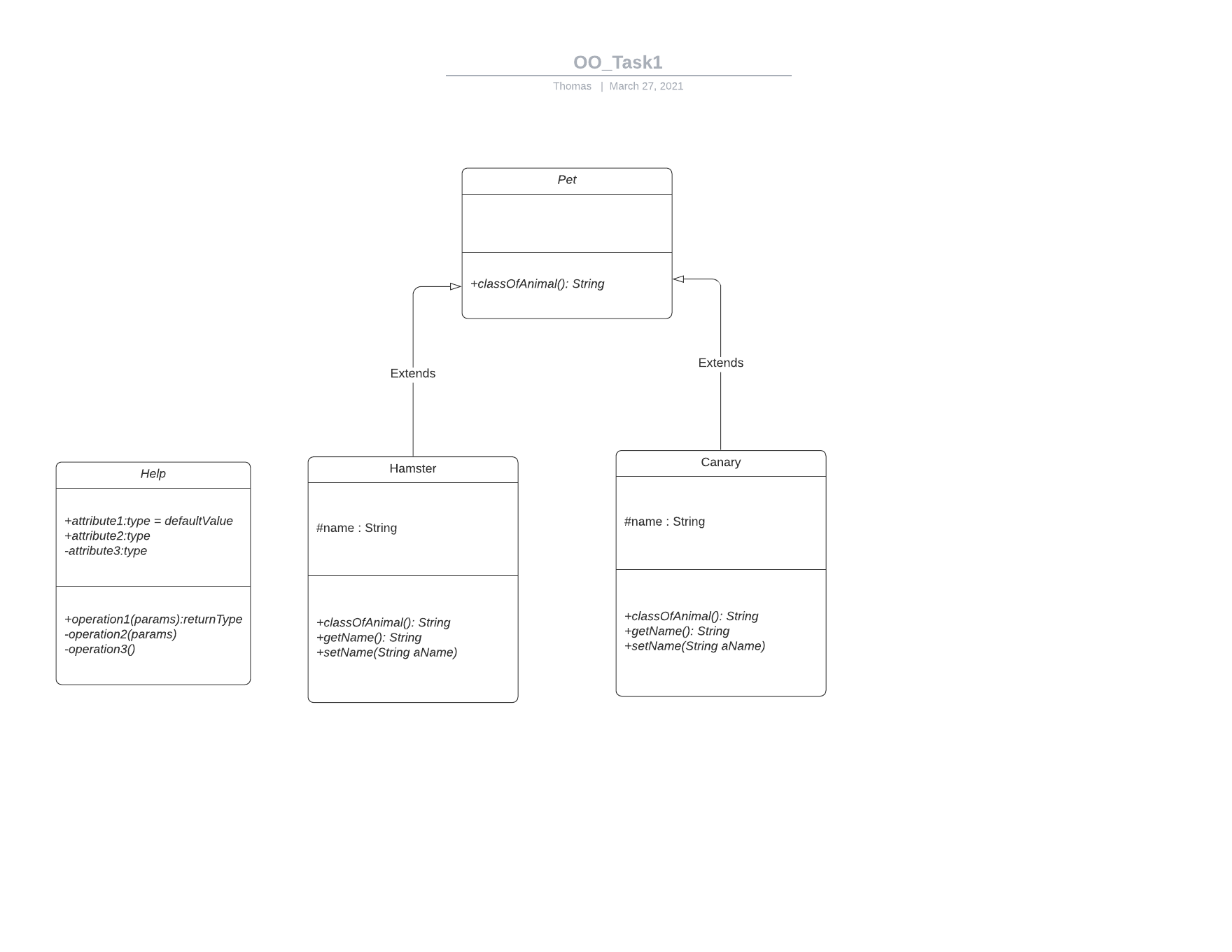
**Task 1.1**



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;

}

}

In addition to modifying LCG file as requested, I also edited the erroneous comment in the Game file. The random interface had not been instantiated correctly as that part was commented out.

