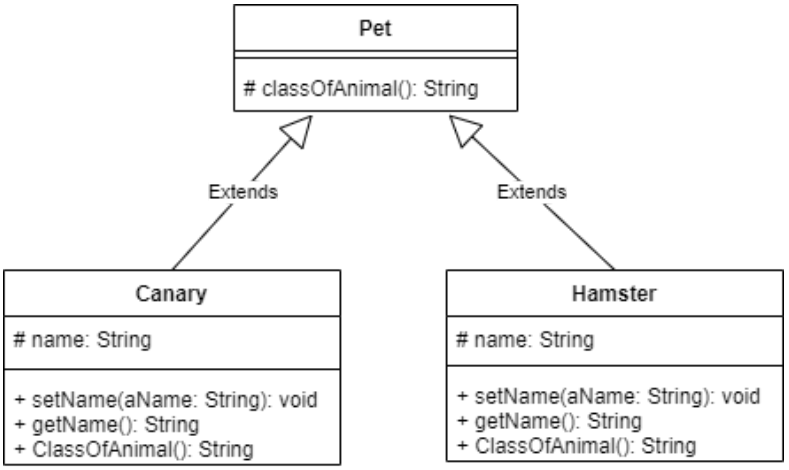
Object-Oriented Modelling and Programming Assignment 2

Task 1

i)



ii) The original classes of Canary and Hamster, were almost identical, with the exception of the final lines of code in each which specified if it was a canary or a hamster. By refactoring using the pull-up method these redundancies are removed. This is done simply by moving the redundant information up to the Pet class, demonstrated below.

Pet.java

```
1  public class Pet {
2    protected String name;
3    public String classOfAnimal() {return("Pet"); }
4    public void setName(String aName) { name=aName; }
5    public String getName() { return name; }
6  }
```

Canary.java

```
1  public class Canary extends Pet {
2    public String classOfAnimal() { return("Canary"); }
3  }
```

Hamster.java

```
1  public class Hamster extends Pet {
2    public String classOfAnimal() { return("Hamster"); }
3  }
```

Client.java

```
1  public class Client {
2    public static void main(String[] theArguments) {
3      Pet p=new Canary();
4      p.setName("Annabel");
5      System.out.println(p.getName());  }
6  }
```

iii) Hamster is a subclass of Pet, which implements the vegetarian interface, this shows multiple inheritance demonstrating the polymorphic behaviour of the Hamster class. In the code, (below for reference)

```
Hamster h=new Hamster();
Pet p = h;
Vegetarian v = h;
h.setName("Cookie");
System.out.println(p.getName() + " eats " + v.food());
```

h refers to a single hamster, Cookie. This hamster can make use of the superclass methods and attributes, such as getName() and food().

Vegetarian.java

```
public interface Vegetarian {
   public String food() ;
}
```

Pet.java

```
  protected String name;
  public String classOfAnimal() {return("Pet"); }
  public void setName(String aName) { name=aName; }
  public String getName() { return name; }
  public String food() {return ("beans");}
}
```

Client1.java

```
public class Client1 {
  public static void main(String[] theArguments) {
    Hamster h=new Hamster();
    Pet p = h;
    Vegetarian v = h;
    h.setName("Cookie");
    System.out.println(p.getName() + " eats " +
    v.food());
  }
}
```

Task 2

i) The purpose of the singleton design pattern is to ensure that a class has only one instance, while providing a global access point to this instance.

ii) Using the UML diagram I was able to implement the ExampleSingleton class so that the ExampleTest could run. First, I initialised the variable accessCount at 0 then I initialised the static variable singletonInstance. Then I created the methods ExampleSingleton, getInstance, and accessCount(), and gave them the appropriate outputs when necessary.

The correctly generated output stream is demonstrated below.

```
C:\Users\hebew\OneDrive\Documents\University\Year 2\CM2307 - Object Orientation,
Algorithms, and Data Structures\Spring Coursework\Task2>java ExampleTest
I, the ExampleSingleton, am being created
The sole instance of ExampleSingleton is being retrieved
The ExampleSingleton has been accessed via the getInstance() method 1 time(s)
The sole instance of ExampleSingleton is being retrieved
The ExampleSingleton has been accessed via the getInstance() method 2 time(s)
```

Task 3

i) The adapter design pattern, demonstrated here as BookAdapter, is used to connect the Book and IncompatibleBook classes which otherwise would not have a direct connection. The purpose of the adapter design pattern, simply put, is to make incompatible objects compatible.

ii) Below I have implemented the IncompatibleBook class using accessors to set and get the title. I used the UML diagram to inform what the methods should do.

