# Proiect Laborator MIP

**Dan Ștefan-Călin**

**Informatică Aplicată, grupa 10LF331**

# Introducere

În acest proiect am implementat în limbajul de programare Java o aplicatie de tip consolă care gestionează  informațiile despre animale dintr-un adăpost, permițând adăugarea, ștergerea și căutarea acestora în funcție de diferite criterii, precum vărsta sau de cât timp sunt la adăpost.

Pentru a gestiona lista de animale folosesc un obiect “Shelter” care face marea majoritate a operațiilor pe lista. Folosesc și clase utilitare pentru implementarea comparatorilor, citiri de la tastatura și afișări de animale în consolă. Clasa “Animal” are proprietăți polimorfice, aceasta fiind clasa de bază pentru câteva tipuri diferite de animale (pisici, câini, păsări, etc.) care sunt toate reținute în aceeași listă, lista de animale din clasa “Shelter”.

Pentru a realiza acest proiect am folosit multiple elemente de bază ale limbajului de programare Java și bibliotecile externe Jackson (pentru serializare și deserializare pentru obiecte) si JUnit5 (pentru unit testing).

# Componente folosite

1. Elemente de bază java (Introducere în Java, laboratorul 1)

Pentru implementare am folosit tipuri de date din Java pentru a reține membrii claselor. Pentru a modela datele calendaristice, am utilizat tipul LocalDate din pachetul java.time, care oferă funcții avansate pentru manipularea datelor.
În figura de mai jos se află membrii clasei abstracte Animal:

```
private int id;  3 usages
private String name;  3 usages
private LocalDate birthDate;  4 usages
private LocalDate admissionDate;  3 usages
private String healthStatus;  3 usages
private boolean isAdopted;  3 usages
```

Pentru scrierea în consolă a informațiilor unui animal am folosit o clasă numită “AnimalPrinter” care are o functie statică care preia un parametru de tip generic care trebuie sa aibe “forma” unui obiect de tip Animal. Mai jos se poate observa modul în care se realizează afișarea în consolă a informațiilor unui animal, inclusiv câmpurile specifice diferitelor tipuri de animale

```
public static <T extends Animal> void printAnimalDetails(T animal) {  3 usages  Dan Stefan Calin
    System.out.println("ID: " + animal.getId());
    System.out.println("Name: " + animal.getName());
    System.out.println("Species: " + animal.getClass().getSimpleName());
    System.out.println("Birth Date: " + animal.getBirthDate());
    System.out.println("Admission Date: " + animal.getAdmissionDate());
    System.out.println("Health Status: " + animal.getHealthStatus());
    System.out.println("Is Adopted: " + animal.getisAdopted());

    if (animal instanceof Dog) {
        Dog dog = (Dog) animal;
        System.out.println("Breed: " + dog.getBreed());
        System.out.println("Training Level: " + dog.getTrainingLevel());
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal;
        System.out.println("Breed: " + cat.getBreed());
        System.out.println("Is Sociable: " + cat.getisSociable());
    } else if (animal instanceof Bird) {
        Bird bird = (Bird) animal;
        System.out.println("Wing Span: " + bird.getWingSpan());
        System.out.println("Can Fly: " + bird.getCanFly());
    } else if (animal instanceof Other) {
        Other other = (Other) animal;
        System.out.println("Description: " + other.getDescription());
    }

    System.out.println("\n");
}
```

În plus, ca un exemplu de funcție, pentru animale, vârsta nu este reținuta, dar este calculată în funcție de data nașterii.

```
@JsonIgnore  4 usages  Dan Stefan Calin
public int getAge() { return LocalDate.now().getYear() - birthDate.getYear(); }
```

## 2. Elemente de bază java (Introducere în Java, laboratorul 2)

Pentru citiri de la tastatură am folosit un obiect de tip scanner. Un exemplu de utilizare a unui astfel de obiect se poate regăsi în clasa “AnimalReader” din funcția statică din interiorul acesteia. Această funcție preia date de la tastatură și returnează un obiect de tip animal. În snippet-ul de cod de mai jos se citesc datele commune tutuor tipurilor de animale.

```
System.out.println("Enter species(Dog, Cat, Bird, Other):");
String species = scanner.nextLine();

System.out.println("Enter name:");
String name = scanner.nextLine();

System.out.println("Enter birth date (yyyy-mm-dd):");
LocalDate birthDate = LocalDate.parse(scanner.nextLine());

System.out.println("Enter admission date (yyyy-mm-dd):");
LocalDate admissionDate = LocalDate.parse(scanner.nextLine());

System.out.println("Enter health status:");
String healthStatus = scanner.nextLine();

System.out.println("Is the animal adopted? (true/false):");
boolean isAdopted = scanner.nextBoolean();
scanner.nextLine();
```

