ob man auf die Nord- oder Südhalbkugel zieht. Auf einer Insel der Nordhalbkugel orientiert sich der Verlauf der Jahreszeiten an den Jahreszeiten in Europa und umgekehrt.

Der Charakter kann nun seine Insel selbst gestalten. Nach und nach ziehen neue Nachbarn auf die Insel und der Charakter kann Freunde einladen oder selbst besuchen. Zudem kann er verschiedene Materialien sammeln, um daraus zu Dinge bauen. Darüber hinaus hat der Charakter die Möglichkeit, jegliche Gegenstände und Materialien zu verkaufen, um Sternis (Währung der Insel) dafür zu erhalten. Ebenso können Tiere und Pflanzen verkauft werden.

Hierfür kann der Charakter Fische angeln, Insekten fangen, nach Meerestieren tauchen sowie Pflanzen ernten. Relevant ist dabei vor allem die Jahreszeit sowie die Uhrzeit, manchmal aber auch das Wetter. Teilweise sind auch bestimmte Handlungen notwendig oder die Tiere können nur an festgelegten Orten gefunden werden. Jedes einzelne Tier hat verschiedene Voraussetzungen, die erfüllt sein müssen, damit der Charakter sie fangen kann.

Das Animal Crossing: New Horizons Hilfsprogramm kann das Fangen von Tieren auf der Insel erleichtern. Hierfür kann für jeden Charakter im Spiel ein Benutzer im Hilfsprogramm angelegt werden. Verknüpft mit diesem Benutzer ist ein Inventar, welches dem Spieler eine Übersicht über bereits gefangene Tiere gibt. Diesem Inventar können jederzeit neue Tiere hinzugefügt werden.

Neben der Inventar-Funktion bietet die Applikation die Möglichkeit, alle Tiere nach Jahreszeit, Halbkugel und Uhrzeit zu filtern. Diese Parameter kann der Benutzer eingeben, um anschließend Informationen über die fangbaren Tiere zu erhalten.

Wie funktioniert die Applikation?

1. Applikation nach Anleitung starten
2. Nach der Willkommens-Nachricht kann man seinen Benutzernamen eingeben, den man mit Enter bestätigen muss
3. Anschließend hat man die Wahl zwischen vier verschiedenen Funktionen:

    1: Tier zum Inventar hinzufügen

    2: Inventar anzeigen

    3: Was kann ich fangen?

    4: Programm beenden

4. Zahl zur Wahl der Funktion eingeben und mit Enter bestätigen

   bei 1: Name des Tiers zum Hinzufügen eingeben und mit Enter bestätigen

   bei 2: ist nichts weiter zu tun

   bei 3: Angaben vervollständigen und jeweils mit Enter bestätigen

   bei 4: ist nichts weiter zu tun

5. weiter bei 3.

Wie startet man die Applikation?

Man startet die Applikation, indem man innerhalb einer IDE (beispielsweise IntelliJ) die Klasse Start (Start.main()) ausführt.

Voraussetzungen hierfür sind:

- JDK 13
- JUnit 5.8.1

Schritt-für-Schritt-Anleitung:

1. Eine IDE (z.B. IntelliJ) öffnen, zuvor ggf. IDE herunterladen
2. Den Code des Programmentwurfs aus dem Repository clonen (Befehl: git clone https://github.com/marina99999/ASE-Animal_Crossing_NH.git)
3. Start.main() ausführen
4. Die Applikation startet in der Konsole der IDE
5. Anweisungen der Applikation befolgen

Wie testet man die Applikation?

Voraussetzungen hierfür sind:

- JDK 13
- JUnit 5.8.1

Schritt-für-Schritt-Anleitung:

1. Eine IDE (z.B. IntelliJ) öffnen, zuvor ggf. IDE herunterladen
2. Den Code des Programmentwurfs aus dem Repository clonen (Befehl: git clone https://github.com/marina99999/ASE-Animal_Crossing_NH.git)
3. Beliebige Test-Klasse (oder alle gleichzeitig) ausführen:

   In der IDE IntelliJ: indem man mit Rechtsklick auf die Test-Klasse und anschließend auf die entsprechende Run ‘Test’-Auswahl klickt

oder mit Rechtsklick auf ein gesamtes Package oder das gesamte Projekt alle Tests ausführt.

4. Die Tests starten in der IDE
5. Wahlweise kann auch “Run with coverage” gewählt werden, um zusätzlich die Test Coverage angezeigt zu bekommen

Kapitel 2: Clean Architecture

Was ist Clean Architecture?

Clean Architecture ist ein Architekturstil, bei dem ein technologieunabhängiger Kern, d.h. die eigentliche Anwendung, im Mittelpunkt steht. Dabei sind äußere Schichten, wie beispielsweise die Datenbank oder verschiedene Frameworks, vollständig entkoppelt davon. Die verschiedenen Schichten erinnern dabei an eine Zwiebel, daher wird Clean Architecture auch Onion Architecture genannt.

Der Sourcecode des Kerns ist langlebig, während in den äußeren Schichten, der Peripherie, kurzlebiger Code aufzufinden ist. Je konkreter dabei der Code wird, desto weiter außen sollte er liegen. Dadurch wird ermöglicht, dass äußere Schichten einfach austauschbar sind. Beispielsweise sollte es unkompliziert möglich sein, eine andere Datenbank oder Benutzeroberfläche einzusetzen und dabei die inneren Schichten völlig unberührt zu lassen. Dieses Phänomen wird durch die sogenannte Dependency Rule beschrieben. Abhängigkeiten bestehen also immer nur von äußeren zu weiter innen liegenden Schichten, nicht aber umgekehrt.

Dieses Vorgehen lohnt sich vor allem bei komplexeren Systemen oder Anwendungen. So kann der Kern entkoppelt von verschiedensten Technologien existieren und diese können bei Bedarf mit deutlich weniger Aufwand ausgetauscht werden. Zudem besteht der Vorteil, dass man sich erst spät für Technologien entscheiden muss und somit den “inneren” Teil der Anwendung bereits implementieren und testen kann.

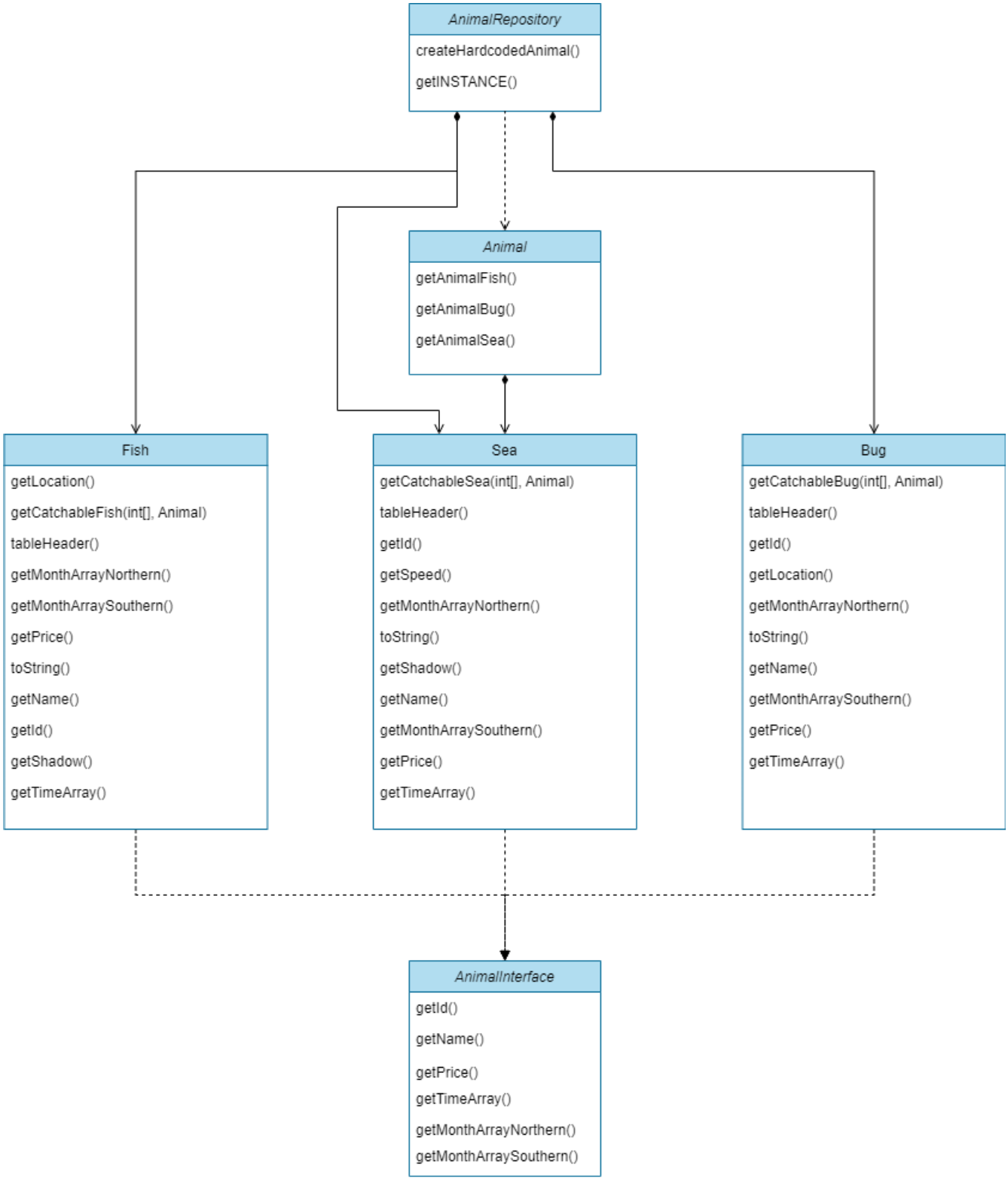
Analyse der Dependency Rule

Positiv-Beispiel: Dependency Rule

Klasse: AnimalRepository

Die Klasse AnimalRepository hält die Dependency Rule ein. Die Klassen Animal, Fish, Sea und Bug hängen von dem AnimalRepository ab, allerding ist die Klasse AnimalRepository von keiner anderen Klasse abhängig.

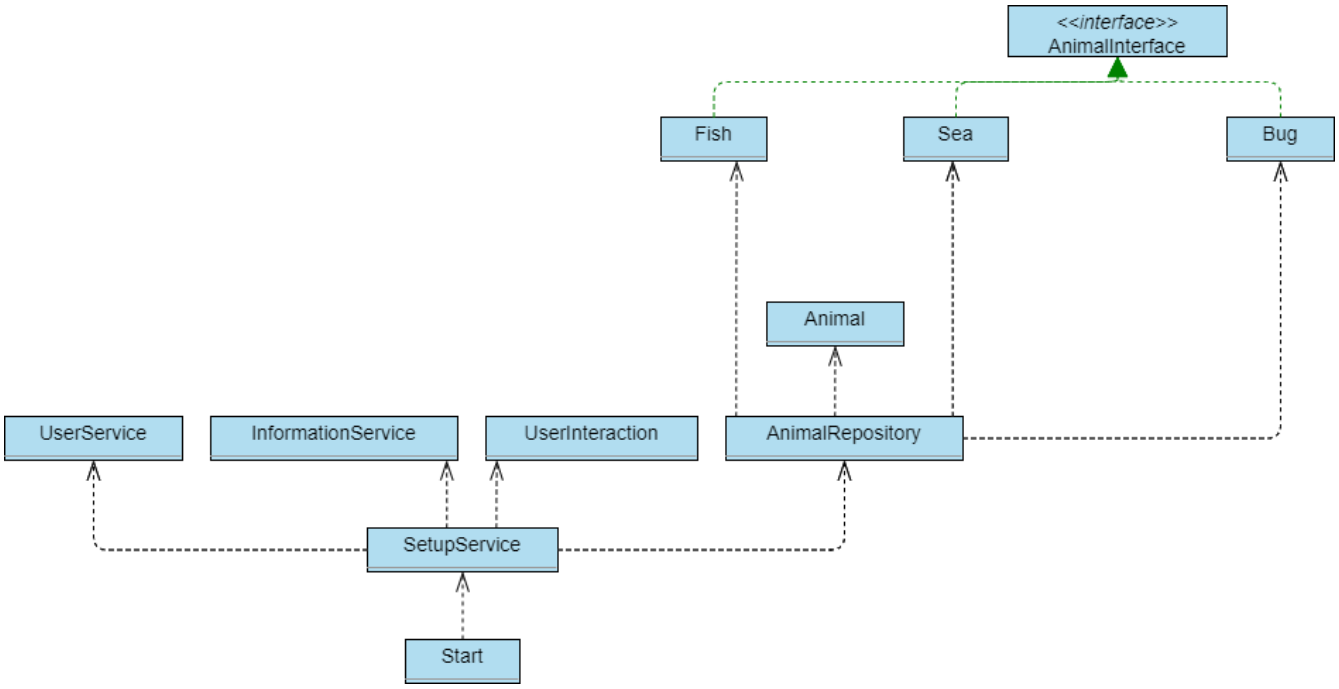
UML:



Negativ-Beispiel: Dependency Rule

Im nachfolgenden UML-Diagramm sind alle Dependencies des Programmentwurfs zu sehen. Insgesamt hält somit jede Klasse die Dependency Rule ein. Es existiert keine Abhängigkeit

von innen nach außen, die Abhängigkeiten existieren immer nur in eine Richtung (von außen nach innen). Der Zentrale Punkt der Anwendung, die Animals, sind von nichts anderem abhängig, während darauf aufbauend Klassen wie beispielsweise UserInteraction oder SetupService, weiter außen liegen und ausgetauscht/verändert werden könnten, ohne, dass dabei Klassen wie z.B Fish berührt werden.

