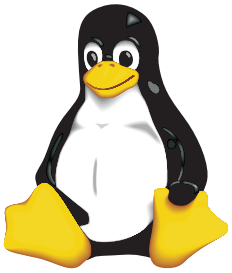

COLLEGE OF COMPUTING TECHNOLOGY - DUBLIN
BACHELOR OF SCIENCE IN INFORMATION TECHNOLOGY

OBJECT ORIENTED CONSTRUCTS

Assessment 2 – CCTZoo

Test Version 1.0

Miguelantonio Guerra - 2016324
Asmer Bracho - 2016328
Adelo Vieira - 2017279

Lecturer: Mr. Mark Morrissey

May 13, 2018

Contents

List of Figures

1 Project description

We have been tasked to create a Zoo management system.

The Zoo has a number of Animals. These Animals are broken down into types:

- Mammal
- Reptile
- Avian
- Aquatic
- Insect

Each Animal has a Zoo keeper that looks after it:

- Zoo keepers are only allowed to care for animals if they are qualified to do so
- A zoo keeper can look after a max of 3 Animal types for a max of 10 animals

The system must allow a user to:

- Search for Animals
- Search for Keepers
- Add new animals
- Add new keepers
- Update animals
- Update keepers

The system must be run on test data before the Zoo will accept it. We are required to have a data set of at least 100 animals and 40 zoo keepers.

2 Requirements gathering and analysis

The system analysis was made from:

- The requirements explained in the project description (Section 1).
- A scenario in which the function of the system (step by step with a typical user) is detailed.

We first determine the main **Use cases** of the system:

- Store Animal data

- Store Keeper data
- Add data (keepers / Animals)
- Update data
- Display data
- Allow to perform a search on the data based on different criteria

In order to build the system requirements we visualized a scenario in which a typical user uses the system (Table 1)

Description			**Coding implications** A more detailed reasoning of the classes is shown in the section 3.
We will design a GUI. When this system is launched:			A view **package view** to allocate all the classes related to the GUI. We will create at least these frames: - View / Search Animal - Add Animal - Update Animal - View / Search Keeper - Add Keeper - Update Keeper
- A login window will appears: * Username * Password * Login button * Exit button			We will only create one **admin** account to test the system.
- After login, the Main Menu will appear showing: * A welcome message * A Log out button * Two sections: ~ **Animals** ~ **Keepers**			- A log out button or a Back button will be implemented in every frame. - When the user press the log out button or close the frame, the system have to save the data created or updated during the session.
	Animals	We have to manage different types of animals. We can also have combinations of types: MammalAquatic, ReptileAquatic	- An Animal abstract class will be create: **Animal** «class» - We will create one interface for each Animal type: *** Mammal, Reptile, Avian, Aquatic, Insect** «interfaces» - In order to create animal objects, we will create animal classes for each Animal type or combinations: *** CreateMammal / CreateReptil / CreateAvian / CreateAquatic** *** CreateInsect / CreateMammalAquatic / CreateReptilAquatic** - Each animal needs to have a unique id number: * An static variable should be used in the Animal class - We also need to manage offspring, vaccines and medications. These Should be attributes of the animal class: * This will need to create classes to manage this attributes. - Tthe Animal objects will have at least these attributes: * exhibitionNumb, specie, name, dateOfBirth, dateOfArrival gender, medication, vacine, offsprings; - Because we will encapsulate the variables, we will need get and set methods: * getName, setName, getSpecie, setSpecie, etc. - We need to create a set of random data (Animals) to test the system:: * In order to store the data created, we will use serialisation: ~ a class responsible of serialized and deserialized the data will be created.
	View	When this option is selected in the GUI, the View Menu will appears	- A GUI frame need to be created to allow the user to deal with this option. * The data will be show in a Jtable
	Add	When this option is selected in the GUI, the Add Menu will appears	- A GUI frame need to be created to allow the user to deal with this Option.
	Update	When this option is selected in the GUI, the Update Menu will appears	- A GUI frame need to be created to allow the user to deal with this Option.
	Search	When this option is selected in the GUI, the Search Menu will appears	We will need to create methods to allow the user to search using different criteria: **bySpecies, byOffsprings, byType, byKeeper**

Keepers			- Tthe keeper objects will have at least these attributes: * id, name, dateOfBirth, * qualification: max of 3 Animal types * keeperAnimals: max of 10 animals objects - Because we will encapsulate the variables, we will need get and set methods: * addQualification, getQualification, getkeeperAnimals, getName, addAnimal etc - We need to create a set of random data (keepers) to test the system:
	View	When this option is selected in the GUI, the View Menu will appears	- A GUI frame need to be created to allow the user to deal with this Option.
	Add	When this option is selected in the GUI, the Add Menu will appears	- A GUI frame need to be created to allow the user to deal with this Option.
	Update	When this option is selected in the GUI, the Update Menu will appears	- A GUI frame need to be created to allow the user to deal with this Option.
	Search	When this option is selected in the GUI, the Search Menu will appears	- A GUI frame need to be created to allow the user to deal with this Option.

Table 1: Scenario in which a typical user uses the system

3 Classes diagram and reasoning behind the choices

In Figure 3.1 we show the *Class diagram* of the CCTZoo System. From this diagram, we can have a very good idea of the overall structure of the system. The relationships between classes is detailed in this diagram as well as the structure of every object in the system.

3.1 MVC design patterns

We first decided to use a Model-View-Controller (MVC) design patterns. A design patterns allows to organize the code in order to make it easier to understand and maintain. It defines the first layer of the structure of the System. In the case of the Model-View-Controller (MVC) design patterns, all the classes in the system will be organized inside three main packages:

- Model: Directly manages the data, logic, and rules of the application.
- View: Can be any output representation of information, such as a chart or a diagram. In our software we developed a GUI.
- Controller: Accepts input and converts it to commands for the model or view.

In the following sections, we are going to try to explain the reasoning behind the building of this structure. We are going to focus in the most important concepts and decisions involved in our design.

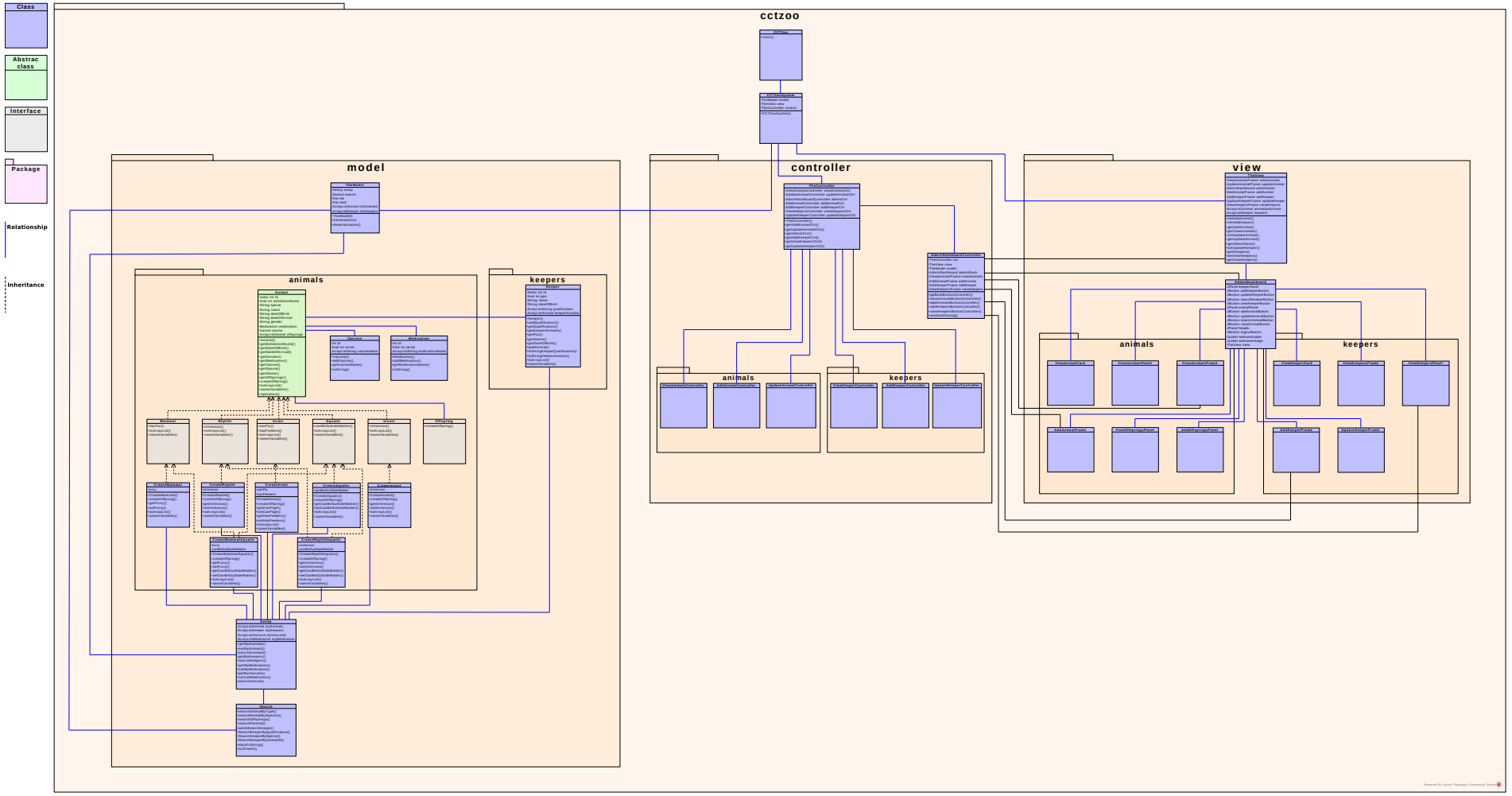

Figure 3.1: Class diagram

3.2 model «package»