IncompatibleBook.java

```
public class IncompatibleBook {
  private String titleString;
  public void setTitle(String aString){titleString = aString;}
  public String getTitle(){return titleString;}
}
```

iii) The code for this class is demonstrated below. I began by instantiating the IncompatibleBook as incompBook which I then was able to reference in the adapter. Then I used the setTitle and getTitle methods from the IncompatibleBook class in the BookAdapter class,

BookAdapter.java

```
1  public class BookAdapter extends Book{
2    IncompatibleBook incompBook = new IncompatibleBook();
3    public void setTitleString(String aString)
•    {incompBook.setTitle(aString);}
4    public String getTitleString(){
5      return incompBook.getTitle() ;}
6  }
```

Task 4

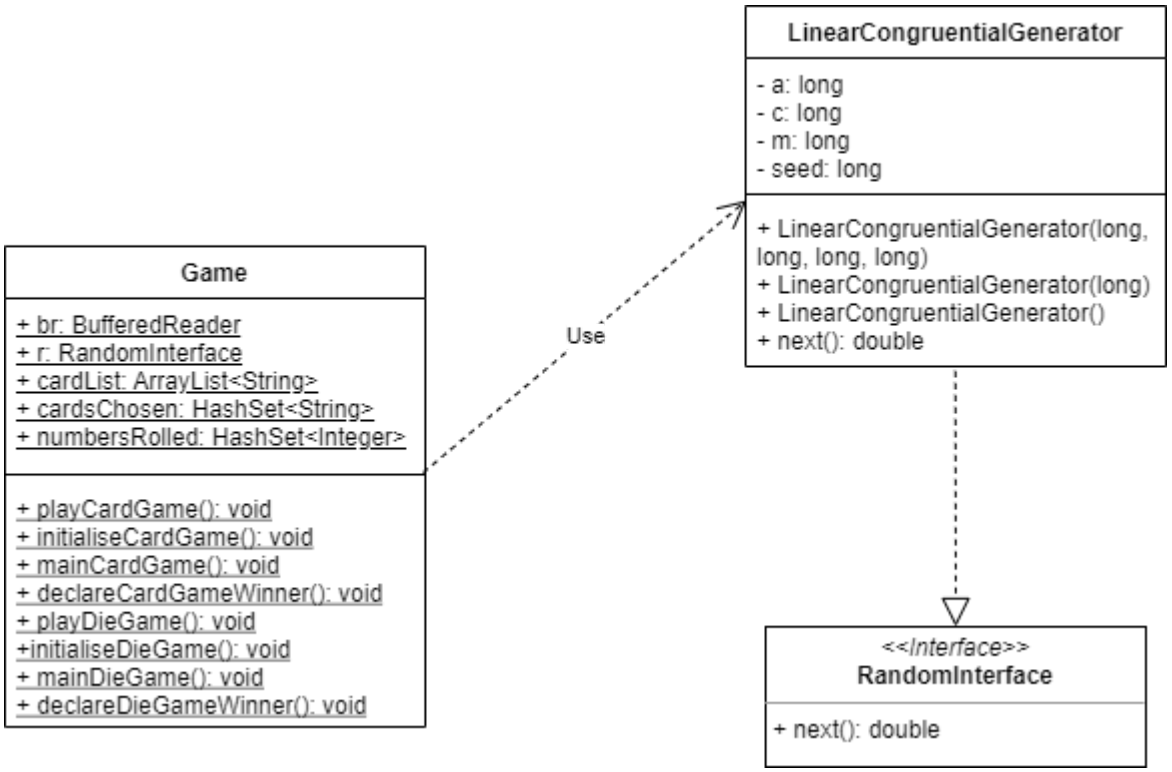
i) To “fix” the problem I altered both LinearCongruentialGenerator.java and Game.java. In LinearCongruentialGenerator.java, I changed the first line from “implements IncompatibleRandomInterface” to “implements RandomInterface” and changed all instances of “getNextNumber” to “next”. In Game. java I edited line 9 to set r using the LinearCongruentialGenerator.

Below are the Die and Card results streams to demonstrate that this change to the program was enough to fix it.

```
 Data Structures\Spring Coursework\Task4>java Game
Card (c) or Die (d) game? d
Hit <RETURN> to roll the die

You rolled 5
Hit <RETURN> to roll the die

You rolled 2
Numbers rolled: [2, 5]
You lost!
```

```
 Data Structures\Spring Coursework\Task4>java Game
Card (c) or Die (d) game? c
[AClbs, 9Dmnds, QDmnds, 3Clbs, 8Spds, QSpds, JClbs, 5Hrts, 6Hrts, 2Hrts, KSpds, 9Clbs, 6Clbs, 6S
pds, 4Hrts, 5Dmnds, 9Spds, 4Spds, 2Spds, 4Dmnds, 7Hrts, 8Hrts, 3Dmnds, KClbs, 4Clbs, 2Dmnds, ADm
nds, 10Clbs, QClbs, QHrts, 9Hrts, JSpds, 3Spds, 8Dmnds, AHrts, 2Clbs, 10Spds, 5Spds, 10Dmnds, KD
mnds, 5Clbs, 6Dmnds, KHrts, 7Spds, 10Hrts, 3Hrts, JHrts, ASpds, 7Clbs, 8Clbs, JDmnds, 7Dmnds]
Hit <RETURN> to choose a card

You chose 9Clbs
Hit <RETURN> to choose a card

You chose 6Dmnds
Cards chosen: [9Clbs, 6Dmnds]
Remaining cards: [AClbs, 9Dmnds, QDmnds, 3Clbs, 8Spds, QSpds, JClbs, 5Hrts, 6Hrts, 2Hrts, KSpds,
 6Clbs, 6Spds, 4Hrts, 5Dmnds, 9Spds, 4Spds, 2Spds, 4Dmnds, 7Hrts, 8Hrts, 3Dmnds, KClbs, 4Clbs, 2
Dmnds, ADmnds, 10Clbs, QClbs, QHrts, 9Hrts, JSpds, 3Spds, 8Dmnds, AHrts, 2Clbs, 10Spds, 5Spds, 1
0Dmnds, KDmnds, 5Clbs, KHrts, 7Spds, 10Hrts, 3Hrts, JHrts, ASpds, 7Clbs, 8Clbs, JDmnds, 7Dmnds]
Cards chosen: [9Clbs, 6Dmnds]
You lost!
```

ii) Below is the UML diagram to show the program that resulted from the modifications above. The IncompatibleRandomInterface is not shown as it is not connected as it is not used nor does it use any of the other classes. The diagram shows the class LinearCongruentialGenerator implementing the interface RandomInterface. Additionally, it shows that the Game class uses both the LinearCongruentialGenerator class and the RandomInterface interface.



iii) The purpose of this program is simple, give the user the ability and option of playing 1 of 2 different games; a card game or a dice game.

Both games are very simple and overall very similar; get a 1 in either game and you win; the games have their own distinct methods that are pretty similar but with minor differences, these create redundancies. For instance, both games get the user to hit the enter key twice, for the card game each of these represents choosing a card

while in the die game it is to signify each roll. The winner declaration methods for both are also very similar, yet more redundancy issues.

The program is made up of just 2 classes and 1 interface. This Game has low cohesion as the Game class does not represent only one type of object and one type only; it represents two - the Die Game and the Card Game. The methods of the program are relatively cohesive, in general they each do one thing and one thing only.

The code uses many static methods, which essentially translates this object oriented program into a procedural one, particularly evident in the Game class.