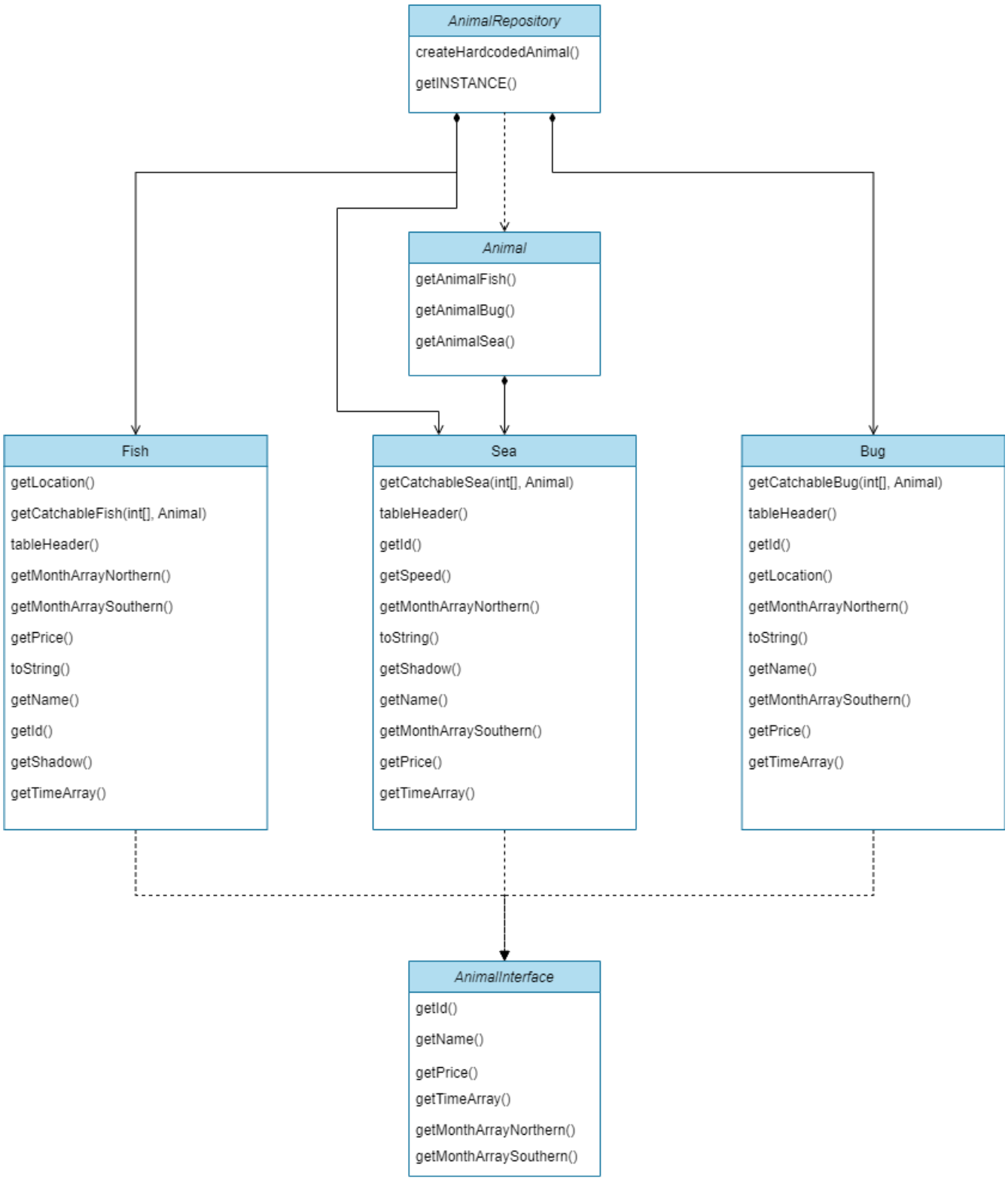


Analyse der Schichten

Schicht: Domain Code

Klasse: AnimalRepository

UML:



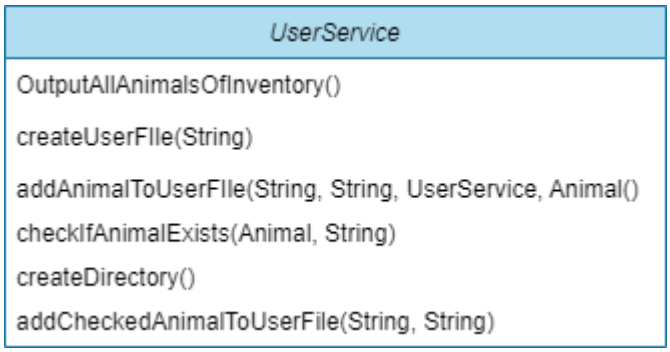
Aufgabe: Das AnimalRepository dient der Persistierung der Tiere. Es wird ein beispielhafter Auszug aller Tiere, die in ACNH existieren, als Datensatz genutzt. Somit gibt die Klasse ein paar Tiere hart codiert zurück.

Einordnung: Das AnimalRepository stellt mit den Animal-Klassen den Kern unserer Anwendung dar. Dementsprechend ist der Code in der Schicht des Domain Codes, also in der innersten Schicht, einzuordnen.

Schicht: Application Code

Klasse: UserService

UML:



Aufgabe: Die Klasse UserService dient allen Aufgaben rund um den Benutzer und dessen Inventar. Sie erstellt ein Verzeichnis, in dem anschließend die von ihr erzeugten User Files abgelegt werden. In diesen User Files werden später die zugehörigen Inventare gespeichert. Daher dient die Klasse zusätzlich der Überprüfung, ob ein Tier existiert und ob dieses noch nicht im Inventar gespeichert wurde. Ist dies beides der Fall, wird das Tier von der Klasse in das User File geschrieben und somit dem Inventar hinzugefügt. Letztendlich gibt die Klasse auch den Inhalt eines User Files, also alle Tiere des Inventars, zurück.

Einordnung: In der Applikationsschicht liegen die Services unserer Anwendung, welche die Use-Cases implementieren. Dazu gehört auch der UserService, welcher beispielsweise von dem AnimalRepository, d.h. der darunterliegenden Domain Schicht, abhängig ist. Gemäß der Grundprinzipien der Clean Architecture liegt somit der Code der Services, welcher konkreter ist, weiter außen in der Architektur.

Kapitel 3: SOLID

Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel

Klasse: SetupService

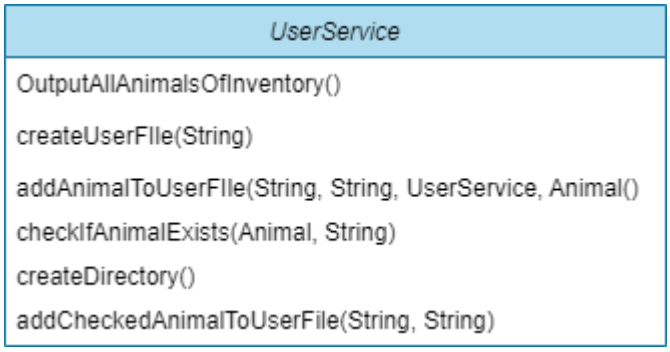
UML:

Aufgabe: Die Klasse SetupService dient dem ersten Aufsetzen aller benötigten Objekte und besitzt daher nur eine einzige Funktion, die firstSetup-Methode. Innerhalb dieser wird z.B. eine Instanz des AnimalRepositorys erstellt. Zudem wird auch die erste Konsolen-Nachricht an den User ausgegeben, indem die dafür zuständige Funktion der Klasse UserInteraction aufgerufen wird.

Negativ-Beispiel

Klasse: UserService

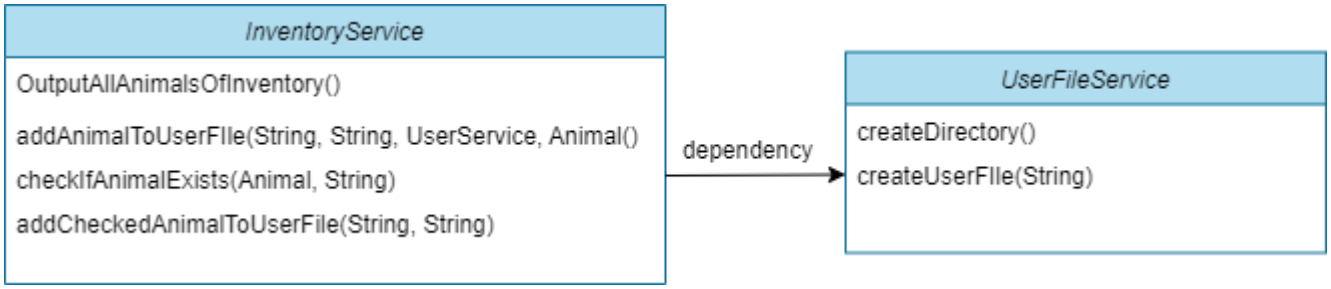
UML:



Aufgabe: Die Klasse UserService stellt jegliche Funktionen, die der User benötigt um sein Inventar zu erstellen sowie zu verwalten, bereit. Dadurch ist das SRP verletzt, da es zum einen Funktionen gibt, die dem Erstellen eines Benutzer-Inventars dienen und zum anderen Funktionen, die dem Hinzufügen von Tieren zum Inventar oder dem Ausgeben dessen dienen.

Lösung: Um das SRP einzuhalten, macht es Sinn, die Klasse UserService in eine Klasse InventoryService und in eine Klasse UserFileService aufzuteilen. Der InventoryService stellt dann alle Funktionen bereit, welche zur Verwaltung und zur Ausgabe der Inventare benötigt werden. In der Klasse UserFileService befinden sich die Methoden um UserFiles und ein Verzeichnis zur Speicherung dieser zu erstellen. Somit können die Aufgaben durch die Auslagerung von Methoden thematisch aufgeteilt werden. Dies erhöht insgesamt die Wiederverwendbarkeit des Codes.