Am folosit bucle for pentru a cicla prin lista de animale din shelter. În următoarea figură se realizează updatarea id-ului cu care va fi salvat primul animal din acea rulare a programului, după încarcarea datelor din json.

```
for(Animal animal:shelter.getAnimals())
{
    if(animal.getId() > id)
        id = animal.getId();
}
id++;
```

În cadrul clasei ShelterService am folosit o buclă do...while pentru rularea aplicației până când utilizatorul dorește sa o oprească și o structură switch pentru alegerea operației pe care utilizatorul dorește sa o aplice. În figură se pot observa afișările în consolă ale opțiunilor, începutul buclei do...while și primul case din switch.

```
System.out.println("Welcome!");
do {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Please choose an option: ");
    System.out.println("1. Add animal (keyboard input)");
    System.out.println("2. Show animal by id");
    System.out.println("3. Show oldest animal");
    System.out.println("4. Show animal that has been in the shelter for the longest amount of time");
    System.out.println("5. Print all animals");
    System.out.println("6. Remove animal by id");
    System.out.println("7. Exit \n");
    option = scanner.nextInt();
    scanner.nextLine();
    switch (option) {
        case 1:
            try{
                readAnimalFromInput(id,scanner,shelter); id++;
            }
            catch(Exception e)
            {
                System.out.println(e.getMessage());
            }

            break;
```

## 3. Colecții Java (laboratorul 3)

Din capitolul “colecții” am utilizat o listă, mai exact un ArrayList, în care mi-am reținut animalele din adăpost.

```
private List<Animal> animals;  11 usages

public Shelter() {  3 usages  Dan Stefan Calin
    this.animals = new ArrayList<>();
}
```

Pe această listă am aplicat diferite sortări după comparatori custom, și am preluat primul sau ultimul element, în funcție de ce operație s-a apelat.

Sortări:

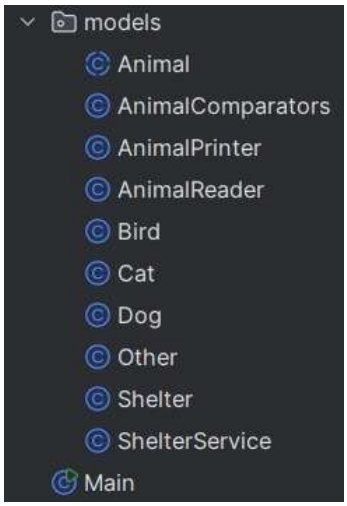
```
public void sortAnimalsbyName() {  1 usage  Dan Stefan Calin
    this.animals.sort(AnimalComparators.NAME_COMPARATOR);
}
public void sortAnimalsbyAge() {  2 usages  Dan Stefan Calin
    this.animals.sort(AnimalComparators.AGE_COMPARATOR);
}
public void sortAnimalsbyAdmissionDate() {  2 usages  Dan Stefan Calin
    this.animals.sort(AnimalComparators.ADMISSION_DATE_COMPARATOR);
}
public void sortAnimalsbyID() {  2 usages  Dan Stefan Calin
    this.animals.sort(AnimalComparators.ID_COMPARATOR);
}
```

Exemplu de utilizare a uneia dintre aceste sortări, în contextul afișării animalului care se află la adăpost de cel mai mult timp:

```
case 4: shelter.sortAnimalsbyAdmissionDate();
    var animalLongestTime = shelter.getAnimals().getFirst();
    LocalDate currentDate = LocalDate.now();
    Period timeSpent = Period.between(animalLongestTime.getAdmissionDate(), currentDate);
    System.out.println(animalLongestTime.getName() + " Time spent here: " +
            timeSpent.getYears() + " years, " +
            timeSpent.getMonths() + " months, and " +
            timeSpent.getDays() + " days.");
    break;
```

