Animal Shelter Management System

Team 2;

Louis Chartier, Rima Dagher and

David Demers

Project Structure

- We restructured our project from the previous version to better adhere to software design principles that we have learnt over the course of this program and this class. As the singleton principle suggests all tasks should accomplish only one thing. We tried to use this principle as well as possible.
- We tried to minimize each class and divide the functionalities into as many classes as possible.
- Along with this we also tried to divide our project folders into as many categories as possible. With our previous version our model folder was one big folder with 10+ classes inside it. This made it very confusing and hard to navigate. So, for this version of our project, we refactored the project and divided classes into appropriate folders.
- We also added folders with classes which are important for a proper software application such as;
    - Test folder with unit tests for exception handling.
    - Creational folder with entity factories.
    - Observer folder with notification classes for volunteers who need to be notified of tasks.
    - Strategy folder with search classes for easy search methods to filter through entity data retrieval.

How we Managed Entities

- For our project we needed collections to manage bundling's of objects;
  - Animal stores medical entries
  - Shelter stores animals
  - Volunteer stores tasks and dates
  - Adopter stores adopted animals
- For our previous version of our project, we always used Array List's. This time however we used a Queue for the shelter to allow the Adopter to pick an Animal that has been staying at the shelter for the longest, giving them a much-needed home.
- We also could have used a Queue for storing Volunteer tasks to have easily take the tasks that need to be done most urgently from the top of the Queue.

```java
public class ShelterQueue {
    private Queue<Animal> animalQueue;

    /**
     * Default constructor without parameters.
     */
    public ShelterQueue() {
        this.animalQueue = new LinkedList<>();
    }

    /**
     * Adds an animal to the end of the queue.
     * @param animal
     */
    public void enqueueAnimal(Animal animal) {
        if (animal != null && !animal.isAdopted()) {
            animalQueue.add(animal);
        } else {
            throw new IllegalArgumentException("Animal must be valid a
        }
    }

    /**
     * Removes and returns the animal at the front of the queue.
     * @return animal at the front of the queue, or null if the queue
     */
    public Animal dequeueAnimal() {
        return animalQueue.poll();
    }
```

Exception Handling and Testing

- To ensure our code did not have any runtime errors we designed all our classes with exception handling.
- All our constructors use setters which have built-in exception handling to ensure that no errors occur during object creation, update or retrieval.
- To ensure our exception handling was adequate we created tests for all our entities;
    - They checked if exception were thrown when they should be expected.
    - They checked that return statements matched the expected value.
    - They tested edge cases for our setters.
- Using tests allowed us to see what kind of inputs would cause errors and made it clear how we had to fix our classes to handle these exceptions.

```java
@Test  David Demers *
void removeAdoptedAnimal_ExistingAnimal_ShouldRemove() {
    adopter.addAdoptedAnimal(adoptedAnimal);
    assertEquals(adoptedAnimal, adopter.getAdoptedAnimals().get(0));
    adopter.removeAdoptedAnimal(adoptedAnimal);
    assertTrue(adopter.getAdoptedAnimals().isEmpty());
}
```

```java
@ParameterizedTest  David Demers
@ValueSource(strings = {"", " "})
void setName_ShouldThrowForEmptyName(String name) {
    assertThrows(IllegalArgumentException.class, () -> testDog.setName(name));
}
```

```java
@Test  David Demers *
void setEmail_InvalidFormat_ShouldThrow() {
    assertThrows(IllegalArgumentException.class, () -> volunteer.setEmail("email with no at sign"));
}
```

New Feature: Animal Search

- Purpose: Allow user to search for specific criteria (age, name, species, etc.)
- Strategy Pattern: Implements a search strategy to support multiple search algorithms without modifying core logic.
- The interface SearchStrategy defines a method to be implemented by each specific search type we need to create for example; AgeSearch, NameSearch or SpeciesSearch.
- SOLID Principles:
  - S: Each strategy class handles only one type of search logic
  - O: New search types can be added without the change of existing code
  - L: All strategies can be used wherever SearchStrategy is expected
  - I: Strategy clients only rely on search()
  - D: AnimalService depends on SearchStrategy abstraction

Animal Search Continued

SearchStrategy.java

```java
// Interface
public interface SearchStrategy {  7 usages  3 imp
    List<Animal> search(List<Animal> animals);
}
```

AnimalService.java

```java
// Service
public List<Animal> searchAnimals(SearchStrategy strategy) {
    List<Animal> animals = animalRepository.findAll();
    return strategy.search(animals);
}
```

NameSearchStrategy.java

```java
// Example Strategy
public class NameSearchStrategy  implements SearchStrategy {  3 usages  Louis-TM

    private final String name;  2 usages

    public NameSearchStrategy(String name) {  1 usage  Louis-TM
        this.name = name;
    }

    @Override  1 usage  Louis-TM
    public List<Animal> search(List<Animal> animals) {
        return animals.stream()
                .filter( Animal animal -> animal.getName().toLowerCase().contains(name.toLowerCase())).collect(Collectors.toList());
    }
}
```