UML Lösung:

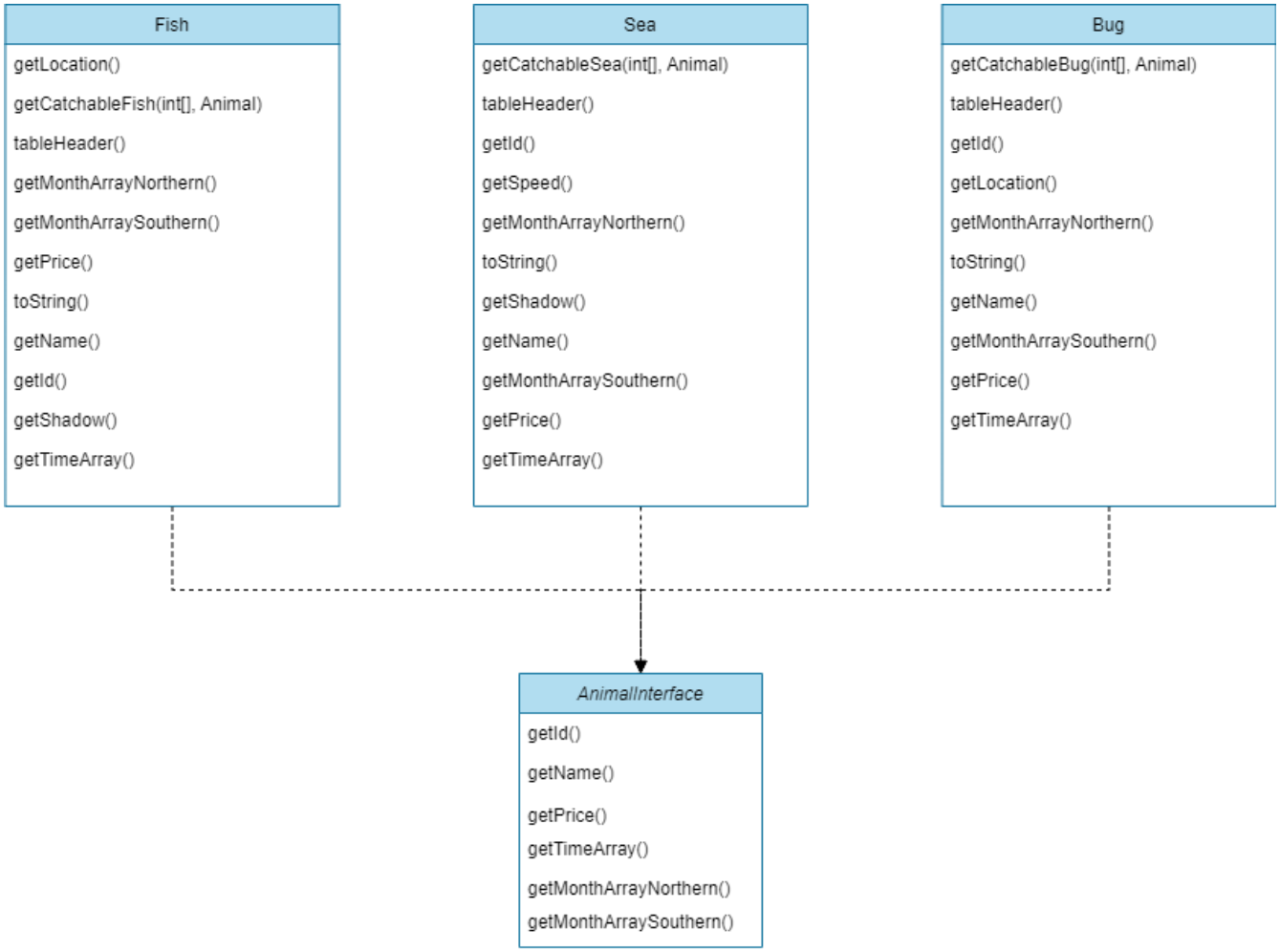


Analyse Open-Closed-Principle (OCP)

Positiv-Beispiel

Klasse: AnimalInterface mit Bug, Fish, Sea

UML:



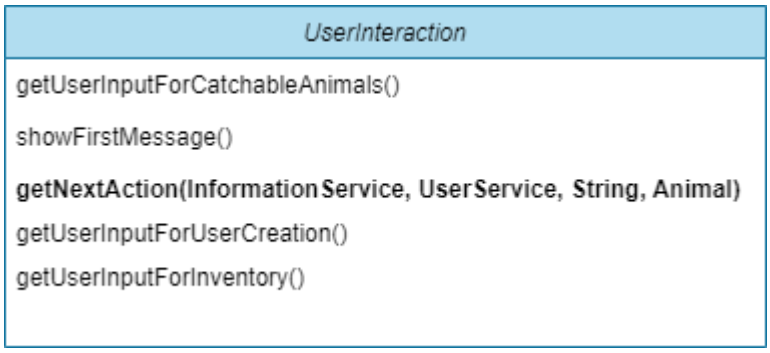
Begründung: Dadurch, dass die Ausgabe der jeweiligen Tiere auf verschiedene Klassen aufgeteilt ist, ist das OCP erfüllt. Eine Erweiterung um weitere Tierarten ist ohne Probleme möglich (offen für Erweiterungen), aber die Modifikation der Tierarten ist nicht vorgesehen.

Warum hier sinnvoll/welche Probleme gab es?
Würde man die Ausgaben der Tiere, also die Funktionen toString und tableheader innerhalb einer einzigen Klasse (innerhalb des AnimalInterfaces) realisieren, wäre eine einfache Erweiterung nicht mehr möglich. Dadurch, dass aber für jede Tierart eine andere Ausgabe nötig ist, macht es Sinn, diese Methoden in den Klassen Bug, Fish und Sea aufzuteilen bzw. erst dort zu implementieren.

Negativ-Beispiel

Klasse: UserInteraction (konkret die Funktion: getNextAction)

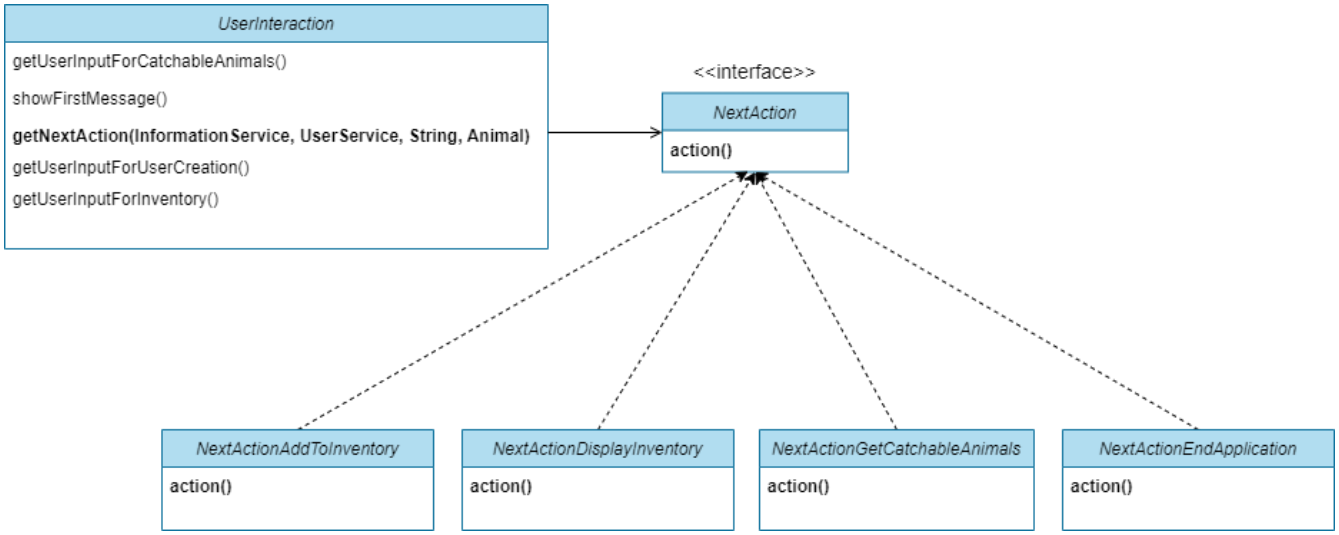
UML:



Begründung: Die Funktion getNextAction der Klasse UserInteraction verletzt das OCP, da eine Erweiterung der Funktionalität nicht einfach möglich ist. Sollte beispielsweise die Möglichkeit, ein Tier aus dem Inventar zu entfernen, hinzugefügt werden, müsste hierfür ein neuer “else if”-Block angehängt werden. Auf diese Weise wird der Code der Funktion immer länger und unübersichtlicher. Zudem dauert eine Erweiterung der Funktion sehr lange.

Lösung: Die Funktion getNextAction müsste in verschiedene Funktionen aufgeteilt werden, es muss also eine Abstraktion mithilfe eines Interfaces stattfinden. Die nächstes Aktion soll anschließend in der jeweils dafür zuständigen Klasse aufgerufen werden, und nicht wie zuvor in der UserInteraction Klasse.

UML:



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

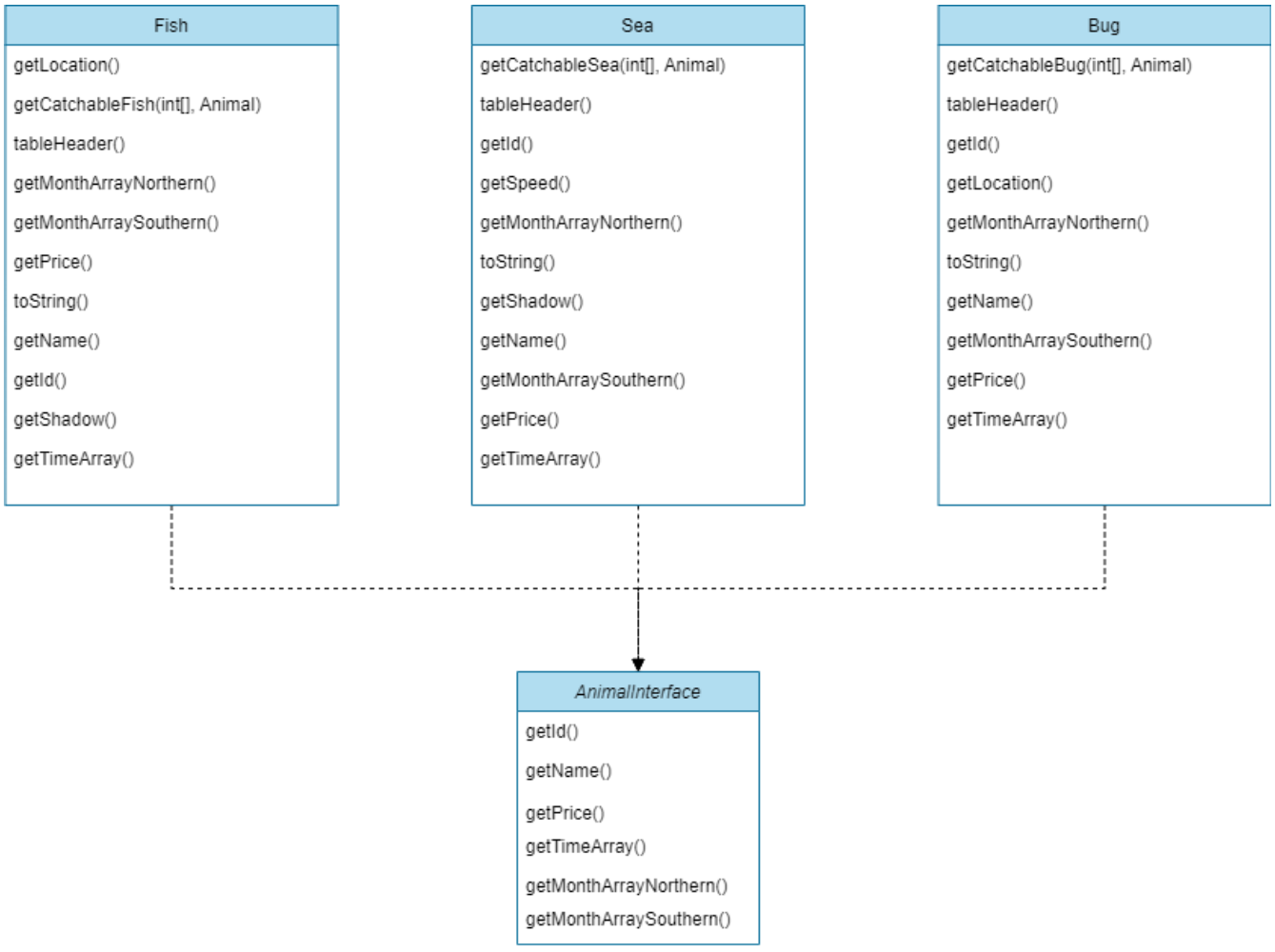
Negativ-Beispiel ISP

Klasse: AnimalInterface + Bug, Fish, Sea

Commit:

- es handelt sich um Code aus einem älteren Stand, siehe Commit: d4ae8897a1b7a596808a2221d1e43a3680079d1e

UML:

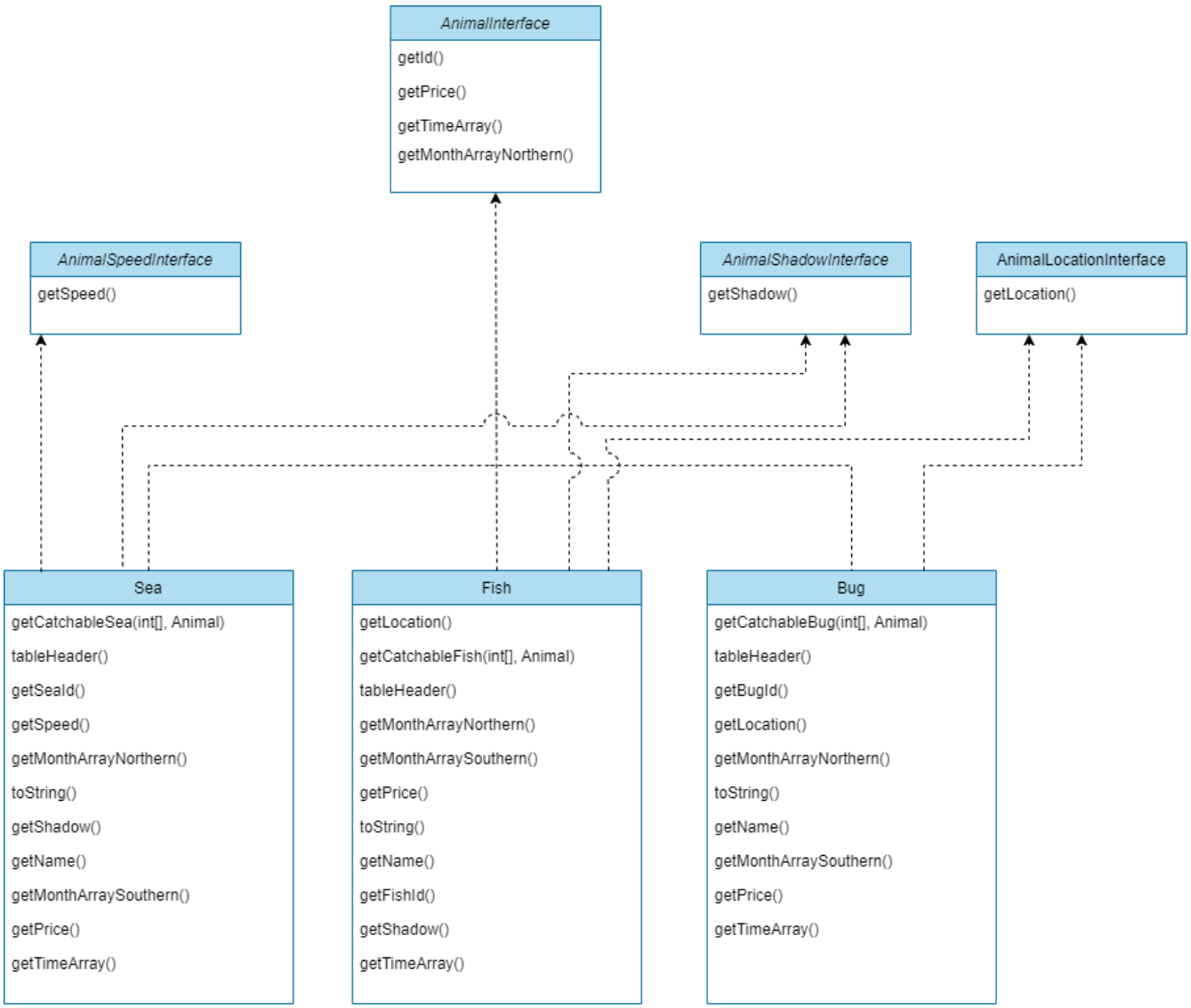


Begründung: Das AnaimalInterface ist ein allgemeines Interface und beschreibt lediglich die Basis-Methoden, die jede TIerart auf jeden Fall besitzt. Spezifische Methoden, die dennoch bei mehreren Tierarten genutzt werden, sind in dem AnaimalInterface nicht enthalten und müssen immer neu implementiert werden. Dies führt zu vermeidbarem Mehraufwand.

Positiv-Beispiel ISP

Klasse: AnimalInterface, AnimalLocationInterface, AnimalSpeedInterface, AnimalShadowInterface, Fish, Bug, Sea

UML:



Begründung: Da das AnimalInterface nun in vier verschiedene Interfaces aufgeteilt wurde, ist der Code modularer und einfacher wartbar. Neue Tierarten können mit weniger Aufwand unter Nutzung der verschiedenen Interfaces eingefügt werden. Zudem sind die Aufgaben klarer verteilt.

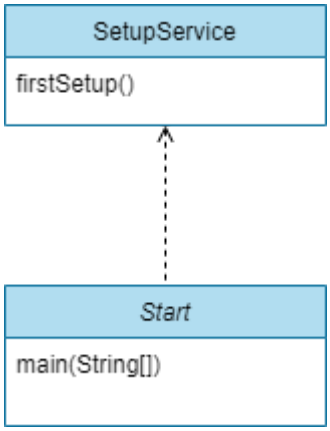
Kapitel 4: Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

Positiv-Beispiel

Klasse: Start

UML:



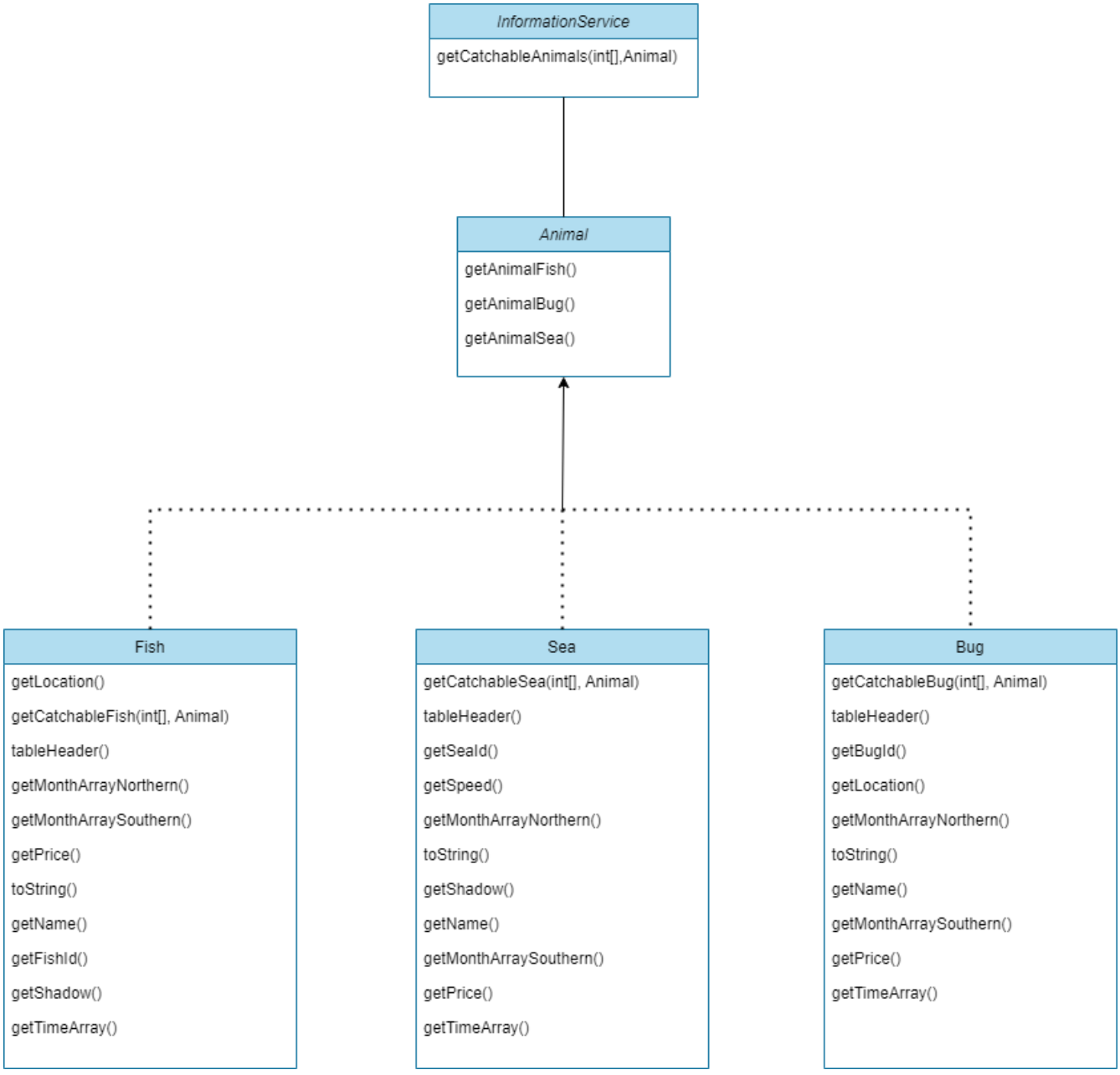
Aufgabenbeschreibung: Die Klasse Start dient zur Zeit lediglich dem Aufruf des SetupService und ist somit der Einstiegspunkt der Applikation.

Begründung: Start ist grundsätzlich an keine anderen Klassen gekoppelt, da die Klasse immer den zentralen EInstiegspunkt darstellen soll. Somit finden hier auch keine Änderungen mehr statt. Konkrete Änderungen die den Start des Programms betreffen, sollen im SetupService oder in anderen Klassen vorgenommen werden. Somit hätten diese auch keine Auswirkung auf die Klasse Start.

Negativ-Beispiel

Klasse: InformationService

UML:



Aufgabenbeschreibung: Die Klasse InformationService ruft die jeweiligen Funktionen auf, die dazu zuständig sind, aus der Benutzereingabe eine Ausgabe aller fangbaren Tiere zu generieren.

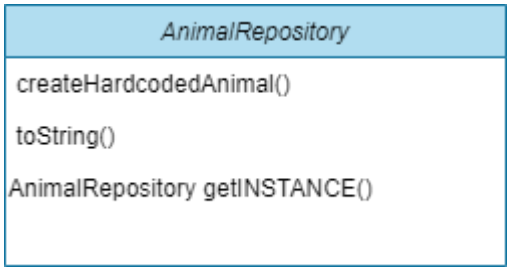
Lösung: Abstraktion der Klassen Bug, Fish und Sea mit einem Interface Animal, sodass die konkrete Funktion, die die fangbaren Insekten, Fische, Meerestiere zurückzugeben, nicht an die Klasse selbst gekoppelt ist. Der InformationService hängt somit nur noch vom Interface

Animal ab, dieses wird selten geändert. Konkrete Änderungen in den Klassen Bug, Fish und Sea haben keine Auswirkungen auf die Klasse InformationService.

Analyse GRASP: Hohe Kohäsion

Klasse: AnimalRepository

UML:



Begründung: Die Klasse AnimalRepository ist nur für die Instanziierung der hart codierten Tiere zuständig. Die drei Listen werden in allen (in diesem Fall einer) Methode genutzt. Somit ist die Kohäsion sehr hoch.

Don’t Repeat Yourself (DRY)

Commit:

- d4ae8897a1b7a596808a2221d1e43a3680079d1e
- remove duplicated code (time + month arrays)

Code-Beispiele:

Vorher:

```
public Animal createHardcodedAnimal() {
   Fish bitterling = new Fish(1, "Bitterling", "winzig(1)", "Fluss", 900,
new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23}, new int[]{11, 12, 1, 2, 3}, new int[]{5, 6, 7,
8, 9});
   Fish doebel = new Fish(2, "Döbel", "winzig(1)", "Fluss", 200, new
int[]{9, 10, 11, 12, 13, 14, 15}, new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12}, new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12});
   animalFish.add(bitterling);
   animalFish.add(doebel);

   Bug kohlweissling = new Bug(1, "Kohlweißling", "fliegend", 160, new
```

```
int[]{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}, new int[]{9,
10, 11, 12, 1, 2, 3, 4, 5, 6}, new int[]{3, 4, 5, 6, 7, 8, 9, 10, 11,
12});
   Bug zitronenfalter = new Bug(2, "Zitronenfalter", "fliegend", 160, new
int[]{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}, new int[]{3,
4, 5, 6, 9, 10}, new int[]{9, 10, 11, 12, 3, 4});
   animalBug.add(kohlweissling);
   animalBug.add(zitronenfalter);

   Sea wakameAlge = new Sea(1, "Wakame-Alge", "groß", 600,
"stillstehend", new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23}, new int[]{10, 11, 12, 1, 2, 3,
4, 5, 6, 7}, new int[]{4, 5, 6, 7, 8, 9, 10, 11, 12, 1});
   Sea kriechsprossalge = new Sea(2, "Kriechsprossalge", "klein", 900,
"stillstehend", new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23}, new int[]{6, 7, 8, 9}, new
int[]{12, 1, 2, 3});
   animalSea.add(wakameAlge);
   animalSea.add(kriechsprossalge);

   Animal hardcodedAnimal = new Animal(animalFish, animalBug, animalSea);
   return hardcodedAnimal;
}
```