## 4. Clase Java (laboratorul 4)

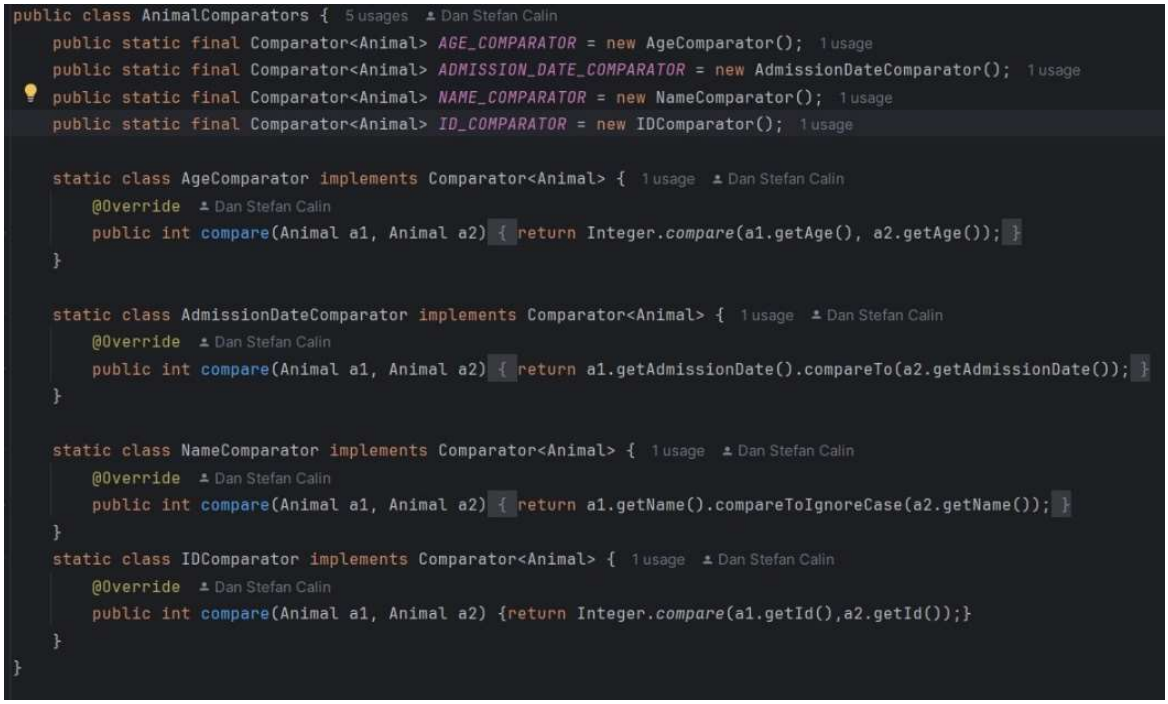
În cadrul proiectului am utilizat o multitudine de clase, fiecare cu un rol specific. Lista de clase din cadrul proiectului:




Spre exemplu, clasa “Cat” reține informațiile specifice unei pisici, iar în cadrul subobiectului său de tip Animal se rețin datele comune cu restul tipurilor de animale. Ca metode în această clasă se află setterii și getteri pentru aceste informații specifice tipului “Cat”.

```
public class Cat extends Animal implements iCat {  18 usages  Dan Stefan Calin

    private String breed;  3 usages
    private boolean isSociable;  3 usages

    public Cat() {}  8 usages  Dan Stefan Calin
    public Cat(int id, String name, LocalDate birthDate, LocalDate admissionDate, String healthStatus,
               boolean isAdopted, String breed, boolean isSociable) {
        super(id, name, birthDate, admissionDate, healthStatus, isAdopted);
        this.isSociable = isSociable;
        this.breed = breed;
    }

    public boolean getisSociable() {  3 usages  Dan Stefan Calin
        return isSociable;
    }

    public void setisSociable(boolean sociable) { isSociable = sociable; }

    public String getBreed() { return breed; }

    public void setBreed(String breed) { this.breed = breed; }
}
```

Ca un alt exemplu, clasa AnimalComparators se ocupă de stocarea a câțiva comparatori ca membri statici. Acești membri statici se pot accesa, de exemplu într-un apel de sort(), fără a instanția clasa.



```
public class AnimalComparators {  5 usages  Dan Stefan Calin
    public static final Comparator<Animal> AGE_COMPARATOR = new AgeComparator();  1 usage
    public static final Comparator<Animal> ADMISSION_DATE_COMPARATOR = new AdmissionDateComparator();  1 usage
    public static final Comparator<Animal> NAME_COMPARATOR = new NameComparator();  1 usage
    public static final Comparator<Animal> ID_COMPARATOR = new IDComparator();  1 usage

    static class AgeComparator implements Comparator<Animal> {  1 usage  Dan Stefan Calin
        @Override  Dan Stefan Calin
        public int compare(Animal a1, Animal a2) { return Integer.compare(a1.getAge(), a2.getAge()); }
    }

    static class AdmissionDateComparator implements Comparator<Animal> {  1 usage  Dan Stefan Calin
        @Override  Dan Stefan Calin
        public int compare(Animal a1, Animal a2) { return a1.getAdmissionDate().compareTo(a2.getAdmissionDate()); }
    }

    static class NameComparator implements Comparator<Animal> {  1 usage  Dan Stefan Calin
        @Override  Dan Stefan Calin
        public int compare(Animal a1, Animal a2) { return a1.getName().compareToIgnoreCase(a2.getName()); }
    }
    static class IDComparator implements Comparator<Animal> {  1 usage  Dan Stefan Calin
        @Override  Dan Stefan Calin
        public int compare(Animal a1, Animal a2) {return Integer.compare(a1.getId(),a2.getId());}
    }
}
```