3.2.1 animal «package»

We start from thinking about how our most important data (The Animal data) would be managed. The first thing to consider when talking about Animal data is that, as was specified in the Project Description (Section 1), we would need to manage different Animal types (Mammal, Reptile, Avian, Aquatic, Insect). Therefore, each specie (Tiger, Chimpanzee, Crocodile) would belong to at least one type. It is very important to notice that one specie can belong to more than one type. For instance, a *Crocodile* belong to *Reptile* and *Aquatic*. From these reflexions, we could notice that the use of inheritance is indispensable.

So, we would create a class *Animal* from which every simple *type* and *specie* would be child. Therefore, we would create an *Animal class* (abstract)[1] and one *Interface* for each *type* (*Mammal, Reptile, Avian, Aquatic, Insect*). This way, we are able to *extends* the *Animal class* and *implements Mammal, Reptile, Avian, Aquatic, Insect* when defining Animals.

At this moment we arrived to the key point of our design. We needed to decide how we would manage *Species* (e.g. Tiger, Chimpanzee, Crocodile, etc) in our model:

- Should we have a *class* for every single Specie?
  - In this case we would create each Specie *class* by extending the *Animal class* and implementing *interfaces*. For example:

```
public class Chimpanzee extends Animal implements Mammal{}

public class Dolphin extends Animal implements Mammal, Aquatic {}

public class Crocodile extends Animal implements Reptile, Aquatic {}

.
.
.
```

  - We can see the above model is not reasonable since we would have to create as many classes as species we have in the Zoo; which is an approach too complex to manage a system.
- Based on what was explained above, we decide to use a simpler approach in which we have *classes* only for each animal type and for each combination of types. In this approach, each Specie is not represented by a particular class; Specie is an *attribute* (*String*) of the *Animal class*. In this way, a *Dolphin*, for example, is

[1] An *abstract class* allows to define *abstract methods* that can have different implementations in child classes

represented by a *CreateMammalAquatic object* that *extends* the *Animal class*, *implement* the *Mammal* and *Aquatic interfaces*, and have an attribute *String Specie* that is equal to *Dolphin*. Let's see that in code:

```
public abstract class Animal {
    private static int id;
    private final int exhibitionNumb;
    private String specie;
    private String name;
    private String dateOfBirth;
    private String dateOfArrival;
    private int gender;
    private Medication medication;
    private Vacine vacine;
    private ArrayList<Animal> offsprings;

    public Animal() {
        exhibitionNumb = id;
        id++;

    }
```

```
public interface Mammal {
    default String hasFur(int t) {
        return t == 1 ? "Yes" : "No";
    }
}
```

```
public interface Aquatic {
    default String canBeOutsideWatter(int t) {
        return t == 1 ? "Yes" : "No";
    }
}
```

```
public class CreateMammalAquatic extends Animal implements Mammal, Aquatic {

    private int furry;
    private int canBeOutSideWatter;

    public CreateMammalAquatic(String specie, String name, String dateOfBirth, String dateOfArrival, int gender, Medication ↩
         medication, Vacine vacine) {
        super(specie, name, dateOfBirth, dateOfArrival, gender, medication, vacine);

    }
```

3.2.1.1 offspring «class»

An offspring of an Animal is represented for an attribute *«offspring»* of the *Animal «abstract class»*. This attribute is a *ArrayList<Animal>*.

```
public abstract class Animal {
    .
    .
    private ArrayList<Animal> offsprings;
    .
    .
```

Now, to create an offspring, we made a method (*createOffpring*) that is used in each *Animal type «class»* [2]. This method creates a new Animal object of the same type of the *class* through which it is called and add this object to the *offspring attribute*.

```
public class CreateMammalAquatic extends Animal implements Mammal, Aquatic {

    private int furry;
    private int canBeOutSideWatter;

    public CreateMammalAquatic(String specie, String name, String dateOfBirth, String dateOfArrival, int gender, Medication medication, ↩
         Vacine vacine) {
```

[2] *Mammal, Reptile, Avian, Aquatic, Insect, CreateMammalAquatic, CreateReptileAquatic*

```
        super(specie, name, dateOfBirth, dateOfArrival, gender, medication, vacine);

    }

    @Override
    public void createOffpring(String name, String dateOfBirth, String dateOfArrival, int gender, Medication medication, Vacine vacine) ↩
         {
        getOffsprings().add(new CreateMammalAquatic(this.getSpecie(), name, dateOfBirth, dateOfArrival, gender, medication, vacine));
    }
    .
    .
    .
```

3.2.2 keeper «package»

To meet the project requirements, in the *Keeper «class»* we included an attribute that define the *qualifications* of the keeper (Mammal, Reptile, Avian, Aquatic, Insect) and an attribute to define the animals assigned to each keeper. This last attribute add a relationship between the *Animal* and the *Keeper «classes»*. Let's see that in code:

```
public class Keeper implements {
    private static int id;
    private final int pps;
    private String name;
    private String dateOfBirth;

    private int countAnimals = 0;
    private int countTypes   = 0;

    private ArrayList<String> qualification;

    private ArrayList<Animal> keeperAnimals;

    public Keeper() {
    .
    .
    .
    }
    .
    .
    .
```

3.2.3 SetUp «class»

In order to run the system on test data before the Zoo will accept it, we needed to create the test data: our *Animal* and *keepers objects*.

The *SetUp «class»* have the function of create a random set of *Animal* and *keepers objects*. Basically, this *class* creates one *arrayList* of *Animal objects* and one *arrayList* of *keeper objects*:

```
public class TheModel {
    .
    .
    .

    SetUp setup;

    ArrayList<Animal> listAnimals = setup.setListAnimals(200); // Creating at least 200 Animal objects
    ArrayList<Keeper> listKeepers = setup.setListKeepers(50);  // Creating 50 Keeper objects
```

```
    .
    .
    .
}
```

A detailed explanation of this class will be addressed by Adelo and Asmer in their individual reports.

3.2.3.1 Storing the data with serialization After we took the decisions of storing the data created in the *SetUp «class»* we arrived to one of the most important discussions we went through this project: how to storage the Data? We had to options:

- Create a Database
- Use Serialization

Even if we are aware that a database is imperative is this king of systems, we decided to choose serialization for this Test Version 1.0. Like this we could also learn to use this tool since none of the member of our team had worked before with serialization.

We decided to put the methods responsible for doing the *Serialization* and *deserialization* in the *TheModel «class»*; where the data created and stored in the *model «package»* is linked with the other packages of our MVC design pattern.

Therefore, in the *TheModel «class»* we created the methods *serialization()* and *deserialization()*

- **When the program is launched:** In the *TheModel «class»* there is a section of code that first check if there is Data stored in the System[3]:
  - IF There is data:
    - *deserialization();*
  - ELSE:
    - Create data using the SetUp «class»;
- **When the program is closed:**
  - *serialization();*

[3]The data serialized is stored in two binary files: *src/cctzoo/model/animalsData.ser* and *src/cctzoo/model/keepersData.ser*

3.2.4 Search «class»

This *class* contains all the methods that are used to search into the data:

```
public ArrayList<Animal> searchAnimalByType(ArrayList<Animal> list, String[] type, boolean mix){}

public ArrayList<Animal> searchAnimalBySpecies(ArrayList<Animal> list, String species){}

public ArrayList<Animal> searchOffsprings(ArrayList<Animal> list){}

public ArrayList<Animal> searchOffsprings(ArrayList<Animal> list, int exhibitionNumb){}
```

We also included in this *class* a method that take an ArrayList of Animal or Keeper objects and return an String[][] that will be used in the creation of the JTables of the GUI.

A detailed explanation of this class will be addressed by Adelo in his individual report.

3.3 view «package»

In this package we found all the *classes* responsible for the creations of the GUI.

A detailed explanation of this class will be addressed by Miguelantonio in his individual report.

4 Work plan and allocation of roles and responsibilities to each member of the development team

One of the key points of this project has been the planning of a team work. Therefore, after reasoning and decided the Class design, the responsibilities were allocated in the following way:

	Model					**View**	**Controller**
	TheModel «class»	**animals** «package»	**keepers** «package»	**SetUp** «class»	**Search** «class»		
Asmer	X	X	X	X			X
Miguelantotio						X	X
Adelo	X			X	X		X

Table 2: Allocation of roles and responsibilities

It is important to point out that a meticulous teamwork was necessary to reach the requirements of the project. A set of group work sessions were crucial in order to discuss issues and decide key points in the development process.

Declaration

We hereby declare that all of the work shown here is our own work.

Miguelantonio Guerra

Asmer Bracho

Adelo Vieira

Date: May 13, 2018

Bibliography

CODEJAVA.NET : *Java List Collection Tutorial and Examples*. URL http://www.codejava.net/java-core/collections/java-list-collection-tutorial-and-examples.

TUTORIALSPOINT.COM : *isInstance() Method*. URL https://www.tutorialspoint.com/java/lang/class_isinstance.htm.