**Task 4.2**

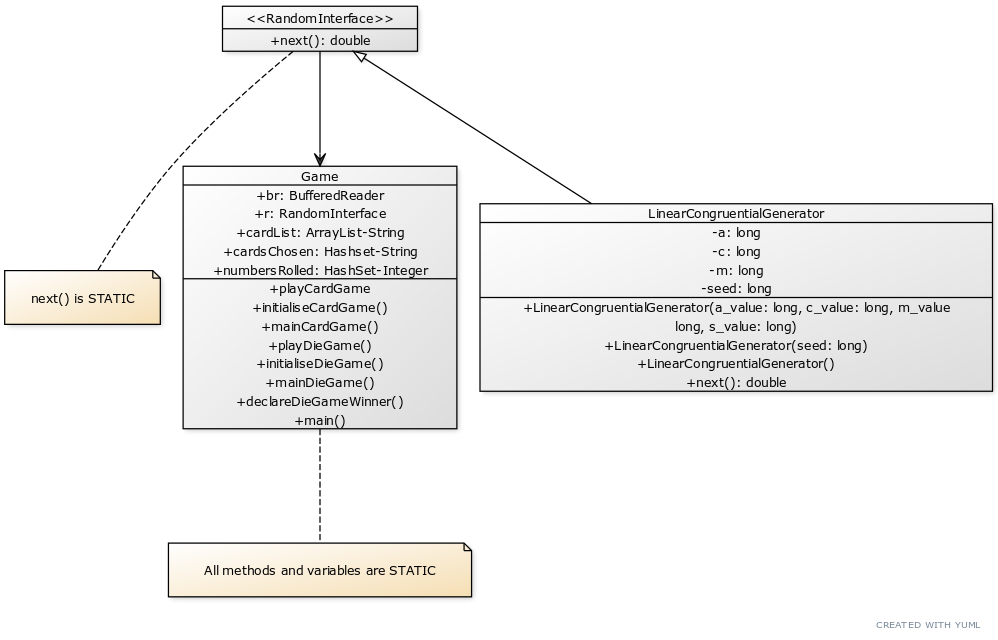
[<<RandomInterface>>;|+next(): double]-[note: next() is STATIC{bg:wheat}]

[Game; +br: BufferedReader; +r: RandomInterface; +cardList: ArrayList-String; +cardsChosen: Hashset-String; +numbersRolled: HashSet-Integer; |+playCardGame; +initialiseCardGame(); +mainCardGame(); +playDieGame(); +initialiseDieGame(); +mainDieGame(); +declareDieGameWinner(); +main()]-[note: All methods and variables are STATIC{bg:wheat}]

[LinearCongruentialGenerator| -a: long; -c: long; -m: long; -seed: long|+LinearCongruentialGenerator(a\_value: long, c\_value: long, m\_value; long, s\_value: long); +LinearCongruentialGenerator(seed: long); +LinearCongruentialGenerator(); +next(): double]

[<<RandomInterface>>]^[LinearCongruentialGenerator]

[<<RandomInterface>>]->[Game]]



Created in YUML. As YUML does not denote STATIC variables or STATIC functions, I have added notes to the diagram to show which variables and functions are static.

**Task 4.3**

This program lets the user play either a card game or a dice game. The user decides which game by inputting either the letter ‘c’ or ‘d’ respectively. Both the games utilise an LCG random number generator.

The card game works by creating a pack of cards in memory, which is shuffled by choosing two cards randomly (number generated by the LCG) and swapping their locations, 100 times.

The user is then prompted to choose two cards randomly which are removed from the deck. The user wins if either of the Cards were an ace, otherwise they lose. The game then ends.

The dice game works by letting the user roll a die, twice. The result of the die is randomly determined using the LCG. The user wins the game if either of the die have the value of 1; otherwise, they lose. The game then ends.

The game communicates to the user through command line, and waits for user input.

**Task 5.1**

|  |  |
| --- | --- |
| **Classes** | **Role** |
| LCG | To produce truly, or as close to truly as possible, random numbers for both games. |
| <<RandomInterface>> | To act as an adapter for the card game and dice game file to the LCG. |
| ChooseGame | To allow the user to select which game they want to play |
| <<GameInterface>> | To act as an adapter for ChooseGame to interact with CardGame and DiceGame |
| CardGame | To allow the user to play the card game |
| DiceGame | To allow the user to play the dice game |

LCG the current version of the file does need changing; I can’t find any fault.

<<RandomInterface>> I will retain the adapter. Keeping the adapter allows for consistency between the different games. For example, only a single line of code in a single file will need to be changed if a different random generator were to be used compared to modifying two lines in two separate files. Also, this reduces the coupling between the classes to loosely coupled.