Nachher:

```
private final int[] timeArrayWholeDay = new int[]{0, 1, 2, 3, 4, 5, 6, 7,
8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23};
private final int[] monthArrayWholeYear = new int[]{1, 2, 3, 4, 5, 6, 7,
8, 9, 10, 11, 12};

public Animal createHardcodedAnimal() {
   Fish bitterling = new Fish(1, "Bitterling", "winzig(1)", "Fluss", 900,
timeArrayWholeDay, new int[]{11, 12, 1, 2, 3}, new int[]{5, 6, 7, 8, 9});
   Fish doebel = new Fish(2, "Döbel", "winzig(1)", "Fluss", 200, new
int[]{9, 10, 11, 12, 13, 14, 15}, monthArrayWholeYear,
monthArrayWholeYear);
   animalFish.add(bitterling);
   animalFish.add(doebel);
```

```
   Bug kohlweissling = new Bug(1, "Kohlweißling", "fliegend", 160, new
int[]{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}, new int[]{9,
10, 11, 12, 1, 2, 3, 4, 5, 6}, new int[]{3, 4, 5, 6, 7, 8, 9, 10, 11,
12});
   Bug zitronenfalter = new Bug(2, "Zitronenfalter", "fliegend", 160, new
int[]{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}, new int[]{3,
4, 5, 6, 9, 10}, new int[]{9, 10, 11, 12, 3, 4});
   animalBug.add(kohlweissling);
   animalBug.add(zitronenfalter);

   Sea wakameAlge = new Sea(1, "Wakame-Alge", "groß", 600,
"stillstehend", timeArrayWholeDay, new int[]{10, 11, 12, 1, 2, 3, 4, 5,
6, 7}, new int[]{4, 5, 6, 7, 8, 9, 10, 11, 12, 1});
   Sea kriechsprossalge = new Sea(2, "Kriechsprossalge", "klein", 900,
"stillstehend", timeArrayWholeDay, new int[]{6, 7, 8, 9}, new int[]{12,
1, 2, 3});
   animalSea.add(wakameAlge);
   animalSea.add(kriechsprossalge);

   Animal hardcodedAnimal = new Animal(animalFish, animalBug, animalSea);
   return hardcodedAnimal;
}
```

Begründung/Auswirkung: Dadurch, dass sich die Arrays, in denen die Zeiten sowie Monate zum Fangen der Tiere angegeben sind, bei verschiedenen Tieren wiederholen und es so oft zu Duplikaten der Arrays kommt, macht es Sinn, die Arrays in globalen (final) Variablen auszahlungen. Dadurch können diese bei neu hinzukommenden Tieren im AnimalRepository genutzt werden, wodurch viel Zeit und Speicherplatz eingespart werden kann.

Sofern neue Tiere hinzugefügt werden, muss erneut geprüft werden, ob sich weitere Arrays (Zeitspannen oder Monats-Spannen) wiederholen. Diese sollten zusätzlich zu timeArrayWholeDay und monthArrayWholeYear als final Array eingeführt werden.

Kapitel 5: Unit Tests

10 Unit Tests

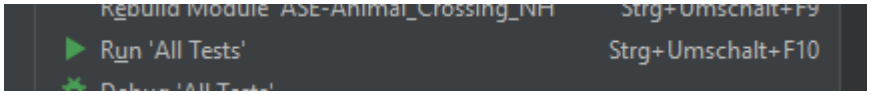
Unit Test	Beschreibung
AnimalTest#getAnimalFish	Es wird getestet, dass: - die Größe der testAnimal-Liste genau 1 ist, also genau der eine Fisch, der vorgesehen ist, hinzugefügt wurde

	- die fishID des ersten (und einzigen) Fisches in der testAnimal-Liste 1 ist - der Name “Test-Fisch” in der Liste existiert - die String-Ausgabe (toString) des ersten Fisches in der testAnimal-Liste mit der erwarteten Ausgabe des Test-Fisches übereinstimmt → äquivalent werden getAnimalSea und getAnimalBug getestet
BugTest#getCatchableBug	Es wird getestet, dass: - die Größe die zurückgegebene Liste der Funktion getCatchableBug genau 1 ist, also genau das Insekt testBug gefangen werden kann - die zurückgegebene Liste der Funktion getCatchableBug das Bug testBug enthält
FishTest#getShadow	Es wird getestet, dass: - der Schatten des Test-Fisches genau “winzig(1)” entspricht
SeaTest#getTimeArray	Es wird getestet, dass: - das TimeArray des Test-Meerestieres, in dem die Zeit gespeichert wird, innerhalb welcher das Meerestier fangbar ist, gleich des erwarteten Arrays “{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23}” ist
AnimalRepositoryTest#testCreateHardcodedAnimal	Es wird getestet, dass: - das zurückgegebene “hardcodedAnimal” not null ist - das zurückgegebene “hardcodedAnimal” ein Objekt der Klasse Animal ist
InformationServiceTest#getCatchableAnimals	Es wird getestet, dass: - die Konsolenausgabe für den vorgegebenen userInput die erwarteten Strings zu den fangbaren Tieren enthält -
UserInteractionTest#showFirstMessage	Es wird getestet, dass: - das Programm ordnungsgemäß mit der ersten Konsolen-Ausgabe an den Benutzer (“Willkommen zum Animal Crossing: New Horizons Hilfsprogramm”) startet
UserInteractionTest#getUserInputForCatchableAnimals	Es wird getestet, dass: - der zurückgegebene Array des Mock-Objektes tatsächlich dem erwarteten Array {1, 5, 20} entspricht
UserInteractionTest#getUserInputForInventory	Es wird getestet, dass: - das UserInteractionMock bei dem Aufruf der Methode getUserInputForInventory() den erwarteten String “Bitterling” zurückgibt

UserServiceTest#OutputAllAnimalsOfInventory	Es wird getestet, dass: - bei dem Aufruf der Funktion mit dem User “Testuser” ein “Testoutput” des Mock-Objektes auf der Konsole ausgegeben wird.

ATRIP: Automatic

Die eigenschaft Automatic wurde dadurch realisiert, dass alle Tests des Programm-Verzeichnisses innerhalb der IDE IntelliJ ausgeführt werden können, ohne weitere Maßnahmen durchzuführen.



Ist dies in anderen IDEs nicht möglich, besteht die Möglichkeit, eine Test Suite im Projekt einzuführen. Mit der Ausführung dieser Test Suite Klasse als JUnit Test werden alle Tests des Projektes ausgeführt.
Neben diesen Möglichkeiten könnte die Ausführung der Tests in einer GitHub Pipeline eingebunden werden, damit die Tests vollautomatisch ausgeführt werden können - dies wurde jedoch nicht innerhalb des Programmentwurfs umgesetzt.

Bei der Ausführung aller Tests müssen keine manuellen Eingaben erfolgen, zudem gibt es nur die Ergebnisse Success oder Failure - beides zählt zur Anforderung Automatic.

ATRIP: Thorough

Positiv-Beispiel: InformationService.getCatchableAnimals()

Code:

```
public void getCatchableAnimals(int[] userInput, Animal hardcodedAnimal){
   hardcodedAnimal.getAnimalFish().get(1).getCatchableFish(userInput,
hardcodedAnimal);
   hardcodedAnimal.getAnimalSea().get(1).getCatchableSea(userInput,
hardcodedAnimal);

hardcodedAnimal.getAnimalBug().get(1).getCatchableBug(userInput,hardcoded
Animal);
}
```

Analyse und Begründung: Hier werden 100% der Code-Zeilen überprüft. Somit ist der Test vollständig und die A-TRIP EIgenschaft Thorough ist erfüllt.

Negativ-Beispiel: Fish.getCatchableFish()

Code:

```
public List getCatchableFish(int[] userInput, Animal animalList) {
   tableheader();
   List<Fish> catchableFish = new ArrayList<>();
   List<Fish> checkTime = new ArrayList<>();
   for (Fish fish : animalList.getAnimalFish()) {
       boolean contains = IntStream.of(fish.getTimeArray()).anyMatch(x ->
x == userInput[2]);
       if (contains) {
           checkTime.add(fish);
       }
   }

   if (userInput[0] == 1) {
       for (Fish fish : checkTime) {
           boolean contains =
IntStream.of(fish.getMonthArrayNorthern()).anyMatch(x -> x ==
userInput[1]);
           if (contains) {
               catchableFish.add(fish);
           }
       }
   } else if (userInput[0] == 2) {
       for (Fish fish : checkTime) {
           boolean contains =
IntStream.of(fish.getMonthArraySouthern()).anyMatch(x -> x ==
userInput[1]);
           if (contains) {
               catchableFish.add(fish);
           }
       }
   }
   for (Fish fish : catchableFish) {
       System.out.println(fish.toString());
   }
   return catchableFish;
}
```

Analyse und Begründung:
Der Test der Klasse Fish testet den rot markierten else if-Fall der Methode nicht. Dementsprechend wird nicht alles notwendige getestet. Der Fall, dass ein Nutzer auf der Südhalbkugel lebt und die fangbaren Fische erfahren möchte, wird somit überhaupt nicht getestet. Dies widerspricht der A-TRIP Eigenschaft Thorough, da der Test somit nicht vollständig ist.

ATRIP: Professional

Positiv-Beispiel: AnimalRepositoryTest.testCreateHardcodedAnimal()

Code:

```
private List<Fish> animalFish = new ArrayList<>();
private List<Bug> animalBug = new ArrayList<>();
private List<Sea> animalSea = new ArrayList<>();
private Animal testHardcodedAnimal;
private Animal hardcodedAnimal;

@BeforeEach
void setUp() {
   testHardcodedAnimal = new Animal(animalFish, animalBug, animalSea);
   AnimalRepository testRepository = new AnimalRepository();
   hardcodedAnimal = testRepository.createHardcodedAnimal();
}

@Test
void testCreateHardcodedAnimal() {
   assertNotNull(hardcodedAnimal);
   assertInstanceOf(testHardcodedAnimal.getClass(), hardcodedAnimal);
}
```

Analyse und Begründung: Der Test testCreateHardcodedAnimal() erfüllt die Eigenschaft Professional, weil er die Standards von gutem Code erfüllt. Sofern neue Tierarten hinzukommen sollten, ist auch der Test einfach erweiterbar. Zudem wird durch das @BeforeEach eine Code-Duplikation verhindert und die Methode setUp() kann bei potenziellen Erweiterungen der Klasse ebenfalls verwendet werden, was das Prinzip DRY erfüllt. Auch Prinzipien wie GRASP und SOLID werden befolgt.

Negativ-Beispiel: BugTest.getBugId()

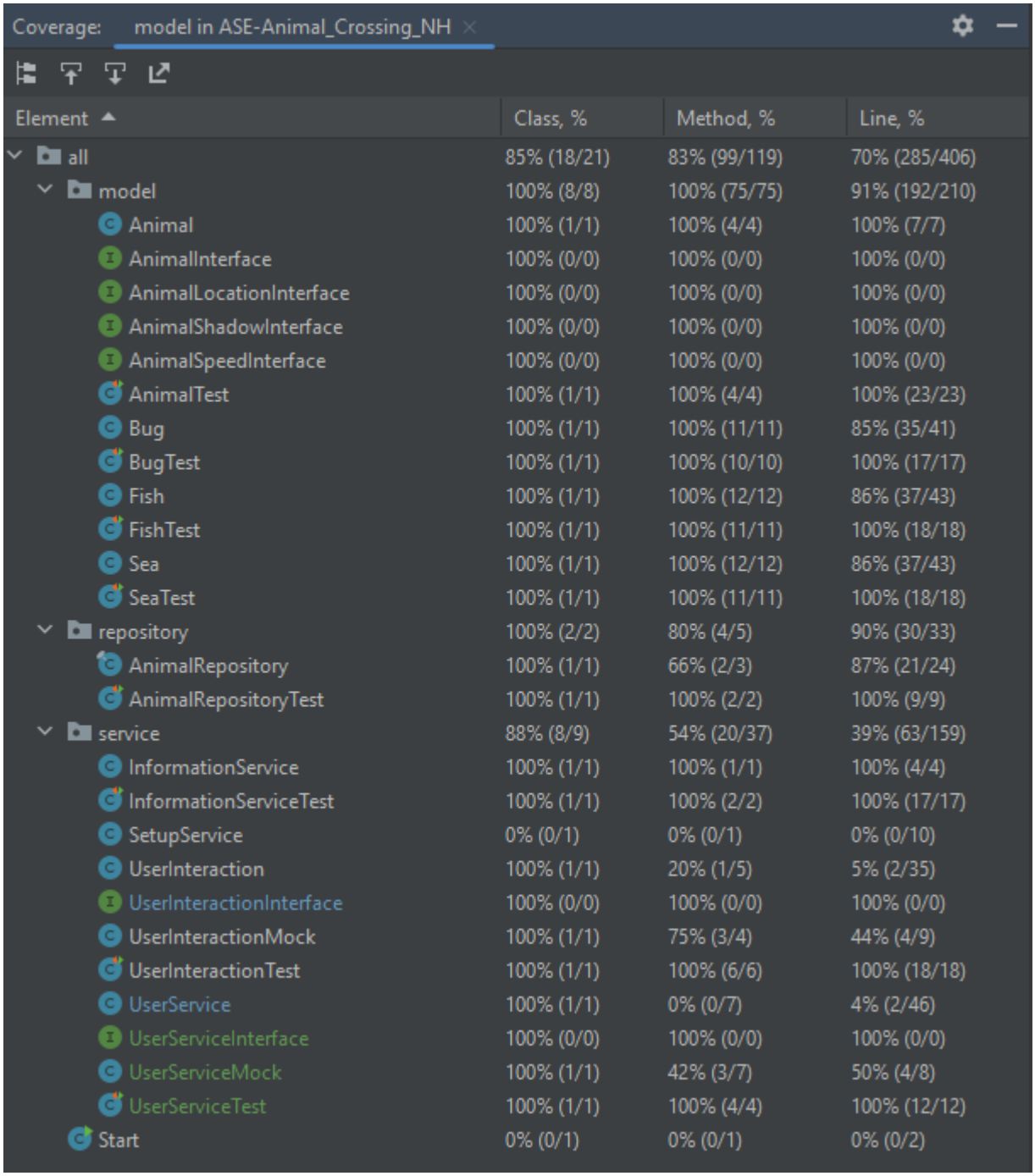
Code:

```
@Test
void getBugId() {
   assertEquals(1, testAnimal.getAnimalBug().get(0).getId());
}
```

Analyse und Begründung:
Das Testen von Gettern wie “getBugId()” ist eigentlich nicht notwendig und stellt daher unnötigen Code dar. Da aber Tests hinsichtlich der Code-Qualität äquivalent zu Produktivcode behandelt werden sollen, führt der Test eines Getters eher zu einer verschlechterung der Qualität. Aus diesem Grund sollte der Test (und ähnliche Tests von Getter-Methoden) aus dem Programmentwurf entfernt werden.

