1) ii) Abstract classes can provide a clear hierarchy of classes as they define clear behaviours and attributes for subclasses to inherit. For example, *Pet* is top in the hierarchy, as *Bird* and *SmallMammal* both inherit attributes from it, and the individual animals inherit from those, with each subclass providing additional information to the attributes. This inheritance allows for effective code reuse as the attributes do not need to be manually entered, like in the source version.

However, abstract classes can be inflexible if more attributes needed to be inherited, as classes like *Canary* and *Chinchilla* can only inherit from one abstract class. Using interfaces would mean that we could implement multiple interfaces in one class if needed. An example of this could be where a new type of pet needs to be added that doesn't fit either *SmallMammal* or *Bird*, such as a fish. In this case a *canSwim* behaviour could only be implemented by an interface and not an abstract class.

2) ii) In a single-threaded case, if the Arraylist *cardList* is empty then the *.get()* in lastCardChoice=cardList.get(cardChoiceIndex); would throw an *IndexOutOfBoundsException* which would be caught by the *catch* block below

In a multi-threaded case, if two threads access *dealCard* at the same time and draw the same card from *cardList* using the *.get()* then one thread will remove the card at that index, and the other thread will throw an *IndexOutOfBoundsException* as the card at that index has already been removed by the first thread. The *catch* block will catch this exception

3) a) i) The program contains two games that the user can chose to play, a card game and a die game. In the card game the user can draw two cards from the deck, and if one is an ace, then the user wins. In the die game the user can roll two times and if one roll is a six then the user wins.

The purpose of the program is to make an improved version that makes better use of object-oriented principles such as encapsulation and demonstrate a better programming style, without changing the actual game itself. The original program is designed to have classes that are not modular and easy to understand, and many can have methods split to have more specific responsibilities, such as *playCardGame*. The original program is also designed to have poor readability to be improved on with issues such as code duplication and redundant methods. The original program is also designed to have poor coupling, with several classes that share too high interdependence with each other and could be encapsulated.

3) a) ii) The program has some poorly implemented features. For example, the program does not have an appropriate number of classes as it combines two games into one class. A better approach would be to have each game split into their own class with their own methods and attributes as this would make it more modular and scalable. The methods *playCardGame* and *playDieGame* also contain the full game from initialisation to result, however this should be split into more methods with specific responsibilities to be more readable and maintainable.

RandomInterface represent poor encapsulation and coupling as it is static, therefore can be accessed from any part of the program, exposing the whole program to the internal implementation of both classes and causing potential concurrency issues and incorrect data between threads. It is also tightly connected to the whole *Game* class, meaning they are dependent on each other and changes to either could easily break the entire program. Instead, instance variables should be used improving modularity, testability, and reusability.

The static variables used to keep track of chosen cards and numbers rolled are poorly implemented. These should be instance variables as there can be multiple instances of the game and if static variables are used these variables can be overwritten by other instances whereas if an instance

variable is used then each instance keeps its own state. This is also an example of poor coupling and encapsulation and could cause similar concurrency issues.

The game entities are modelled to some extent as objects, but improvements can be made. Cards are represented as strings in an *ArrayList* and dice are represented as integers in a *HashSet.* These are objects but they do not take full advantage of object-oriented programming. They can be modelled closer to the real life objects by having a *Die* and a *CardDeck* class. This would improve readability and organisation as the logic can be contained inside the class, and it would improve the encapsulation as the behaviour would be easier to modify inside the class with access to data being more tightly controlled.

Another aspect of this program that isn't well implemented is the *initialiseDieGame* which is completely empty, therefore redundant. This harms readability as it just increases the length of the program for no good reason.

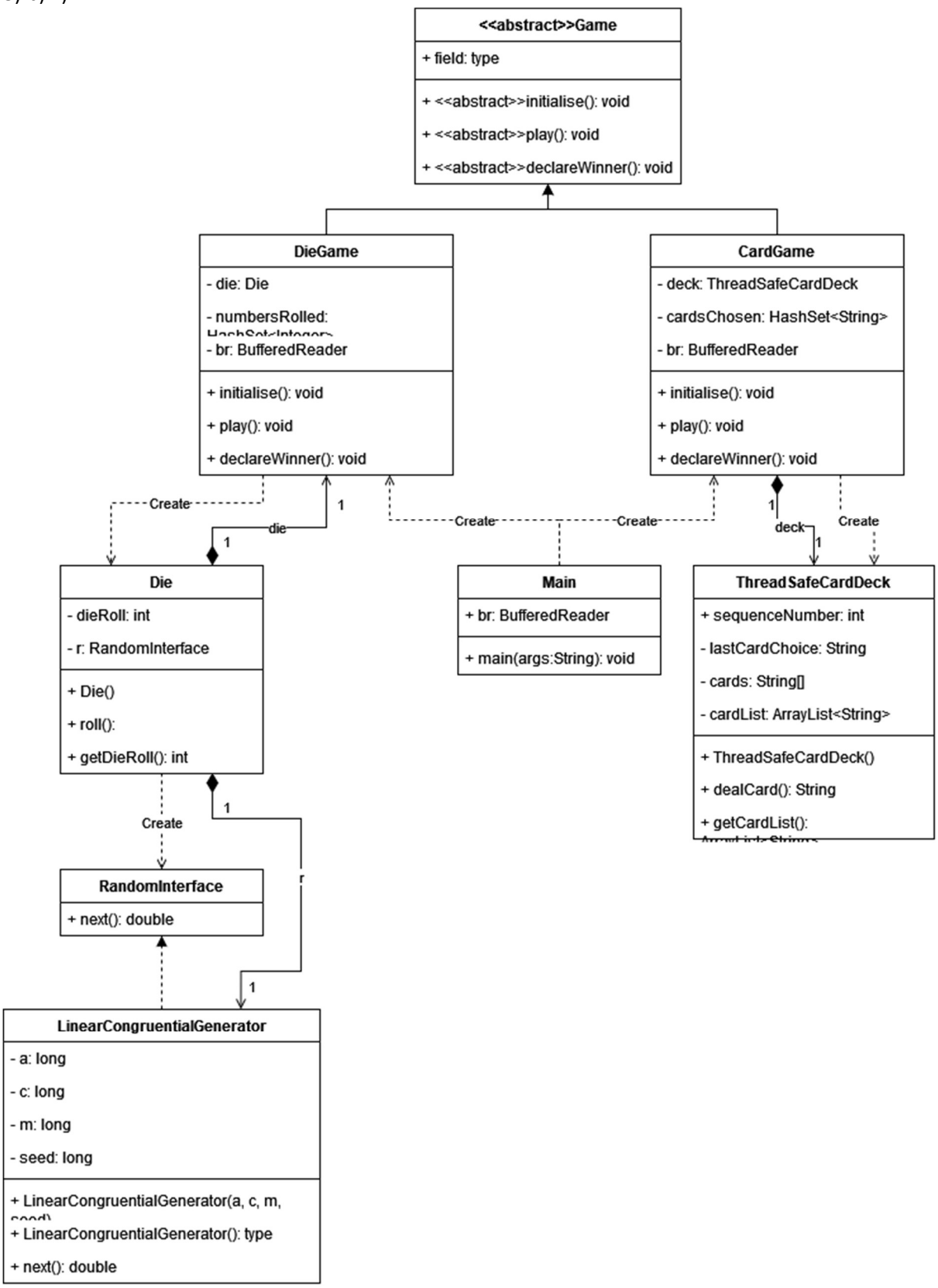
The overall style is functional but does not use objected oriented principles correctly which may lead to issues when multiple instances are created. It is also not designed modularly as the entire game is in one class and the readability can be improved by reducing the amount of duplicate code. However, the use of comments is generally helpful as they explain what each method does, and the formatting is relatively consistent.

3) b) i)

Class	Role of class
Game	Creates methods to initialize, play and declare winner for each game type
CardGame	Implements game interface, encapsulating the logic for the card game
DieGame	Implements game interface, encapsulating the logic for the die game
ThreadSafeCardDeck	Represents a card deck, contains cards, can perform shuffle, and choose cards
Die	Represents a six-sided die, can perform a roll
Main	Initialises the program and menu interface
RandomInterface	Instantiated to generate a random number
LinearCongruentialGenerator	Implements RandomInterface interface, generates pseudo-random numbers

A Template Method pattern will be used where the skeleton of an algorithm is defined in a base class using abstract operations that subclasses override to provide concrete behaviour. This is used as there are parts of the program where code duplication can be avoided as they follow a common template. For example, both the *CardGame* and *DieGame* must contain *initialise, play* and *getResults* behaviours. This can mean that common behaviours are well encapsulated, and a common structure can be enforced while allowing for flexibility for specific behaviours of subclasses.

3) b) ii)



3) b) iii) Most variables that are static can be made private, the cards or die chosen can be encapsulated into the *CardGame* or *DieGame* class so they can not be changed by other areas of the program. The *CardList* variable should also be set to private and put inside the card deck class.

Abstract classes will be used to initialise game methods for initialisin, playing and getting results for the game as individual game classes all have this behaviour therefore can inherit and overwrite these behaviours.

RandomInterface will be better encapsulated as it will only be needed in the *Die* class and therefore will be instantiated there and made private to prevent unwanted access.

4) i) The code creates a class called *Fork* representing each fork on the table, which has a public attribute called *inUse* which is set to False on initialisation. The class *Philosopher* is created and represents each philosopher in the problem. Each *Philosopher* is allocated a thread and they can take two *Fork* objects, their left and right fork. Each *Philosopher* object executes the following loop:

1. Think
2. Pick up left fork
3. Pick up right fork
4. Eat
5. Put down right fork
6. Put down left fork

Each action is done for a random amount of time using the *doAction()* method.

When picking up fork, the philosopher and the fork enter a synchronized block, ensuring that no two philosophers can pick up the same fork at the same time as it only allows one thread to access and each philosopher has their own thread.

In the problem, if all philosophers try to pick up the left fork or the right fork at the same time they could enter a deadlock where they are permanently waiting for the next philosopher to put down their fork. *InUse* is used to keep track of whether a fork is in use, as a fork should only be in use by one philosopher at one time. Each philosopher will pick up the left fork, set the *inUse* to True and then wait for the right fork's thread to become free before picking up. While the philosopher is waiting, no other philosopher will be able to pick up that fork. Once the right fork becomes available (*inUse* set to False) then the waiting philosopher with the left fork picked up will take the right fork.

The *inUse* variable does not affect the execution of the program, it is set to True and False multiple times, however these states do not have any effect on the code.

4) ii) In this case, each philosopher simultaneously picked up their left fork, therefore each thread holds one fork. Then they want to pick up their right fork but cannot as the right forks are already in use by their right neighbours as every thread is waiting on another thread to release their fork. This causes a deadlock where the philosophers cannot proceed and both threads need to wait forever.

4) iii) It prevents the deadlocking as it breaks the circular dependency. In the deadlocking situation described in 4.ii, each philosopher picks up their left fork needs to wait for their right fork to be released, which will never happen as all forks are in use. However, in this situation the first philosopher who believes that his right fork is his left and vice versa will pick up the right fork first and then the left fork, and he will then eat and release the forks. This means that the second philosopher can pick up their left fork without any issues and the cycle can continue without deadlocks.

However, the output does not match the expected behaviour for the first philosopher. When he is picking up his "left" fork it is actually the right fork, therefore the print statements in *doAction()* will

be incorrect. When the first philosopher is picking up the left fork it will print “Picking up right fork” and when he is picking up his right fork it will print “Picking up left fork”

4) iv) In this solution, the last philosopher will eat significantly more, and the first two philosophers eat significantly less. This is caused by the first and second philosophers sharing forks, and the first and last philosophers sharing forks. The fork shared by the first and last philosophers is a “left” fork for both and the fork shared by the first and second is a “right” fork for both. A result, conflicts emerge where certain philosophers wait more and less than others for either fork.

For example the first philosopher eats less as he waits a very long time for the “left” fork but doesn’t wait a long time for the “right”. The second philosopher eats less as he waits a long time for the left fork and a normal amount for the right. The last philosopher eats more as he doesn’t wait a long time for the right fork and waits a normal among for the left fork.