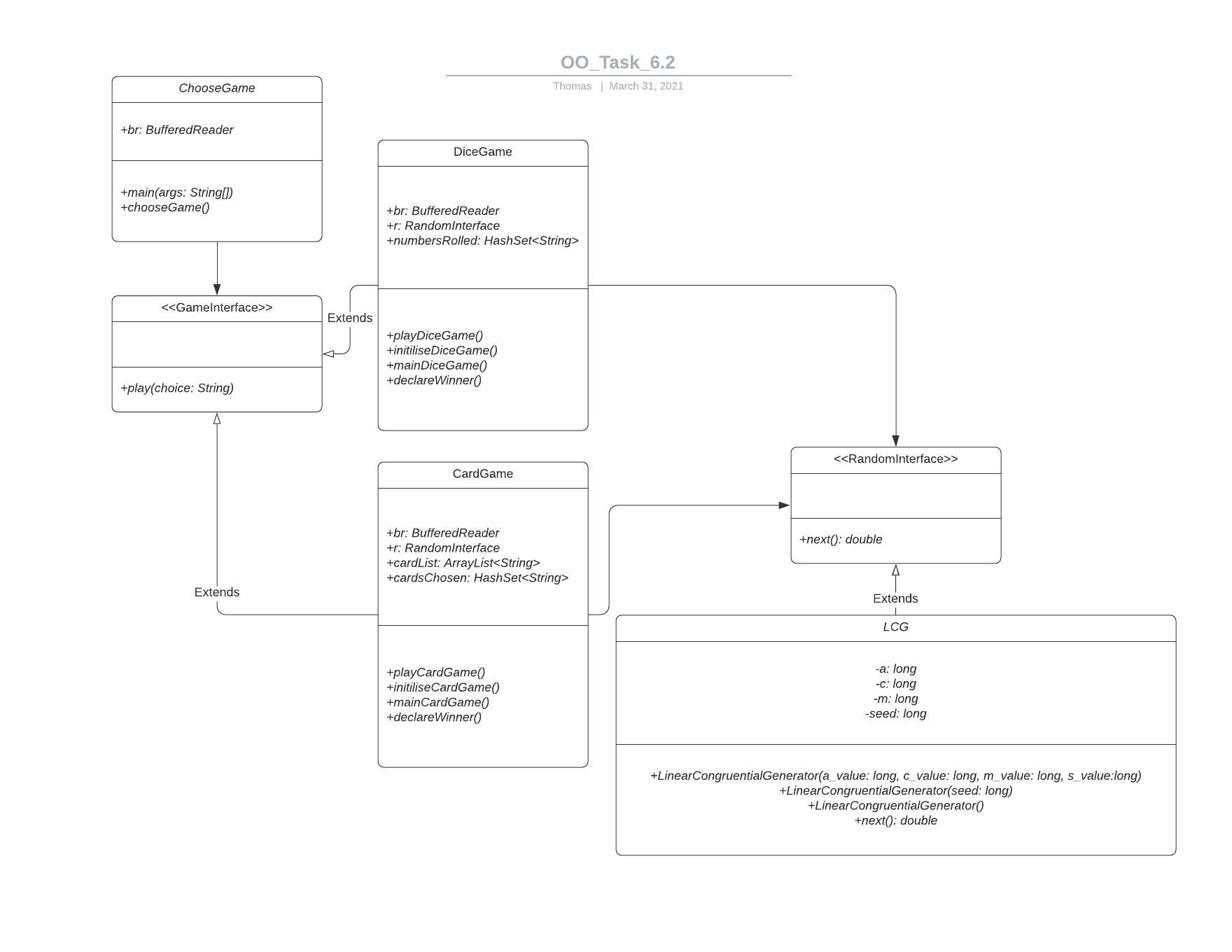
ChooseGame is a new file I will create which will include all the elements involved in choosing which game to play. By placing all aspects of this one task in its own file, I achieve Functional Cohesion.

<<GameInterface>> is a new file I will create which will act as the interface for both games. This interface be used by ChooseGame, to interact without needing to know the specific implementations of either Game files. This reduces coupling, and allows for future expansion: if another game is added to be started by ChooseGame; ChooseGame could use the interface to interact with the new game with much fewer changes.

CardGame consists of the card game related variables and methods. This increases the Cohesion of the design, and any changes to the code need only concern the interface not the parts of the programs. This reduces coupling.

Dice Game follows the same reasoning.

**Task 5.2**



UML diagram for task 5.2 based on the design based on the reasoning explained in Task 5.1

**Task 6.1**

The code is not thread safe as Thread interleaving is occurring. The values of count and val are not synchronised between the threads, meaning one thread could read the data before another had the chance to finish its process updating the data, thus resulting in the data being written twice.

**Task 6.2**