Code Coverage

Code Coverage Report aus IntelliJ:



Element	Class, %	Method, %	Line, %
all	85% (18/21)	83% (99/119)	70% (285/406)
model	100% (8/8)	100% (75/75)	91% (192/210)
Animal	100% (1/1)	100% (4/4)	100% (7/7)
AnimalInterface	100% (0/0)	100% (0/0)	100% (0/0)
AnimalLocationInterface	100% (0/0)	100% (0/0)	100% (0/0)
AnimalShadowInterface	100% (0/0)	100% (0/0)	100% (0/0)
AnimalSpeedInterface	100% (0/0)	100% (0/0)	100% (0/0)
AnimalTest	100% (1/1)	100% (4/4)	100% (23/23)
Bug	100% (1/1)	100% (11/11)	85% (35/41)
BugTest	100% (1/1)	100% (10/10)	100% (17/17)
Fish	100% (1/1)	100% (12/12)	86% (37/43)
FishTest	100% (1/1)	100% (11/11)	100% (18/18)
Sea	100% (1/1)	100% (12/12)	86% (37/43)
SeaTest	100% (1/1)	100% (11/11)	100% (18/18)
repository	100% (2/2)	80% (4/5)	90% (30/33)
AnimalRepository	100% (1/1)	66% (2/3)	87% (21/24)
AnimalRepositoryTest	100% (1/1)	100% (2/2)	100% (9/9)
service	88% (8/9)	54% (20/37)	39% (63/159)
InformationService	100% (1/1)	100% (1/1)	100% (4/4)
InformationServiceTest	100% (1/1)	100% (2/2)	100% (17/17)
SetupService	0% (0/1)	0% (0/1)	0% (0/10)
UserInteraction	100% (1/1)	20% (1/5)	5% (2/35)
UserInteractionInterface	100% (0/0)	100% (0/0)	100% (0/0)
UserInteractionMock	100% (1/1)	75% (3/4)	44% (4/9)
UserInteractionTest	100% (1/1)	100% (6/6)	100% (18/18)
UserService	100% (1/1)	0% (0/7)	4% (2/46)
UserServiceInterface	100% (0/0)	100% (0/0)	100% (0/0)
UserServiceMock	100% (1/1)	42% (3/7)	50% (4/8)
UserServiceTest	100% (1/1)	100% (4/4)	100% (12/12)
Start	0% (0/1)	0% (0/1)	0% (0/2)

Das Testen der Klassen Start und SetupService ist laut A-TRIP nicht notwendig und wurde daher nicht umgesetzt.
Zudem macht es keinen Sinn, einzelne triviale Methoden, wie beispielsweise Getter oder Setter zu testen, weshalb dies teilweise auch nicht implementiert wurde.
Insgesamt ist daher die Code Coverage nicht bei 100%, dennoch wurden für die relevanten Methoden Unit Tests geschrieben.

Teilweise sind Randfälle noch nicht in den Tests abgedeckt, weshalb diese noch ergänzt werden müssen, um die Eigenschaft Thorough zu erfüllen.
Die Eigenschaften Repeatable und Independent wurden erfüllt.

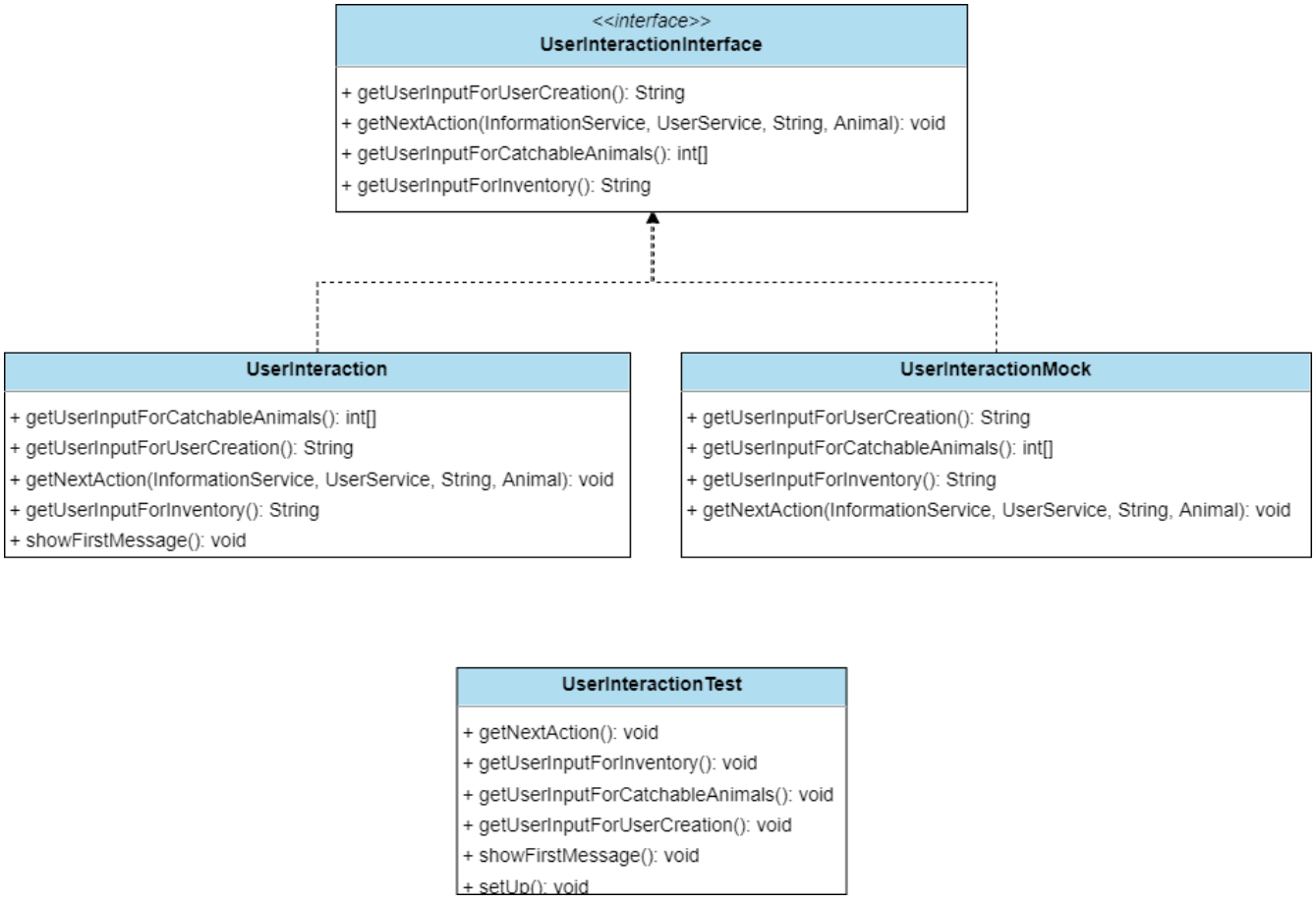
Fakes und Mocks

Mock 1:

Klasse: UserInteraction + UserInteractionMock

Begründung: Für das Testen der Klasse UserInteraction macht es Sinn, ein Mock-Objekt zu benutzen. Daher wurde zunächst das Interface UserInteractionInterface erstellt. Daraufhin wurde die Klasse UserInteractionMock erstellt, die das UserInteractionInterface implementiert. Statt den tatsächlichen Methoden, die auf Benutzereingaben in der Konsole angewiesen sind, geben die Mock-Methoden Standard-Werte (ohne eine Eingabe zu benötigen) zum Testen zurück.

UML:

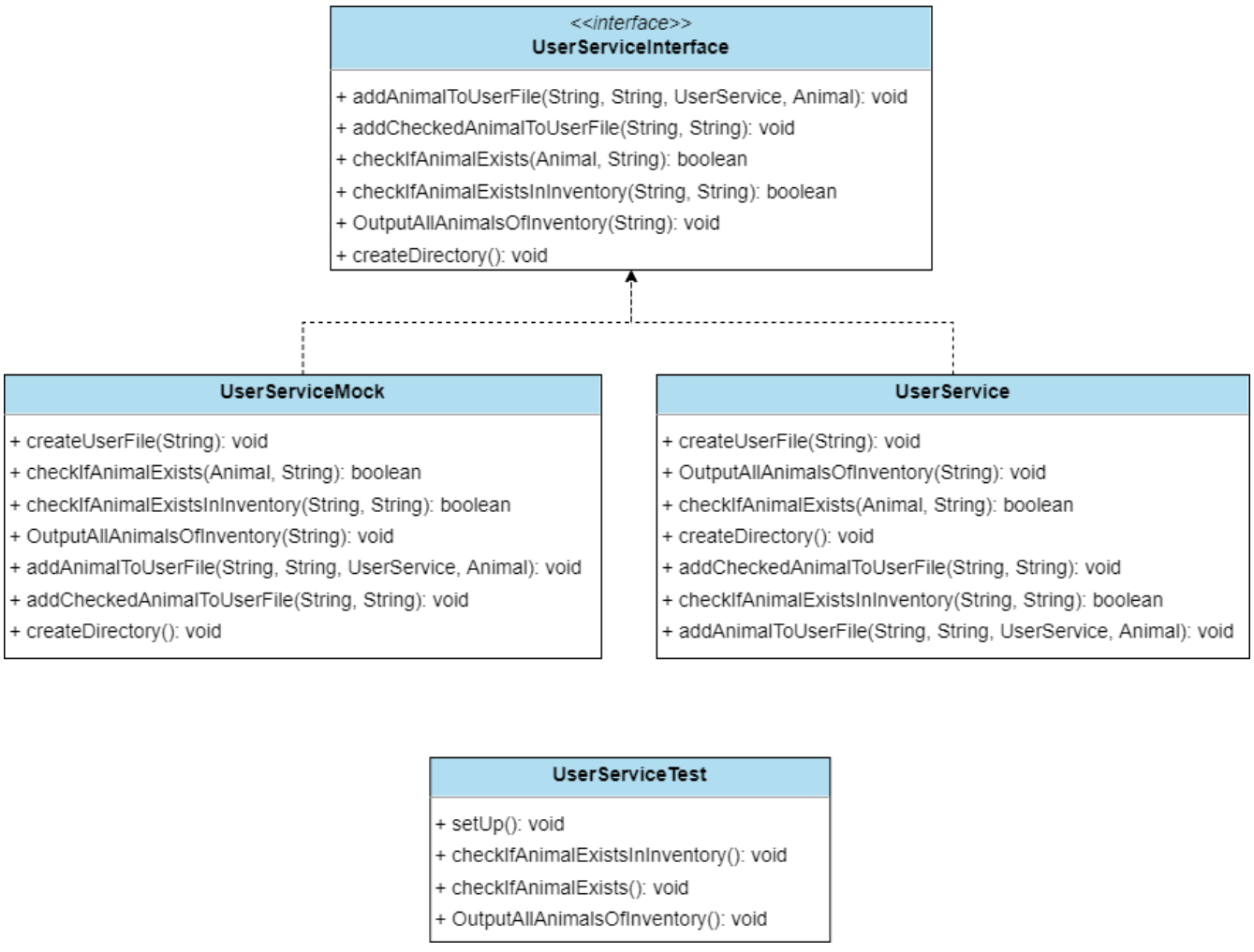


Mock 2:

Klasse: UserService + UserServiceMock

Begründung: Für das Testen der Klasse UserService ist es ebenfalls hilfreich, ein Mock-Objekt zu benutzen. Daher wurde äquivalent zum ersten Mock das Interface UserServcieInterface erstellt. Daraufhin wurde die Klasse UserServiceMock erstellt, die das UserServiceInterface implementiert. Statt den tatsächlichen Methoden, die auf Benutzereingaben in der Konsole angewiesen sind, geben die Mock-Methoden Standard-Werte (ohne eine Eingabe zu benötigen) zum Testen zurück. Ohne das Mock-Objekt wäre beispielsweise das Testen der Funktion checkIfAnimalExists() nicht einfach möglich, da die Methode auf eine Konsoleneingabe eines Tier-Namens angewiesen ist. Mit dem Mock-Objekt kann man dies umgehen.