Fiecare clasă pe care am folosit-o în cadrul proiectului are o menire clară: clasele Animal, Cat, Dog, Bird și Other rețin informații despre diferite tipuri de animale (Animal informații comune, restul informații specifice pe tip); Shelter se ocupă de gestionarea listei de animale; ShelterService este clasa cu care interacționează utilizatorul, în cadrul acesteia aflându-se un obiect Shelter și apeluri către implementările operațiilor aplicate listei de animale; Clasele AnimalReader, AnimalPrinter și AnimalComparators sunt clase ajutătoare, care gestionează logici externe obiectului de tip Shelter (ex. citirea detaliilor unui animal de la tastatură).

## 5. Moștenire în Java (laboratorul 5)

Ca exemplu de moștenire din cadrul proiectului este ierarhia de clase ”Animals”. Clasa Animal este clasa abstractă de bază, în care se rețin informații comune tuturor tipurilor de animale. Fiind abstractă această clasă nu poate fi instanțiată. Clasele care o moștenesc adaugă informații specifice diferitelor tipuri de animale, spre exemplu nivelul de antrenare al unui câine, sau daca o pisică este sociabilă sau nu.

```
public abstract class Animal implements iAnimal
```

```
public class Dog extends Animal implements iDog {  19 usages  Dan Stefan Calin

    private String breed;  3 usages
    private String trainingLevel;  3 usages

    public Dog() {}  9 usages  Dan Stefan Calin
    public Dog(int id, String name, LocalDate birthDate, LocalDate admissionDate, String healthStatus,  2 usages  Dan Stefan Calin
               boolean isAdopted, String breed, String trainingLevel) {
        super(id, name, birthDate, admissionDate, healthStatus, isAdopted);
        this.breed = breed;
        this.trainingLevel = trainingLevel;
    }

    public String getBreed() {  3 usages  Dan Stefan Calin
        return breed;
    }

    public void setBreed(String breed) {  2 usages  Dan Stefan Calin
        this.breed = breed;
    }

    public String getTrainingLevel() {  3 usages  Dan Stefan Calin
        return trainingLevel;
    }

    public void setTrainingLevel(String trainingLevel) {  2 usages  Dan Stefan Calin
        this.trainingLevel = trainingLevel;
    }
}
```

În cadrul clasei shelter se reține o listă de animale, dar lista este de tipul Animal, tipul abstract, dar, cum o clasă abstractă nu poate fi instanțiată, în acea listă de animale, de fapt, se află câini, pisici și alte tipuri de animale. Această proprietate se numește polimorfism, și este indispensabilă implementării logicii adăpostului de animale. Se poate verifica de ce tip este un anumit element al listei folosind apeluri de “instanceof”. În următoarea figură se poate observa un exemplu din clasa AnimalPrinter:

```
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
    System.out.println("Breed: " + dog.getBreed());
    System.out.println("Training Level: " + dog.getTrainingLevel());
```

## 6. Interfețe în Java (laboratorul 6)

Interfețele din Java sunt adesea comparate cu un contract. Pentru ca o clasă să poată extindă o interfață aceasta trebuie sa “respecte contractul”, adică să implementeze funcțiile din interfață. Un exemplu foarte bun din proiectul meu este interfața iShelter, pe care o implementează clasa Shelter.
Interfața iShelter:

```
public interface iShelter {  6 usages  1 implementation  Dan Stefan Calin
    void addAnimal(Animal animal);  4 usages  1 implementation  Dan Stefan Calin
    void removeAnimalById(int id);  2 usages  1 implementation  Dan Stefan Calin
    Animal getAnimalById(int id);  3 usages  1 implementation  Dan Stefan Calin
    List<Animal> getAnimals();  12 usages  1 implementation  Dan Stefan Calin
    void sortAnimalsbyName();  1 usage  1 implementation  Dan Stefan Calin
    void sortAnimalsbyAge();  2 usages  1 implementation  Dan Stefan Calin
    void sortAnimalsbyAdmissionDate();  2 usages  1 implementation  Dan Stefan Calin
    void sortAnimalsbyID();  2 usages  1 implementation  Dan Stefan Calin
    void saveToFile(String filePath);  3 usages  1 implementation  Dan Stefan Calin
    void loadFromFile(String filePath);  2 usages  1 implementation  Dan Stefan Calin
}
```

Pentru ca Shelter să poată “moșteni” interfața iShelter aceasta ar trebui sa aibe implementate toate funcțiile ale căror signatură se află în interfață. Acest lucru se asigură că clasa Shelter oferă toate serviciile pe care ar trebui să le ofere un obiect de acest fel. În plus, această inferfață poate să ajute un alt programator care vrea sa utilizeze o parte din sau poate chiar tot proiectul. Prin centralizarea antetelor funcțiilor într-un singur loc este mult mai ușor de înțeles modul în care clasa poate fi folosită.

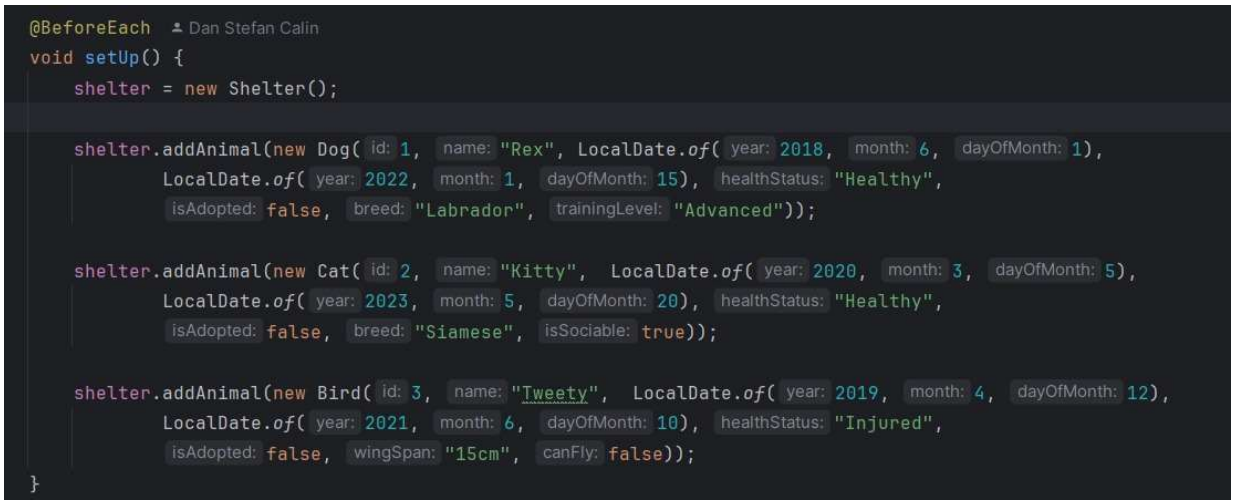
```
public class Shelter implements iShelter {
```

Totodată, în cadrul altor clase putem folosi proprietatea de polimorfism a moștenirii și să instanțiem un obiect de tipul interfeței, inițializat cu un obiect de tipul clasei care implementează interfața. Ca exemplu al acestei proprietăți se poate considera obiectul iShelter din ShelterService sau poate chiar și instanțierea listei de animale din clasa Shelter, aceasta fiind de tip List, dar concret în memorie se află un ArrayList. List este interfața pe care o implementează mai multe clase standard din Java, una dintre acestea fiind ArrayList.

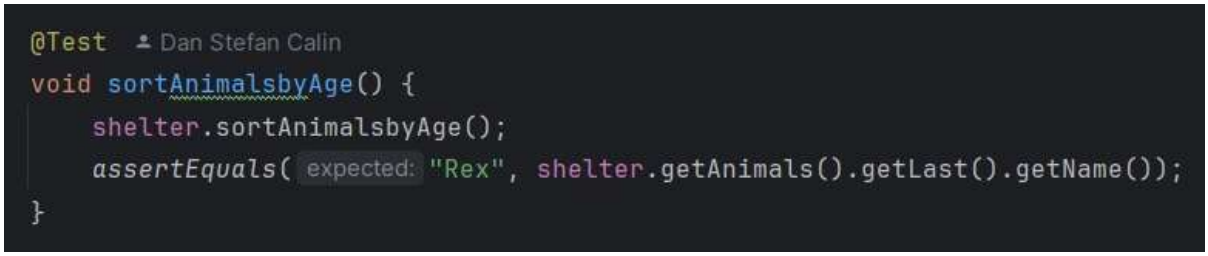
## 7. Teste pentru metode (laboratorul 7)

Pentru a asigura calitatea codului scris, este bine sa facem și unit testing pentru funcții, cu scopul de a detecta erorile înainte ca acestea sa ajungă în varianta finală a codului ca bug-uri. Pentru a face acest lucru am folosit biblioteca externa JUnit. În continuare voi da un exemplu de unit test făcut pe una din metodele din clasa Shelter.

Inainte de orice, trebuie ca obiectul shelter să fie inițializat pentru teste, astfel se pot testa operațiile de adăugare, ștergere și căutare de animale, și operațiile de citire/scriere în fișierul json.

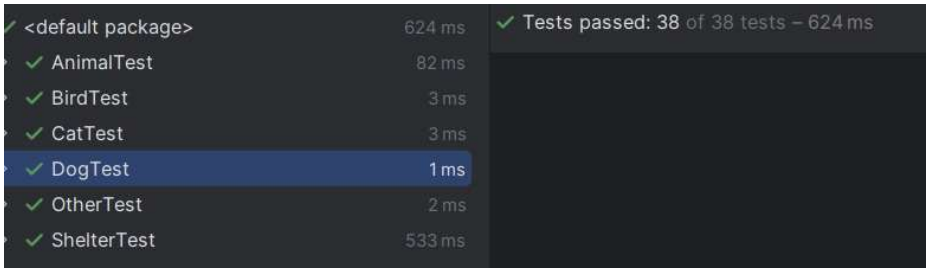



Metoda sortAnimalsbyAge() ar trebui să sorteze lista de animale în ordine crescătoare în funcție de vârstă. Din inițializare reiese ca Rex ar trebui sa fie ultimul din listă, acesta fiind născut in 2018.



În urma rulării testului se pare că aserția noastră a fost corectă și că metoda se comportă în modul dorit și așteptat


Acesta este rezultatul rulării testelor pe care le-am implementat.




Se pare că metodele testate nu au bug-uri și proiectul este pregătit pentru release.

## 8. Persistența datelor (laboratorul 8)

Doresc să menționez ca în cadrul acestui capitol am întâmpinat cele mai multe probleme din punct de vedere al compatibilității metodelor de serializare și deserializare. Voi începe cu detalierea metodei prezente în varianta finală a proiectului și la final voi prezenta pe scurt si provocările cu care m-am înfruntat.

În primul rând, acesta este un exemplu din json-ul în care se vor salva animalele din adăpost.

```
{
  "id": 1,
  "name": "Rex",
  "birthDate": [2018, 6, 1],
  "admissionDate": [2022, 1, 15],
  "healthStatus": "Healthy",
  "isAdopted": false,
  "breed": "Labrador",
  "trainingLevel": "Advanced",
  "species": "Dog"
},
{
  "id": 2,
  "name": "Kitty",
  "birthDate": [2020, 3, 5],
  "admissionDate": [2023, 5, 20],
  "healthStatus": "Healthy",
  "isAdopted": false,
  "breed": "Siamese",
  "isSociable": true,
  "species": "Cat"
},
```

După cum se observă, câmpurile din json se mulează pe modelul câmpurilor din clase, dar a apărut un câmp nou “Species”. Cu ajutorul acelui câmp se poate determina ce fel de animal se citește/scrie în fișier, tipurile de animale având unele câmpuri diferite, este nevoie sa știm ce strategie trebuie să folosim pentru a serializa/deserializa acel obiect.

În continuare voi detalia toți pașii pe care i-am îndeplinit pentru a folosi datele din acest json în proiectul meu.

Pentru a putea integra acest json în proiect am folosit biblioteca externă Jackson, mai jos este o imagine în care se regăsesc dependințele pe care le-am inclus în proiectul Maven în scopul de a folosi biblioteca.

```xml
<!-- main Jackson library -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.15.2</version>
</dependency>

<!-- java.time support -->
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
    <version>2.15.2</version>
</dependency>

<!-- JSON format -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.15.2</version>
</dependency>
```

Implicit, biblioteca nu are suport pentru tipul de date LocalDate din java.time.

Din cauza faptului că obiectele serializate sunt polimorfice trebuie sa îi menționăm bibliotecii Jackson ca există subtipuri și cum sa facă diferența dintre ele. Acest lucru se rezolvă cu niște adnotări din jackson-annotations

```java
@JsonTypeInfo(  4 inheritors  Dan Stefan Calin
        use = JsonTypeInfo.Id.NAME,
        include = JsonTypeInfo.As.PROPERTY,
        property = "species"
)
@JsonSubTypes({
        @JsonSubTypes.Type(value = Dog.class, name = "Dog"),
        @JsonSubTypes.Type(value = Cat.class, name = "Cat"),
        @JsonSubTypes.Type(value = Bird.class, name = "Bird"),
        @JsonSubTypes.Type(value = Other.class, name = "Other")
})
```

Cu aceste adnotări îi identificăm ce subtipuri exista, cum să gestioneze clasele care moștenesc Animal și care este proprietatea care face diferența dintre ele.

```java
@JsonIgnore  4 usages  Dan Stefan Calin
public int getAge() { return LocalDate.now().getYear() - birthDate.getYear(); }

@JsonProperty("species")  Dan Stefan Calin
public String getSpecies() { return getClass().getSimpleName(); }
```

Cu aceste adnotări m-am asigurat ca nu imi adauga un câmp “age” în obiectele din json (JsonIgnore) și că parser-ul de json din Jackson poate sa imi serializeze/deserializeze obiectele din tipurile care moștenesc Animal(JsonProperty).