New Feature: Animal Creation

- Purpose: Allows users to add animals to the shelter
- Dynamic Form: Shows custom fields based on the species selected
- SOLID Principles :
  - S: AddAnimalController only handles UI, AnimalService only handles business logic
  - O: New animal types can be added without changing any existing code
  - L: Each animal type is a valid substitution for Animal thanks to inheritance
  - I: Each animal type defines its own fields
  - D: The controller depends on AnimalService interface not a concrete direct access to the database

Animal Creation Continued

```
@FXML  Louis-TM *
private void handleSave() {
    // Reads UI inputs
    String name = nameField.getText();
    int age = ageSpinner.getValue();
    Species species = speciesCombo.getValue();
    Sex sex = sexCombo.getValue();
    String breed = breedField.getText();
    Size size = sizeCombo.getValue();
    String color = colorField.getText();

    // Validates inputs
    if (name.isEmpty() || species == null || sex == null || breed.isEmpty() || size == null || color.isEmpty()) {
        showAlert( msg: "Please fill in all required fields.");
        return;
    }
```

```
    // Creates the correct subclass
    Animal animal;
    switch (species) {
        case Dog -> {
            boolean isTrained = trainedCheckbox.isSelected();
            String barkVolume = barkVolumeField.getText();
            if (barkVolume.isEmpty()) {
                showAlert( msg: "Please enter bark volume for dogs.");
                return;
            }
            animal = new Dog(name, age, sex, breed, size, color, isTrained, barkVolume);
        }
        case Cat -> {
            boolean isLitterBoxTrained = litterBoxCheckbox.isSelected();
            String temperament = temperamentField.getText();
            if (temperament.isEmpty()) {
                showAlert( msg: "Please enter temperament for cats.");
                return;
            }
            animal = new Cat(name, age, sex, breed, size, color, isLitterBoxTrained, temperament);
        }
        case Bird -> {
            boolean canFly = canFlyCheckbox.isSelected();
            String beakType = beakTypeField.getText();
            if (beakType.isEmpty()) {
                showAlert( msg: "Please enter beak type for birds.");
                return;
            }
            animal = new Bird(name, age, sex, breed, size, color, canFly, beakType);
        }
        default -> {
            showAlert( msg: "Invalid species selected.");
            return;
        }
    }
```

```
    // Sends to service to be saved
    animalService.saveAnimal(animal);
    showAlert( msg: "Animal added successfully!");
    clearForm();
}
```

New Feature: Volunteer Tasks

- Tasks are stored in a separate Task entity, linked to a Volunteer through a one-to-many relationship.
- Tasks can be marked as completed and removed dynamically via the UI.
- The UI displays tasks separately when a volunteer is selected, giving a cleaner overview and better interaction.

Principles Applied: Volunteer Tasks

- **Single Responsibility Principle (SRP):**
- Each class has one job:
  - VolunteerService manages volunteers.
  - TaskService manages tasks.
  - The controller handles UI logic only.
- **Separation of Concerns:**
  - UI, business logic, and data management are kept modular
- **Open/Closed Principle:**
  - New types of tasks can be added without modifying the controller logic

Future Improvement: Observer Pattern

- To **notify volunteers automatically** when new tasks are assigned – without tightly coupling the task logic to the notification system.

Pattern Role Breakdown

- **Subject (Observable):** Task
- **Observer:** Volunteer (or their NotificationService)

How It Works

- When a Task is assigned, it registers the selected Volunteer as an **observer**.
- The Task triggers a notifyObserver() call.
- Only the assigned Volunteer receives the notification — no unnecessary broadcasts.

Future Improvement: Observer Pattern

Benefits / Principles

- **Decouples** task logic from notification logic (*Open/Closed Principle*)
- Follows **Single Responsibility Principle** (each class does one job)
- Promotes **scalability** — more notification types can be added later without rewriting task logic.

Creation Design Pattern

- To make use of the creational design pattern we created factories for our project to quickly and easily create different kinds of entities.
- We created an Animal abstract factory superclass which was extended by the Bird, Cat and Dog factory subclasses to facilitate future factory creation if we chose to use new animals in our shelter.
- These factories are not yet used by our controllers and service classes to easily create entities, but we now know now how to create modular factories that can be used by all our classes and easily modified in future changes.

Future Improvements

- This new version of our project is a great improvement from our last and makes our future development on this project much easier thanks to our refactoring and use of design patterns.
- We unfortunately did not have the time to properly use all our design patterns. We started the logic for our behavioral, strategic and creational patterns, but it can still be improved.
- Due to the time constraint, we did not use these design patterns in the views, since we focused on the design principles and the logic of the project, we did not have time to re-write our views to use our new classes and methods.
- Nonetheless we learned a lot about the project development lifecycle in the making of this project and are eager to continue applying these principles in our future works.