UML:



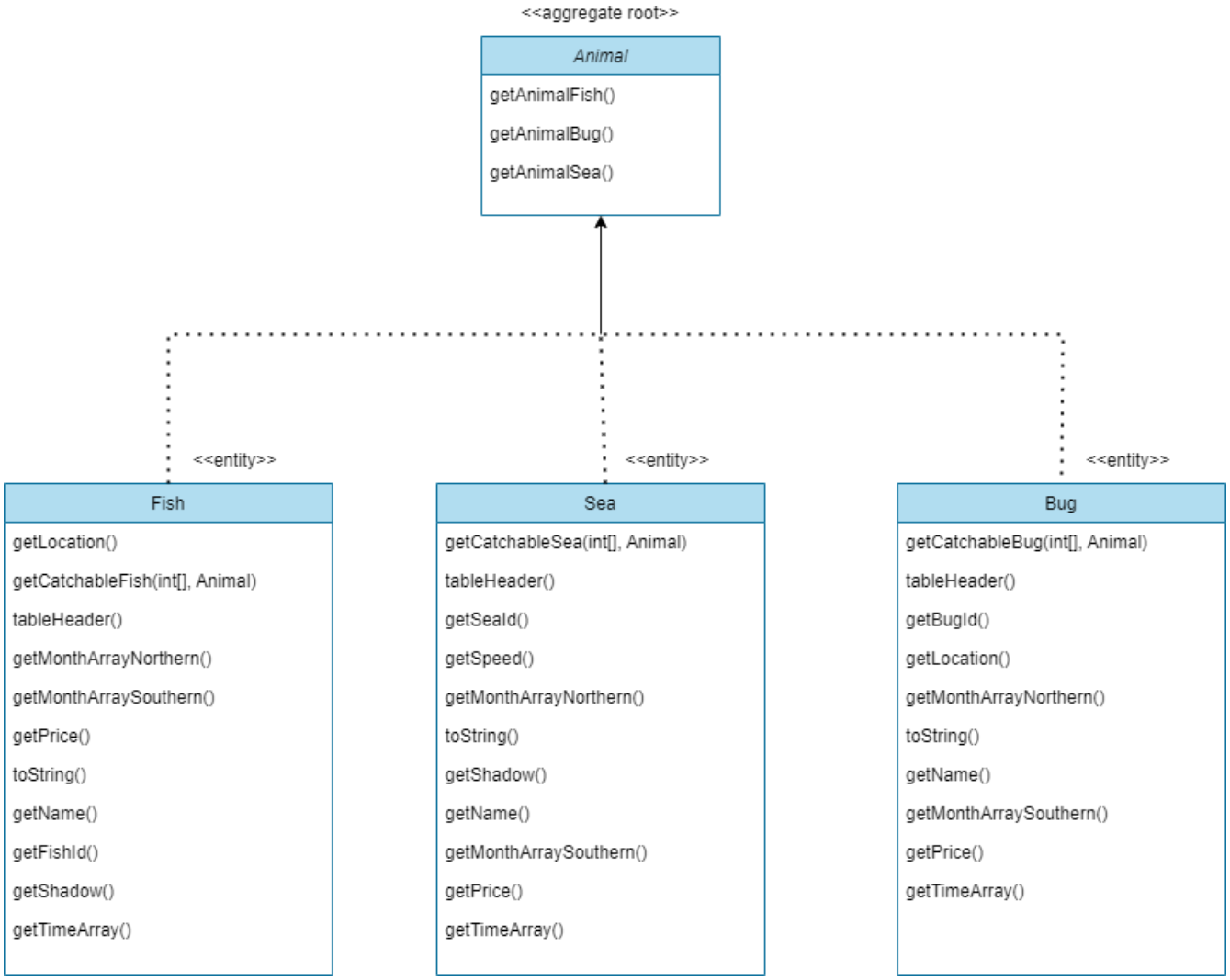
Kapitel 6: Domain Driven Design

Ubiquitous Language

Bezeichnung	Bedeutung	Begründung
Fish	Fisch	Im Spiel Animal Crossing werden die Tierarten unterschieden in Fisch, Meerestier und Insekt. Für die Anwendung werden daher die englischen Äquivalente, die auch in der englischen Version des Spiels genutzt werden, verwendet.
monthArrayNorthern	Array, in welchem Monat das Tier auf der Nordhalbkugel gefangen werden kann	Im Spiel wird zwischen Nord- und Südhalbkugel unterschieden, dementsprechend gibt es im Programm auch zwei verschiedene monthArrays, in denen die jeweiligen Monate hinterlegt sind, in dem das Tier gefangen werden kann.
shadow	Jeder Fisch und jedes Meerestier ist im Wasser durch einen Schatten sichtbar, der je nach Tier eine unterschiedliche Größe hat	Die Größe der Schatten wird passend zu den Begriffen der Domäne gewählt. Beispielsweise wird als Schattengröße groß oder klein verwendet, anstelle von beispielsweise einer Angabe mithilfe von numerischen Werten.
bitterling	Die Instanzen der hart codierten Tiere im AnimalRepository werden nach den originalen Namen im Spiel benannt	Beispielsweise heißt die Instanz des Fisches Bitterling auch bitterling, um möglichst nahe an der Domäne zu bleiben und um die Verwirrung durch andere Namen oder Kurzformen zu verhindern.

Entities

UML:

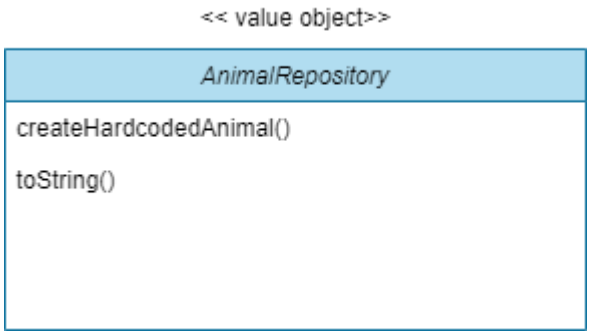


Beschreibung: Die Fische, Meerestiere und Insekten, also alle Tiere, sind Entities.

Begründung: Im Programmentwurf kommen Entities bei den Klassen Bug, Fish und Sea zum Einsatz. Sie haben eindeutige IDs innerhalb der Domäne. Beispielsweise trägt der Fisch Bitterling die ID 1. Der Fisch hat innerhalb der ganzen Anwendung immer die ID, da es sich aus Domänensicht immer um denselben Fisch handelt, somit ist die ID global gültig.

Value Objects

UML:



Beschreibung: Das AnimalRepository stellt die hart codierten Tiere bereit.

Begründung: Im Programmentwurf kommen Value Objects bei der Klasse AnimalRepository zum Einsatz, da sich die Tiere und ihre Werte sowie Eigenschaften nicht mehr ändern sollen. Zusätzlich sind keine Methoden zum Verändern der Objekte, wie beispielsweise setter-Methoden, notwendig.

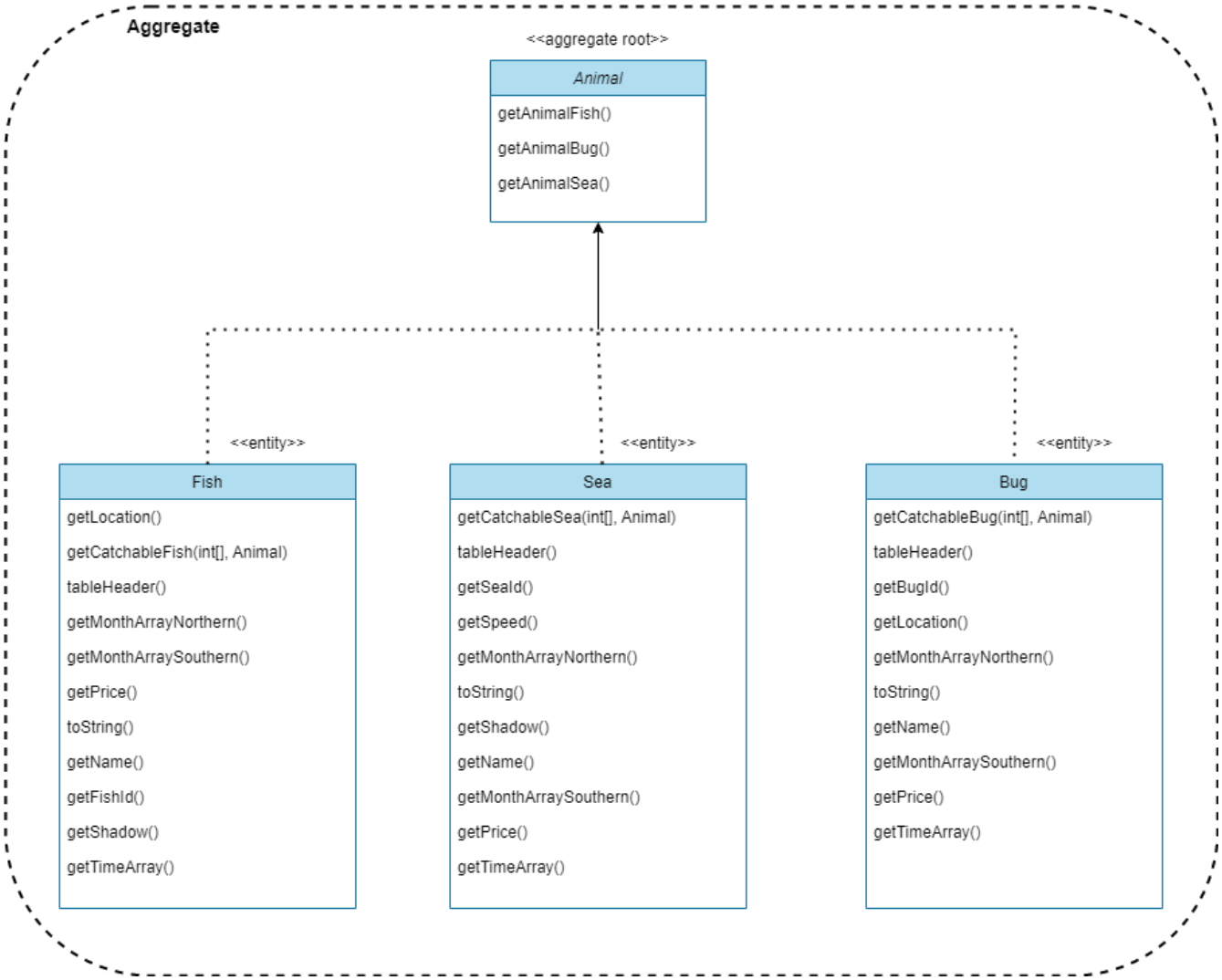
Repositories

In dem Programmentwurf kommt kein Repository zum Einsatz, da die Zugriffsmechanismen über die verschiedenen Klassen aufgeteilt sind. Es existiert keine zentrale Klasse/kein zentrales Repository, welches die Zugriffsmechanismen auf einen potenziellen persistenten Speicher steuert.

Sobald das AnimalRepository nicht mehr für das Zurückgeben von hart codierten Beispiel-Tieren zuständig ist, sondern Zugriffsmechanismen auf einen persistenten Speicher bereitstellt, wird die Klasse zu einem Repository. Dies war für den Programmentwurf allerdings nicht vorgesehen, weshalb es aktuell noch kein Repository gibt.

Aggregates

UML:



Beschreibung: Animal ist ein Aggregate der verschiedenen Tierarten, also Fish, Bug und Sea.

Begründung: Das Animal ist in der Anwendung als eine Einheit zu betrachten, die auch immer als Einheit geladen und verwaltet wird. Dies macht besonder daher Sinn, da bei der Abfrage, was zu einer bestimmten Zeit gefangen werden kann, alle Tierarten betrachtet werden. Auch das Inventar wird übergreifend über alle Tierarten gehalten und verwaltet.