Task 5

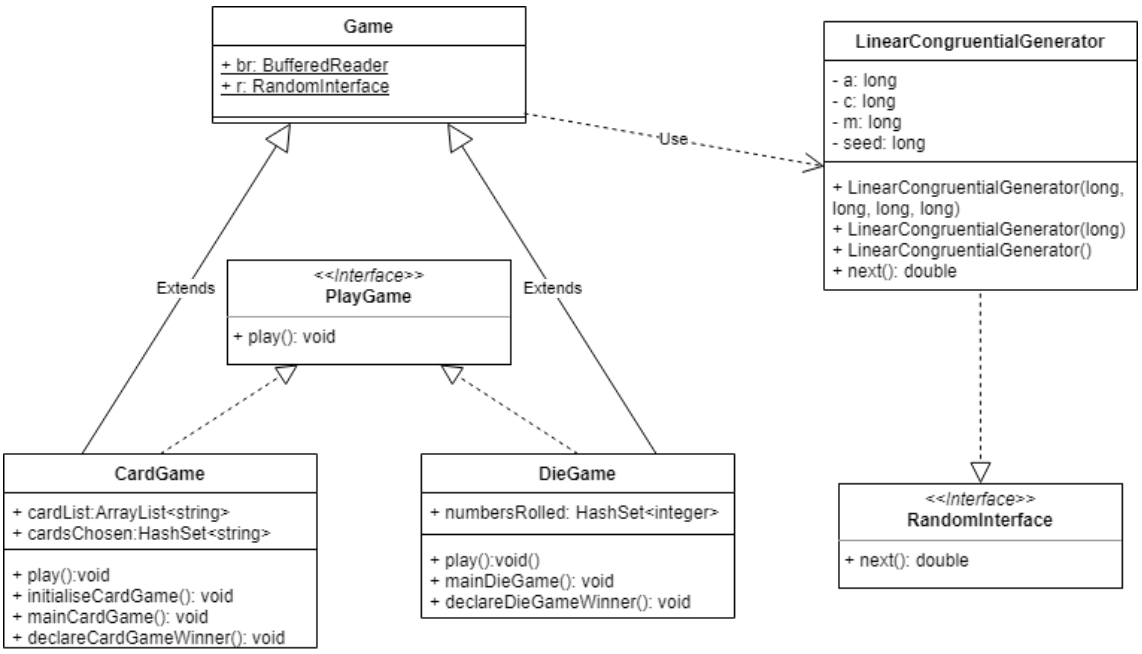
i) For an improved version of the programme the 2 games would be made into their own classes, at very least, that would be extensions of the Game program, this would limit the redundancies.

Currently, the program is tightly coupled and has low cohesion; demonstrated by RandomInterface being called in both the LinearCongruentialGenerator class and the Game class, and by the LinearCongruentialGenerator class being called in the Game class.

Suggestions for new classes:

Class	**Role**	**Why this as a class?**
Die Game	Responsible for all the die game methods and attributes	To separate the two game implementations and to improve cohesion
Card Game	Responsible for all the card game methods and attributes	To separate the two game implementations and to improve cohesion
PlayGame Interface	Implement a Design pattern	To simplify the structure of the program.

ii) UML diagram for the changes to task5:



Task 6

i) The TestThread is not ThreadSafe because a safe thread would be able to call the system simultaneously by multiple threads, while still giving the correct result. After running the TestThread 10 times, the results were: 10000, 9998, 9997, 10000, 9998, 9999, 9997, 9998, 9998, and 10000. As the results each time were not identical this demonstrates that this is not a ThreadSafe program. Another reason this is not ThreadSafe is that it is split into multiple parts, and has created unsafe interleaving, where each thread is getting the same sequence instead of consecutive numbers.

ii) To make the code safe I used the synchronized keyword.

iii) The two strategies I used mean that my code is now thread-safe. The synchronisation keyword allows only one thread to complete a particular task at a time. By using this keyword, the code has been made thread-safe as the inconsistency problem has been solved.

Task 7

i) The synchronized keyword is used in the Philosopher class on Fork objects as the two forks (left and right) are two different threads and this keyword makes it thread-safe. Without the keyword, the two threads would not wait for the other to complete their task before continuing.

ii) For a deadlock to occur, two - or more - threads must not be able to continue as they are waiting on resources from another thread which is also not able to continue as it is also waiting on resources. In this code, there is a deadlock because the right fork thread is within the left fork thread, meaning that the left fork must wait for the right fork to have completed its tasks before the left task can do its own.

iii) By taking the right fork out of the left fork, the deadlock is removed as neither thread is waiting on the other to be able to continue. The threads are still synchronized and so still return the same result, however, they are no longer dependent on each other to progress.