Folosind toate aceste adnotări pentru a specifica modul în care doresc să imi gestioneze ierarhia de clase, serializarea (scrierea) și deserializarea (citirea) se realizează folosind un obiect ObjectMapper din biblioteca Jackson.

```java
public void saveToFile(String filePath) {  3 usages  Dan Stefan Calin
    ObjectMapper mapper = new ObjectMapper();
    mapper.registerModule(new JavaTimeModule());

    try {
        mapper.writeValue(new File(filePath), animals);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public void loadFromFile(String filePath) {  2 usages  Dan Stefan Calin
    ObjectMapper mapper = new ObjectMapper();
    mapper.registerModule(new JavaTimeModule());

    try {
        animals = mapper.readValue(new File(filePath), new TypeReference<List<Animal>>() {});  Dan Stefan Calin
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

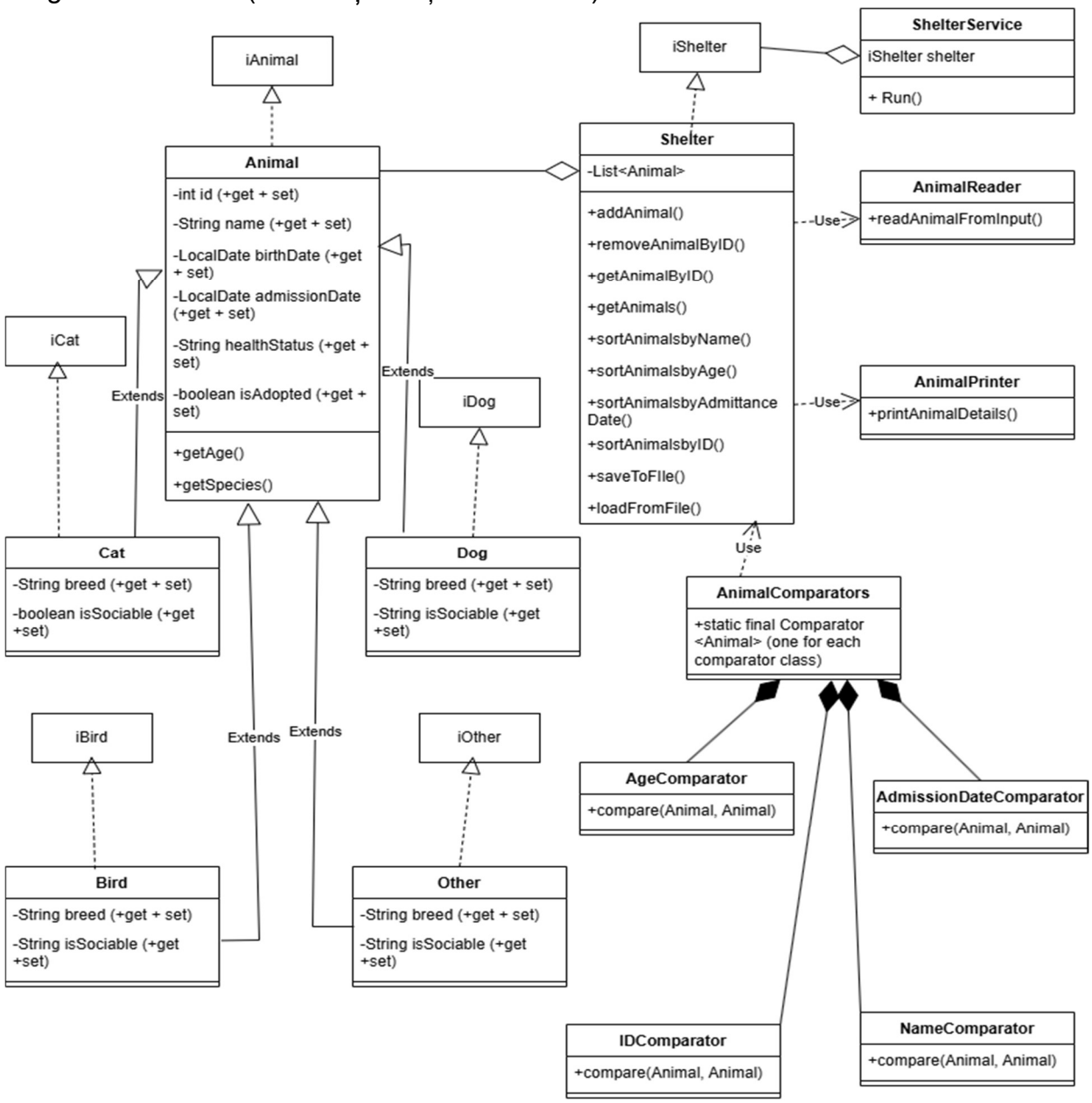
Provocări majore în realizarea acestei părți din proiect am întâmpinat la alegerea strategiei de serializare/deserializare și compatibilitatea tipurilor de date din java.time.
Inițial am vrut să folosesc interfata “Serializable” din Java, pe care Animal ar fi moștenit-o. Avantajele acestei strategii erau compatibilitatea implicită cu tipurile de date din java.time și ar fi eliminat dependințele externe. Totuși, folosind strategia cu interfata Serializable în fișierul json deserializă obiectele ca text in format binar, un format care *nu poate fi citit de om*, fapt ce m-a determinat sa schimb strategia și să folosesc biblioteca Jackson.

Însă, Jackson nu are suport implicit pentru tipurile de date din java.time și mesajele de eroare din excepțiile aruncate în momentul încercării de a citi/scrie o dată calendaristica nu au fost de foarte mare ajutor în identificarea problemei. Într-un final am reușit sa identific problema cu incompatibilitatea cu datele și am adaugat dependința pentru date în Jackson.

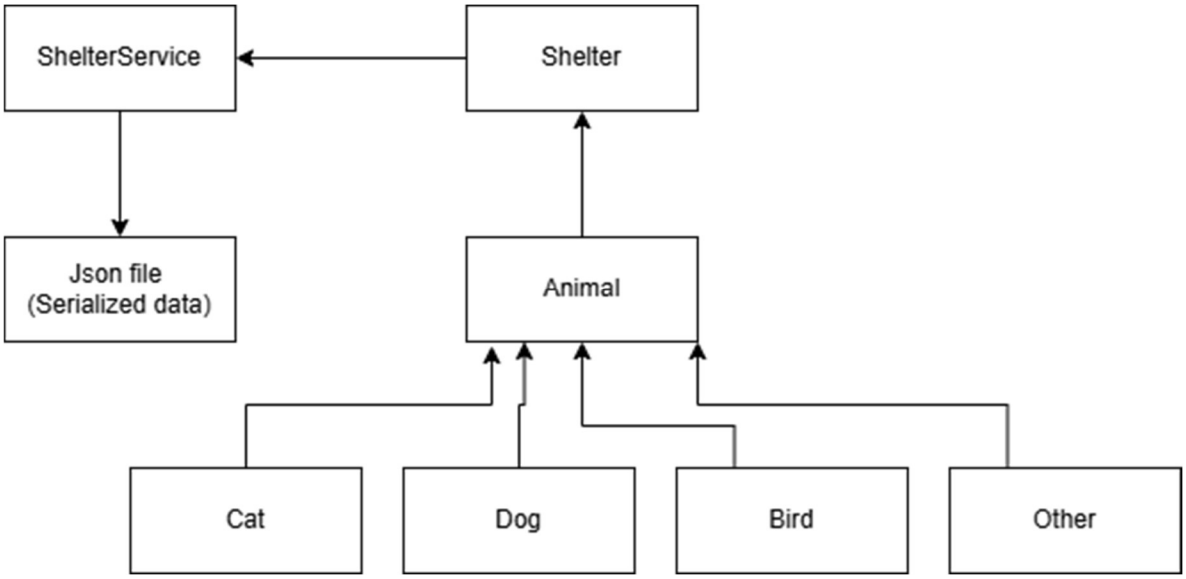
## 9. Diagrame UML (laboratorul 9)

În acest ultim capitol al acestui document voi prezenta 3 diagrame UML care detaliază (ce detaliaza diagramele).

a) Diagrama de clase (clasele și relațiile dintre ele)



b) Diagrama de componente (componentele de bază ale aplicației și cum comunică)




c) Diagrama de workflow al utilizatorului