Kapitel 7: Refactoring

Code Smells

Code Smell 1: Long Method - UserService.addCheckedAnimalToUserFile

```
public void addCheckedAnimalToUserFile(String userName, String
animalName) {
Path path = Paths.get("userDirectory\\" + userName + ".txt");
boolean containsAnimalName = false;
try (PrintWriter pw = new PrintWriter(new FileWriter("userDirectory\\" +
userName + ".txt", true)); BufferedReader br = new BufferedReader(new
FileReader("userDirectory\\" + userName + ".txt"))) {
String line;
           while ((line = br.readLine()) != null) {
           if (line.toLowerCase().contains(animalName.toLowerCase())){
               containsAnimalName = true;
           }
}
     if (!containsAnimalName) {
     animalName = animalName.toLowerCase();
          animalName = animalName.substring(0, 1).toUpperCase() +
animalName.substring(1);
           pw.println(animalName);
     } else {
           System.out.println("Das Tier existiert bereits in deinem
Inventar!");
     }
     } catch (IOException e) {
           System.err.println("Fehler beim Schreiben: " +
e.getMessage());
     }
}
```

Lösungsweg: Die Überprüfung, ob das Tier bereits im Inventar existiert, sollte in eine eigene Methode ausgelagert werden. Dadurch wird die Methode übersichtlicher und einfacher lesbar. Zudem ist das Ändern von einzelnen Methoden weniger aufwändig.

Lösung in Pseudo-Code:

```
public void addCheckedAnimalToUserFile (String userName, String
animalName) {
      try (PrintWriter printWriter) {
            if (checkIfAnimalExistsInInventory(userName, animalName)) {
                  print(userName_file, animalName)
            }
      } catch (IOException e) {
            print e.message
}
}
```

```
public boolean checkIfAnimalExistsInInventory (String userName, String
animalName) {
if (userName_file contains animalName) {
            return true
      } else {
            return false
      }
}
```

Code Smell 2: Large Class - UserInteraction

```
public class UserInteraction {

   public void showFirstMessage(){
       System.out.println("Willkommen zum Animal Crossing: New Horizons
Hilfsprogramm");
   }

   public int[] getUserInputForCatchableAnimals(){
       System.out.println("Um dir zu sagen, was du gerade fangen kannst,
brauche ich zuerst ein paar Informationen:");
       Scanner userInputScanner = new Scanner(System.in);
       System.out.println("Liegt deine Insel auf der Nordhalbkugel(1)
oder Südhalbkugel(2)?");
       int hemisphere = userInputScanner.nextInt();
       System.out.println("Welcher Monat ist gerade auf deiner Insel?
```

```
(Beispiel: 1 = Januar, ..., 12 = Dezember)");
        int month = userInputScanner.nextInt();
        System.out.println("Welche Uhrzeit ist auf deiner Insel? (Bitte
nur die volle Stunde eingeben. Beispiel: 16 für 16:34 Uhr und 1 für 01:30
Uhr");
        int time = userInputScanner.nextInt();
        return new int[]{hemisphere, month, time};
    }

    public String getUserInputForUserCreation(){
        System.out.println("Mit welchem Benutzer möchtest du das Programm
starten?");
        Scanner userNameInput = new Scanner(System.in);
        String userName = userNameInput.next();
        return userName;
    }

    public String getUserInputForInventory(){
        System.out.println("Welches Tier möchtest du deinem Inventar
hinzufügen?");
        Scanner animalNameInput = new Scanner(System.in);
        String animalName = animalNameInput.next();
        return animalName;
    }

    public void getNextAction(InformationService informationService,
UserService userService, String userName, Animal hardcodedAnimal) throws
FileNotFoundException {
        while(true) {
            System.out.println("Möchtest du ein Tier zum Inventar
hinzufügen(1), dein Inventar anzeigen(2) oder möchtest du wissen, was du
gerade fangen kannst(3)? Zum Beenden bitte die 4 eingeben.");
            Scanner nextActionInput = new Scanner(System.in);
            int nextAction = nextActionInput.nextInt();
            if (nextAction == 1) {
                String animalName = getUserInputForInventory();
                userService.addAnimalToUserFile(userName, animalName,
userService, hardcodedAnimal);
            } else if (nextAction == 2) {
                userService.OutputAllAnimalsOfInventory(userName);
```

```
            } else if (nextAction == 3) {
                int[] userInput = getUserInputForCatchableAnimals();
                informationService.getCatchableAnimals(userInput,
hardcodedAnimal);
            } else if (nextAction == 4) {
                System.exit(0);
            } else {
                System.out.println("Bitte gebe 1, 2, 3 oder 4 ein.");
                getNextAction(informationService, userService, userName,
hardcodedAnimal);
            }
        }
    }
}
```

Lösungsweg: In der Klasse UserInteraction sind insgesamt 5 Methoden, die der Kommunikation mit den Nutzern dienen. Daher sollte die Klasse sinnvoll in kleiner Klassen, die jeweils zusammengehörende Teilfunktionen beinhalten, aufgeteilt werden. Hier bietet es sich an, die Klasse in vier verschiedene Klassen aufzuteilen:

- UserInteraction
    - showFirstMessage()
    - getNextAction()
- UserInteractionCatchableAnimals
    - getUserInputForCatchableAnimals()
- UserInteractionUserCreation
    - getUserInputForUserCreation()
- UserInteractionInventory
    - getUserInputForInventory()

Lösung in Pseudo-Code:

```
public class UserInteraction {
     public void showFirstMessage(){...}
public void getNextAction(){...}
}
```

```
public class UserInteractionCatchableAnimals {
     public int[] getUserInputForCatchableAnimals(){...}
}
```

```
public class UserInteractionUserCreation {
      public String getUserInputForUserCreation(){...}
}
```

```
public class UserInteractionInventory {
public String getUserInputForInventory(){...}
}
```

2 Refactorings

Refactoring 1: Extract Method

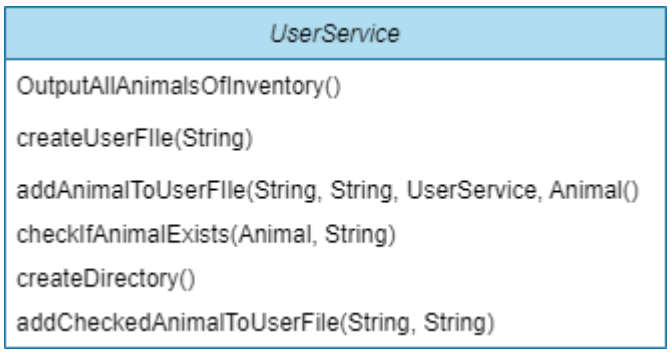
Commit:

- 293f5584489117553a1792ea6d6c99ac13e51edf
- refactoring 1: extract method (at UserService.addCheckedAnimalToUserFile)

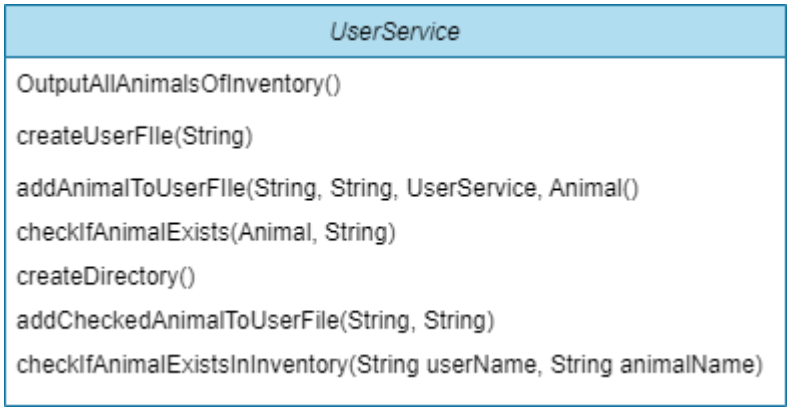
Begründung: Die Methode UserService.addCheckedAnimalToUserFile hatte bisher zwei Funktionen - zum einen wurde überprüft, ob das Tier bereits im Inventar existiert und zum andern wird das Tier (sofern es noch nicht im Inventar existiert) in das Inventar geschrieben.

Die Überprüfung, ob das Tier bereits im Inventar existiert wurde in eine eigene Methode namens checkIfAnimalExistsInInventory ausgelagert. Dadurch übernimmt die ursprüngliche Methode nur noch eine Aufgabe und wird dadurch übersichtlicher und einfacher lesbar. Zudem ist das Ändern der einzelnen Methoden weniger aufwändig.

UML vorher:



UML nachher:



Refactoring 2: Replace Temp with Query

Commit:

- 36973f55695643c50d42e0e2ae03d88843777c1c
- refactoring 2: replace temp with query (at UserInteraction.getUserInputForCatchableAnimals)

Begründung: In der Methode getUserInputForCatchableAnimals wurde zuvor eine temporäre Variable zur Zwischenspeicherung der Nutzereingabe verwendet. Da diese Variable danach nicht mehr verändert wird, ist die Nutzung der Variable in dem Fall unnötig. Daher wurde die Variable entfernt und direkt zurückgegeben, ohne dass eine Zwischenspeicherung der Rückgabe erfolgt.

Da auf einem UML keine Änderung ersichtlich wäre, wird nachfolgend der Code vor und nach dem Refactoring aufgezeigt.

Code vorher:

```
public int[] getUserInputForCatchableAnimals(){
   System.out.println("Um dir zu sagen, was du gerade fangen kannst,
brauche ich zuerst ein paar Informationen:");
   Scanner userInputScanner = new Scanner(System.in);
   System.out.println("Liegt deine Insel auf der Nordhalbkugel(1) oder
Südhalbkugel(2)?");
   int hemisphere = userInputScanner.nextInt();
   System.out.println("Welcher Monat ist gerade auf deiner Insel?
(Beispiel: 1 = Januar, ..., 12 = Dezember)");
   int month = userInputScanner.nextInt();
   System.out.println("Welche Uhrzeit ist auf deiner Insel? (Bitte nur
die volle Stunde eingeben. Beispiel: 16 für 16:34 Uhr und 1 für 01:30
```

```
Uhr");
      int time = userInputScanner.nextInt();
int[] userInput = {hemisphere, month, time};
return userInput;
}
```

Code nachher:

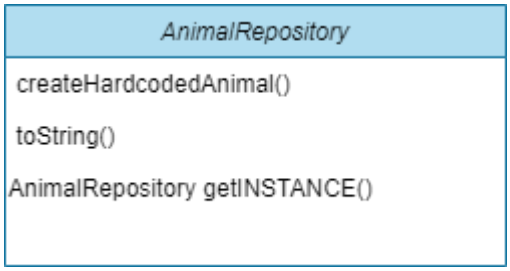
```
public int[] getUserInputForCatchableAnimals(){
   System.out.println("Um dir zu sagen, was du gerade fangen kannst,
brauche ich zuerst ein paar Informationen:");
   Scanner userInputScanner = new Scanner(System.in);
   System.out.println("Liegt deine Insel auf der Nordhalbkugel(1) oder
Südhalbkugel(2)?");
   int hemisphere = userInputScanner.nextInt();
   System.out.println("Welcher Monat ist gerade auf deiner Insel?
(Beispiel: 1 = Januar, ..., 12 = Dezember)");
   int month = userInputScanner.nextInt();
   System.out.println("Welche Uhrzeit ist auf deiner Insel? (Bitte nur
die volle Stunde eingeben. Beispiel: 16 für 16:34 Uhr und 1 für 01:30
Uhr");
   int time = userInputScanner.nextInt();
   return new int[]{hemisphere, month, time};
}
```

Kapitel 8: Entwurfsmuster

Entwurfsmuster: Singleton

Begründung: Das AnimalRepository soll innerhalb der Anwendung immer nur genau ein Exemplar besitzen und einen globalen Zugriffspunkt auf den Tier-Datensatz darstellen. Durch die einmalige “Erzeugung nach Bedarf” können zudem Ressourcen eingespart werden.

UML:



Entwurfsmuster: Kompositum

Begründung: Die Klassen Bug, Fish und Sea haben viele Methoden, wie zum Beispiel getId oder getPrice, die jeweils in jeder Klasse benötigt werden. Daher macht es Sinn, in Form eines Kompositums eine Interface-Klasse einzuführen, die genau diese Methoden bereitstellt. Diese können dann in den Klassen Bug, Fish und Sea konkret implementiert bzw. überschrieben werden. Das bietet den Vorteil, dass man die Anwendung leicht um weitere Tierarten erweitern könnte. Dabei kann man die allgemeingültigen Methoden des Interfaces wiederverwenden. Außerdem kann so eine einheitliche Bedienung und Allgemeingültigkeit sichergestellt werden.

UML: