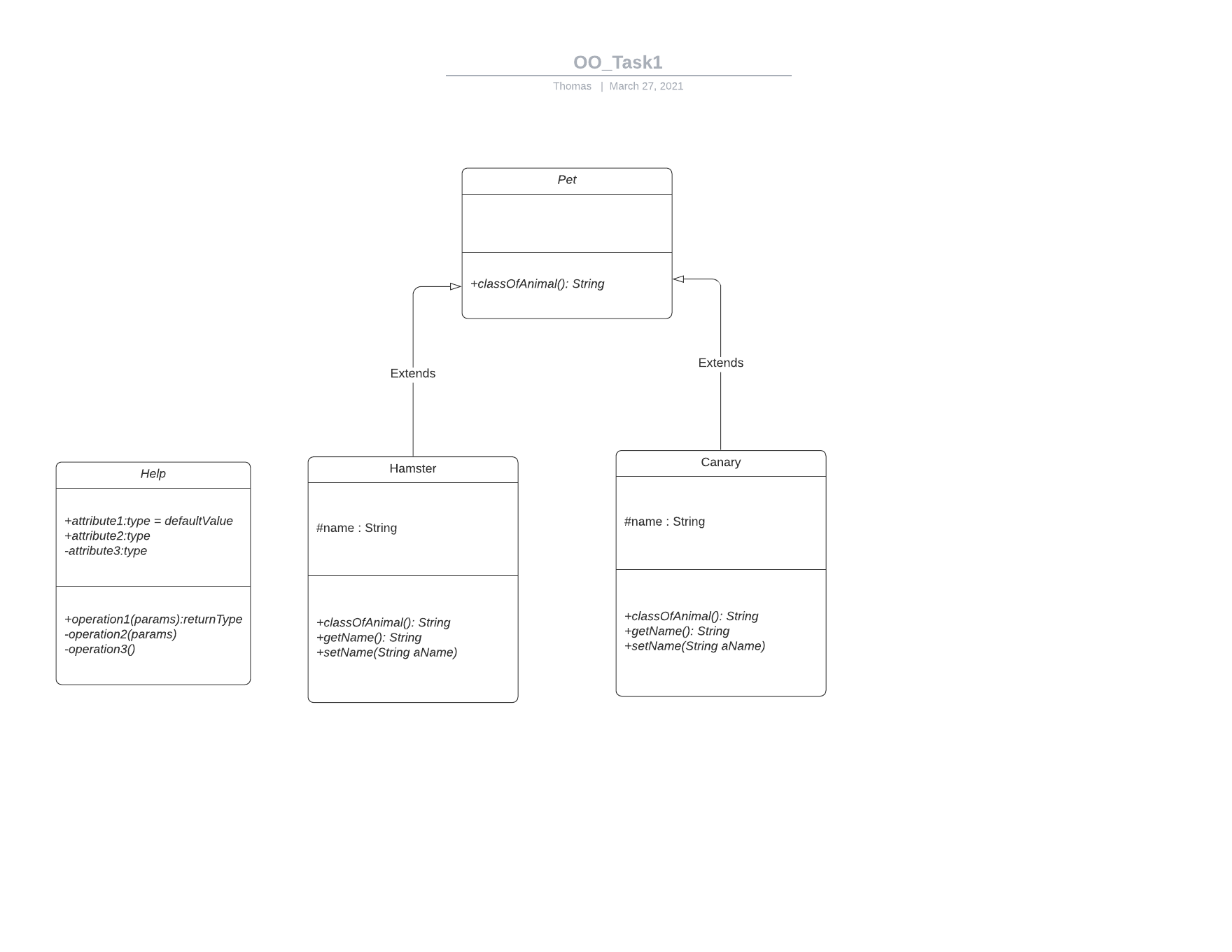
**Task 1.1**



Task 1 UML Diagram for classes Pet, Hamster and Canary with an inheritance relationship

**Task 1.2**

public class Pet {

protected String name;

public void setName(String aName) { name=aName; }

public String getName() { return name; }

public String classOfAnimal() {

return("Pet");

}

}

public class Hamster extends Pet {

public String classOfAnimal() { return("Hamster"); }

}

public class Canary extends Pet {

public String classOfAnimal() { return("Canary"); }

}

I refactored the code by removing the repeated methods and the variable name from class Hamster and class Canary. This code was then written into the super Pet class.

This reduced the amount of code written in the program saving on storage, it also reduced the complexity of the program as now all shared code between the Pet child classes is inherited from the Pet super class. As the code only exists on one place it also is now easier to modify as changing the code in Pet will change it for both Hamster and Canary whereas before changes would have to be made to both possible introducing inconsistencies.

**Task 1.3**

public class Hamster extends Pet implements Vegetarian{

public String classOfAnimal() { return("Hamster"); }

public String food(){return("beans");}

}

public interface Vegetarian{

public String food();

}

The function is polymorphic as the sub class function food overloads the interface food function. The method used is determined at compile time.

**Task 2.1**

The Singleton design pattern ensures that a class only has one instance, and provides a global access point to that instance. This is used as clients may not know they are modifying the same object at the same time.

**Task 2.2**

public class ExampleSingleton{

private static ExampleSingleton singletonInstance = new ExampleSingleton();

private static int accessCount = 0;

private ExampleSingleton(){

// Overwrite Constructor so no new instances

System.out.println("I, the ExampleSingleton, am being created");

}

public static ExampleSingleton getInstance(){

System.out.println("The sole instance of ExampleSingleton is being retrieved");

accessCount++;

return singletonInstance;

}

public int accessCount(){return accessCount;}

}

**Task 3.1**

An adaptor allows objects with incompatible interfaces to collaborate, converting the interface of a class into another interface which the client expects.

**Task 3.2**

public class IncompatibleBook{

private String title;

public void setTitle(String aString){

title = aString;

}

public String getTitle(){

return title;

}

}

**Task 3.3**

public class BookAdapter extends Book{

IncompatibleBook BadBook;

public BookAdapter(){

BadBook = new IncompatibleBook();

}

public void setTitleString(String aString){

BadBook.setTitle(aString);

};

public String getTitleString(){

return(BadBook.getTitle());

};

}

**Task 4.1**

public class LinearCongruentialGenerator implements RandomInterface {

// Generates pseudo-random numbers using:

// X(n+1) = (aX(n) + c) (mod m)

// for suitable a, c and m. The numbers are "normalised" to the range

// [0, 1) by computing X(n+1) / m.

private long a, c, m, seed;

// Need to be long in order to hold typical values ...

public LinearCongruentialGenerator(long a\_value, long c\_value, long m\_value, long s\_value) {

a=a\_value; c=c\_value; m=m\_value; seed=s\_value;

}

public LinearCongruentialGenerator(long seed) {

// Set a, c and m to values suggested in Press, Teukolsky, et al., "Numberical Recipies"

this(1664525, 1013904223, 4294967296l, seed);

// NB "l" on the end is the way that a long integer can be specified. The

// smaller ones are type-cast silently to longs, but the large number is too

// big to fit into an ordinary int, so needs to be defined explicitly

}

public LinearCongruentialGenerator() {

// (Re-)set seed to an arbitrary value, having first constructed the object using

// zero as the seed. The point is that we don't know what m is until after it has

// been initialised.

this(0); seed=System.currentTimeMillis() % m;

}

public static void main(String args[]) {

// Just a little bit of test code, to illustrate use of this class.

RandomInterface r=new LinearCongruentialGenerator();

for (int i=0; i<10; i++) System.out.println(r.next());

// Since RandomInterface doesn't know about the instance variables defined in this

// particular implementation, LinearCongruentialGenerator, we need to type-cast

// in order to print out the parameters (primarily for "debugging" purposes).

LinearCongruentialGenerator temp=(LinearCongruentialGenerator) r;

System.out.println("a: " + temp.a + " c: " + temp.c + " m: " + temp.m + " seed: " + temp.seed);

}

public double next() {

seed = (a \* seed + c) % m;

return (double) seed/m;

}

}

**Task 4.2**