Continuing the implementation of last chapters' assignment, we will this time try to implement some sort of "database" for our zoo. It could get rather tricky, but with enough patience, you should be able to follow it through with relative ease. The homework will involve working with *.xml* files, however, plenty of examples will be provided on how to deal with *.xml* files in Java.

3.1 Introduction to XML

Usually, data is not stored in text format like *.xml*, *.xls* or any other type of file but stored in binary representation using database systems (MySQL, MSSQL etc. with the exception of NoSQL database systems which actually use string-like formats to store the data). Such database systems fall beyond the scope of this assignment so we will simply deal with storing our data in a simple and casual format like *.xml*.

The main idea is quite simple: have 1 file for each type of (major) entity in our application. For example, all animals are to be stored in a file called *animals.xml* and all employees will be stored in *employees.xml*. Let's first explore the *.xml* format.

```
<?xml version="1.0"?><content>
	<ANIMAL>
		<nrOfLegs>4</nrOfLegs>
		<name>Test Cow 1</name>
		<maintenanceCost>2.0</maintenanceCost>
		<dangerPerc>0.15</dangerPerc>
		<takenCareOf>false</takenCareOf>
		<normalBodyTemp>37.8</normalBodyTemp>
		<percBodyHair>87.5</percBodyHair>
		<DISCRIMINANT>COW</DISCRIMINANT>
	</ANIMAL>
	<ANIMAL>
		<nrOfLegs>6</nrOfLegs>
		<name>Test Butterfly 2</name>
		<maintenanceCost>12.0</maintenanceCost>
		<dangerPerc>0.05</dangerPerc>
		<takenCareOf>false</takenCareOf>
		<canFly>true</canFly>
		<isDangerous>false</isDangerous>
		<DISCRIMINANT>BUTTERFLY</DISCRIMINANT>
	</ANIMAL>
</content>
```

This is a short example of an *.xml* file containing 2 animals: a cow and a butterfly. Notice a few things.

1) Each new animal starts and ends with a specific tag (`<ANIMAL>`)
2) Each animal has all its attributes listed between separate tags *corresponding to their exact name in the java class*
3) There is a curious tag (`DISCRIMINANT`) which apparently only dictates what kind of animal we are dealing with. This will be of *paramount* importance as we will see later on.

Simple enough, right? Well, we will learn how to create such an *.xml* file from an array list of animals, however, there are a few things which we have to do beforehand.

3.2 Java and XML

Java provides a nifty package for dealing with *.xml* files called javax.xml.*;
A note of advice: *do not try to memorize everything you encounter in this subchapter*. There are quite a few new methods and new classes which you'll be dealing with but none which cannot be found on a simple Google search. Simply go along with it and whenever you'll be needing them in the future, the most important thing will be that you'll know what to look for and where to do it.

Our main goal is to transform our POJOs (*Plain Old Java Objects*) into *.xml* format. Basically, every entity which we want to transform should have 2 methods: one which encodes it to an *.xml* format and one which decodes it *from xml* into a POJO. What should immediately come to mind is creating an interface. Why, you may ask? Well it's quite simple. If we want an object *act* in a sort of way and give it some functionality (which may propagate downwards toward its subclasses) the most obvious choice in Java is to make it implement an Interface. Let's name our interface *XML_Parsable*. The code is shown below:

```
package javasmmr.zoowsome.models.interfaces;

import javax.xml.stream.XMLEventWriter;
import javax.xml.stream.XMLStreamException;

import org.w3c.dom.Element;

public interface XML_Parsable {

	public void encodeToXml(XMLEventWriter eventWriter) throws XMLStreamException;
	public void decodeFromXml(Element element);

}
```

You can create a new package where to put this: javasmmr.zoowsome.models.interfaces

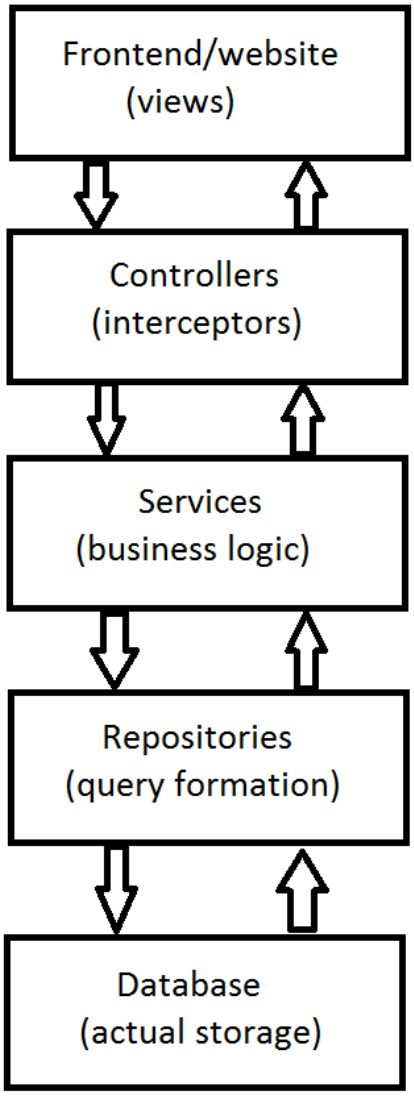
Once we do this, we can simply implement this interface in the *Animal* class. Leave the methods empty for now and let's switch to actually creating and reading from our *xml* files.

3.3 Repositories

What are repositories? That is like asking why ***(spoiler)*** Snape killed Dumbledore or why Malfurion dislikes Illidan. It's simply a part of a bigger story, however, shorty said, (even though I may not do justice to the *Software Architecture* department right now), imagine that each software application is structured on multiple layers based on how "close" or "far" to the end user the code is. Here's a poorly drawn diagram exemplifying what I mean:

The first level is the frontend – everything the users sees and can interact with – and usually, the code for this part goes in a separate package/packages called *views* or however you want to call them (each application may have a different naming convention) but the basic partitioning stays the same.

The second level contains all the *interceptors*, basically, whenever you click on a button or a link, whenever you drag and drop something or maybe submit a form, the first (frontend) classes will delegate that action to the controllers. The controllers are the intermediates between the business logic (what is to be done with your action) and the frontend.



The third level consists of the services which provide the business logic. What is business logic you may ask? Well, for example, consider you are creating a stock-trading application and you decide to sell a few stocks. Well, yes, everything may look very straight forward on the frontend part, you are clicking a button to sell a stock, however this can have quite profound implications on the data. Maybe it was your last stock? Maybe it affects some other stocks? All this is *business logic*.

Finally, repositories are the closest thing you have to the actual database. It would be rather messy to access the database directly from the services which are delegated with dealing with something else, so, the task of forming the queries and accessing the actual database is left

to the repositories. Now, in our case, our so called database consist of one or two files situated on the file system, however, the principle may stay the same! We delegate everything that deals with "database" operations directly to repositories.

Again, and I cannot stress this enough… This was a very crude and basic example of a system architecture. Games for example are **not** structured like this and may have totally different architectures, however you may be surprised at how many websites/mobile application/enterprise applications are built on very similar architectures. And now, coming back from our brief excursion into the world of *software design*, hopefully we gathered enough macro-level information to understand why we are structuring our code this way.

3.4 Zoowsome repositorie(s)

Even though we will slightly modify these repositories in a later chapters in which we will be dealing with *Generics*, let us start off by creating an *AnimalRepository* class inside a new package called `package javasmmr.zoowsome.repositories;`

Your initial class should look like this:

```
public class AnimalRepository {

	private static final String XML_FILENAME = "Animals.xml";

	public AnimalRepository() {

	}

	public void save(ArrayList<Animal> animals) { }

	public ArrayList<Animal> load() { }

}
```

We only have two methods here, *save* and *load*. The first one saves an array list of animals to an *.xml* file (you guessed it, called `"Animals.xml"`) and the second one loads and array list of animals from the same file.

Even though this may seem strange at first, this time I giving you the full code for these two methods instead of having you Google it out. So, without further ado, let's first check out the *save* method:

(space intentionally left blank such that the code may fit into one page)

3.5 Saving to xml

```
public void save(ArrayList<Animal> animals) throws FileNotFoundException, XMLStreamException {
        XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
        // Create XMLEventWriter
        XMLEventWriter eventWriter = outputFactory.createXMLEventWriter(new FileOutputStream(XML_FILENAME));
        // Create a EventFactory
        XMLEventFactory eventFactory = XMLEventFactory.newInstance();
        XMLEvent end = eventFactory.createDTD("\n");
        // Create and write Start Tag
        StartDocument startDocument = eventFactory.createStartDocument();
        eventWriter.add(startDocument);
        // Create content open tag
        StartElement configStartElement = eventFactory.createStartElement("", "", "content");
        eventWriter.add(configStartElement);
        eventWriter.add(end);

        for (XML_Parsable animal : animals) {
                StartElement sElement = eventFactory.createStartElement("", "", Constants.XML_TAGS.ANIMAL);
                eventWriter.add(sElement);
                eventWriter.add(end);

                animal.encodeToXml(eventWriter);

                EndElement eElement = eventFactory.createEndElement("", "", Constants.XML_TAGS.ANIMAL);
                eventWriter.add(eElement);
                eventWriter.add(end);
        }

        eventWriter.add(eventFactory.createEndElement("", "", "content"));
        eventWriter.add(eventFactory.createEndDocument());
        eventWriter.close();
}
```

Read through the code and check out the short comments. You can find all this on Google so there is no point in really dwelling deep into it, however, there are a few lines of interest and they start from the *for* loop.

The most important object here is the *eventWriter*. This object is the top level interface for writing XML documents. Think about it as *the thing you write with* in xml.
If you take a look at that *for* loop and then take a look back at the *xml* example from the first page you will (hopefully) notice some logical links between them. Before doing `animal.encodeToXml(eventWriter);` we place a start element which is the `<ANIMAL>` tag and right after this encoding, we end the xml element by closing with the same `</ANIMAL>` tag. Basically, all that is left here is to add the code in the *encodeToXml* method. A curious thing is that when going through that *for* loop, we consider each animal as being an *XML_Parsable* object, and not an animal. Why can we do that? The reason for which I left it this way will be *revealed* in the *Generics* section of the course.

Before going on with the implementation of the *encodeToXml* method, we will need to do one thing. The following method has to be placed in the *AnimalRepository*:

```
        public static void createNode(XMLEventWriter eventWriter, String name, String value) throws
XMLStreamException {
                XMLEventFactory eventFactory = XMLEventFactory.newInstance();
                XMLEvent end = eventFactory.createDTD("\n");
                XMLEvent tab = eventFactory.createDTD("\t");
                // Create Start node
                StartElement sElement = eventFactory.createStartElement("", "", name);
                eventWriter.add(tab);
                eventWriter.add(sElement);
                // Create Content
                Characters characters = eventFactory.createCharacters(value);
```

```
        eventWriter.add(characters);
        // Create End node
        EndElement eElement = eventFactory.createEndElement("", "", name);
        eventWriter.add(eElement);
        eventWriter.add(end);
    }
```

With these imports at the top of the class:

```
import javax.xml.stream.events.Characters;
import javax.xml.stream.events.EndElement;
import javax.xml.stream.events.StartElement;
import javax.xml.stream.events.XMLEvent;
import javax.xml.stream.XMLEventFactory;
```

And, in every class in which the *encodeToXml* method is to be implemented, simply write the following import:

```
import static javasmmr.zoowsome.repositories.AnimalRepository.createNode;
```

This will enable you to use the *createNode* method which is written in the AnimalRepository class everywhere it is imported by simply writing *createNode(…)* instead of *AnimalRepository.createNode(…).*

This method simply creates an *xml node* like

```
<dangerPerc>0.15</dangerPerc>
                or
<takenCareOf>false</takenCareOf>
```

in the provided event writer based on a given tag *name* and *value* (`dangerPerc` = tag and `0.15` = value) .

An example of using this method is:

```
createNode(eventWriter, "nrOfLegs", String.valueOf(this.nrOfLegs));
```

If you remember in the beginning, I said that each Animal in the *xml* will contain all its attributes, regardless of whether it is a Rabbit, a Cow or a Butterfly, however, it will contain some "discriminant" node which will indicate what animal it is. With this in mind, the first step we need to do is to create the *encodeToXml* method in the *Animal* superclass. All animals share a common set of attributes and therefore there is absolutely no point in placing the creation of the nodes corresponding to these attributes anywhere else in the code. So, with this being said, this is how the *Animal* class method *encodeToXml* looks like:

```
public void encodeToXml(XMLEventWriter eventWriter) throws XMLStreamException {
        createNode(eventWriter, "nrOfLegs", String.valueOf(this.nrOfLegs));
        createNode(eventWriter, "name", String.valueOf(this.name));
        createNode(eventWriter, "maintenanceCost", String.valueOf(this.maintenanceCost));
        createNode(eventWriter, "dangerPerc", String.valueOf(this.dangerPerc));
        createNode(eventWriter, "takenCareOf", String.valueOf(this.takenCareOf));
    }
```

Notice that everything is passed *as a String* to the *createNode* method and all the tags correspond to the name of the class variables. If we were to leave this method implemented only in the *Animal* abstract class, only these common attributes will be present in our xml file. But different types of animals have different attributes, and here comes the best part of having a good class design!
Let's take for example the *Insect* subclass. This subclass of *Animal* contains 2 more attributes `canFly` and `isDangerous`. Because we already created and took care of the common *animal* attributes in the superclass, we no longer need to worry about those, and we can override the *encodeToXml* method in the *Insect* class like this:

```
public void encodeToXml(XMLEventWriter eventWriter) throws XMLStreamException {
	super.encodeToXml(eventWriter);
	createNode(eventWriter, "canFly", String.valueOf(getCanFly()));
	createNode(eventWriter, "isDangerous", String.valueOf(getIsDangerous()));
}
```

Notice how we called `super.encodeToXml(eventWriter);`. This will take care of all the attributes in the superclass and append them to the *eventWriter* and this way, we only have to take care of what is left.

But we are missing one thing, the *discriminant*. There is no way to distinguish which actual implementation of the *Animal* class we are dealing with until now in our xml. Even though we have all the attributes there, we can't possibly check whether it is a Butterfly, Cow or anything else for that manner (or at least not in an elegant way – yet!)

Take for example the *Butterfly* class (or any other subclass of *Insect* in our case). We simply need to write this:

```
public void encodeToXml(XMLEventWriter eventWriter) throws XMLStreamException
{
	super.encodeToXml(eventWriter);
	createNode(eventWriter, Constants.XML_TAGS.DISCRIMINANT,
				Constants.Animal.Insect.Butterfly);
}
```

Aaaaand we are done. Why?
Well, first of all, we call the superclass implementation of *encodeToXml*. This calls the *encodeToXml* method in the *Insect* class. We made sure that inside that method, we also call the *encodeToXml* method from the *Animal* superclass – thus having all the fields covered *plus* a discriminant which will be useful when reading the animals from the *xml* file.

Phew! That is quite some work there. And we're only through with one subclass, *Butterfly!* Well, nevertheless, the others should be quite easily done by copy-pasting a lot of code and changing some names and constants – this is your job!

Going back to our repository, we should now understand what exactly is going on there.

```
for (XML_Parsable animal : animals) {
        StartElement sElement = eventFactory.createStartElement("", "", Constants.XML_TAGS.ANIMAL);
        eventWriter.add(sElement);
        eventWriter.add(end);

        animal.encodeToXml(eventWriter);

        EndElement eElement = eventFactory.createEndElement("", "", Constants.XML_TAGS.ANIMAL);
        eventWriter.add(eElement);
        eventWriter.add(end);
}
```

The call `animal.encodeToXml(eventWriter);` triggers a whole chain of calls through our Animal hierarchy and based on the type of animal (Butterfly, Cow etc.) correctly encodes and appends the xml formatted data into our *eventWritter*.

3.6 Loading from xml

This part will luckily be slightly shorter.
Let's start off with our *load* method which should be placed in the *AnimalRepository* class.

```
public ArrayList<Animal> load() throws ParserConfigurationException, SAXException, IOException {

        ArrayList<Animal> animals = new ArrayList<Animal>();

        File fXmlFile = new File(XML_FILENAME);
        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(fXmlFile);
        doc.getDocumentElement().normalize();

        NodeList nodeList = doc.getElementsByTagName(Constants.XML_TAGS.ANIMAL);

        for (int i = 0; i < nodeList.getLength(); i++) {
                Node node = nodeList.item(i);
                if (node.getNodeType() == Node.ELEMENT_NODE) {
                        Element element = (Element) node;
                        String discriminant =
                        element.getElementsByTagName(Constants.XML_TAGS.DISCRIMINANT).item(0)
                                                                .getTextContent();
                        switch (discriminant) {
                        case Constants.Animal.Insect.Butterfly:
                                Animal butterfly = new Butterfly();
                                butterfly.decodeFromXml(element);
                                animals.add(butterfly);

                        ...
                        default:
                                break;
                        }
                }
        }
        return animals;
}
```

The first thing which interests us the most (after all the opening of the document, normalizing and all that other "bla-blas") is this line:

```
NodeList nodeList = doc.getElementsByTagName(Constants.XML_TAGS.ANIMAL);
```

This basically says: give me all the nodes from the file which contain the *Animal* tag. Going back a little bit, we can see that in our *save* method we enclosed each animal in a special *Animal* tag. Well, here is where we actually use that! Finally…

Well, in short, you iterate through all those nodes which are contained within your *Animal* tag and you end up with an Element (rightfully named *element*) object. This element contains all the attributes of the animal which is contained in that element, and, of course, it contains the *discriminant*. (I told you it will be important). Based on this discriminant and creating a switch statement on it, we know what type of animal we have to create and that in turn is helpful because each animal will have a separate and slightly different *decodeFromXml* method. The principle is exactly the same as the one used when encoding an element.
This will be the *decodeFromXml* method in the *Animal* class:
(Note: setters are being used instead of direct access to the variables using *this* keyword)

```
public void decodeFromXml(Element element) {
        setNrOfLegs(Integer.valueOf(element.getElementsByTagName("nrOfLegs").item(0).getTextContent()));
        setName(element.getElementsByTagName("name").item(0).getTextContent());
        setMaintenanceCost(Double.valueOf(element.getElementsByTagName("maintenanceCost").item(0).getTextContent()));
        setDangerPerc(Double.valueOf(element.getElementsByTagName("dangerPerc").item(0).getTextContent()));
        setTakenCareOf(Boolean.valueOf(element.getElementsByTagName("takenCareOf").item(0).getTextContent()));
}
```

Similarly, the *decodeFromXml* method from the *Insect* class will look like:

```
public void decodeFromXml(Element element) {
        setTakenCareOf(Boolean.valueOf(element.getElementsByTagName("canFly").item(0).getTextContent()));
        setTakenCareOf(Boolean.valueOf(element.getElementsByTagName("isDangerous").item(0).getTextContent()));
}
```

Note two things:

1) We do not need a separate *decodeFromXml* method in the *Butterfly* class unless we have special attributes which only butterflies have – different from all the other insects
2) Please, do not try to memorize this code. Accept as you would accept a thing gotten from a Google search, no more no less, however everything should make sense in your head.

All that is left is for you to "fill the void" and create and implement these methods in all animal classes and subclasses!

3.7 Testing

In your *MainController* simply create an instance of the Animal repository class and play around with the *save* and *load* methods. On *save*, a file *Animals.xml* should be created in the root directory of your project.

3.8 More work

You should implement the same thing for the Employees/Caretaker hierarchy, but of course, in a separate repository. Also, yea, this chapter was about Exceptions… There are a few thrown but I considered them too easy to include in this assignment. Make sure to read on them of course.