CM2307_OO_21092388

1.

ii. The advantages of using abstract classes in this program are that they allow us to define common behaviours and attributes among related classes, while also providing the flexibility for individual classes to implement their own unique behaviours. Abstract classes also allow for code reusability through inheritance, reducing the amount of duplicate code. However, the disadvantage is that a class can only inherit from one abstract class, limiting its flexibility to adapt to changes or add new features.

Using interfaces instead of abstract classes would allow for even greater flexibility as a class can implement multiple interfaces, providing a wider range of behaviours to be defined. However, interfaces do not allow for implementation of common behaviours, only declaration of methods that implementing classes must implement. This may result in duplicate code and a less organised structure if not properly implemented.

2.

ii. The dealCard() method of CardDeck can throw an exception in the following circumstances:

Single-threaded scenario: If the deck is empty, the method can throw a NoSuchElementException when trying to remove a card from an empty list.

Multi-threaded scenario: If multiple threads are executing the dealCard() method concurrently, it is possible for two or more threads to concurrently remove the same card from the deck. This can cause the deck to be in an inconsistent state and lead to a ConcurrentModificationException being thrown.

To prevent this from happening, the ThreadSafeCardDeck class uses a synchronized block to ensure that only one thread can modify the deck at a time. This ensures that the deck remains in a consistent state and that no exceptions are thrown due to concurrent modification.

3a.

i. The program consists of a card game and a die game. The card game shuffles a deck of 52 cards and asks the user to choose two cards at random from the deck. If one of the cards is an Ace, the user wins, otherwise, they lose. The die game asks the user to roll two dice and records the result. If the user rolls a 1, they win, otherwise, they lose. The entire game is run in the Game class, and the RandomInterface interface uses the LinearCongruentialGenerator class to generate pseudo-random numbers for card shuffling, choosing, and die rolling.

ii. This program is a poorly designed and implemented game with several issues. Firstly, there is a lack of cohesion in the program. The program contains two separate games, but both of these games are included in a single class, making the class too large and complicated. Secondly, the coupling in the program is high. The BufferedReader and RandomInterface are both static variables and are used throughout the program. This could

result in bugs, since multiple methods in the class can change these variables, and the state of these variables could be lost. The use of a RandomInterface is also unnecessary but fine to use.

Furthermore, the object-oriented design is weak since the game entities are not modelled as objects. Instead, the game logic is implemented using static methods, which means that the game's state is not encapsulated in objects, and there is no way to represent multiple instances of the game.

The use of static variables is also problematic. The cardList, cardsChosen, and numbersRolled are static variables, which means that they are shared between different instances of the game. This is not desirable since each game should have its own state, and changing the state of one game could affect the state of another game if they were to run concurrently.

The declareCardGameWinner and declareDieGameWinner methods should return a Boolean value indicating whether the user has won or lost, rather than printing a message to the console.

Overall, the program could be improved by encapsulating the game logic in objects and implementing some kind of design pattern, and reducing coupling by using instance variables instead of static variables. The program could also benefit from better variable naming and a more organised code structure.

3b.
i. The classes I have identified for my improved version of the game are:

ImprovedGame: This is the main class that instantiates the GameFactory and prompts the user to choose which game to play. It also handles printing the game result.

GameFactory: This class follows the factory pattern and creates instances of PlayableGame based on user input.

PlayableGame: This is an interface that defines the play() method, which is implemented by CardGame and DieGame.

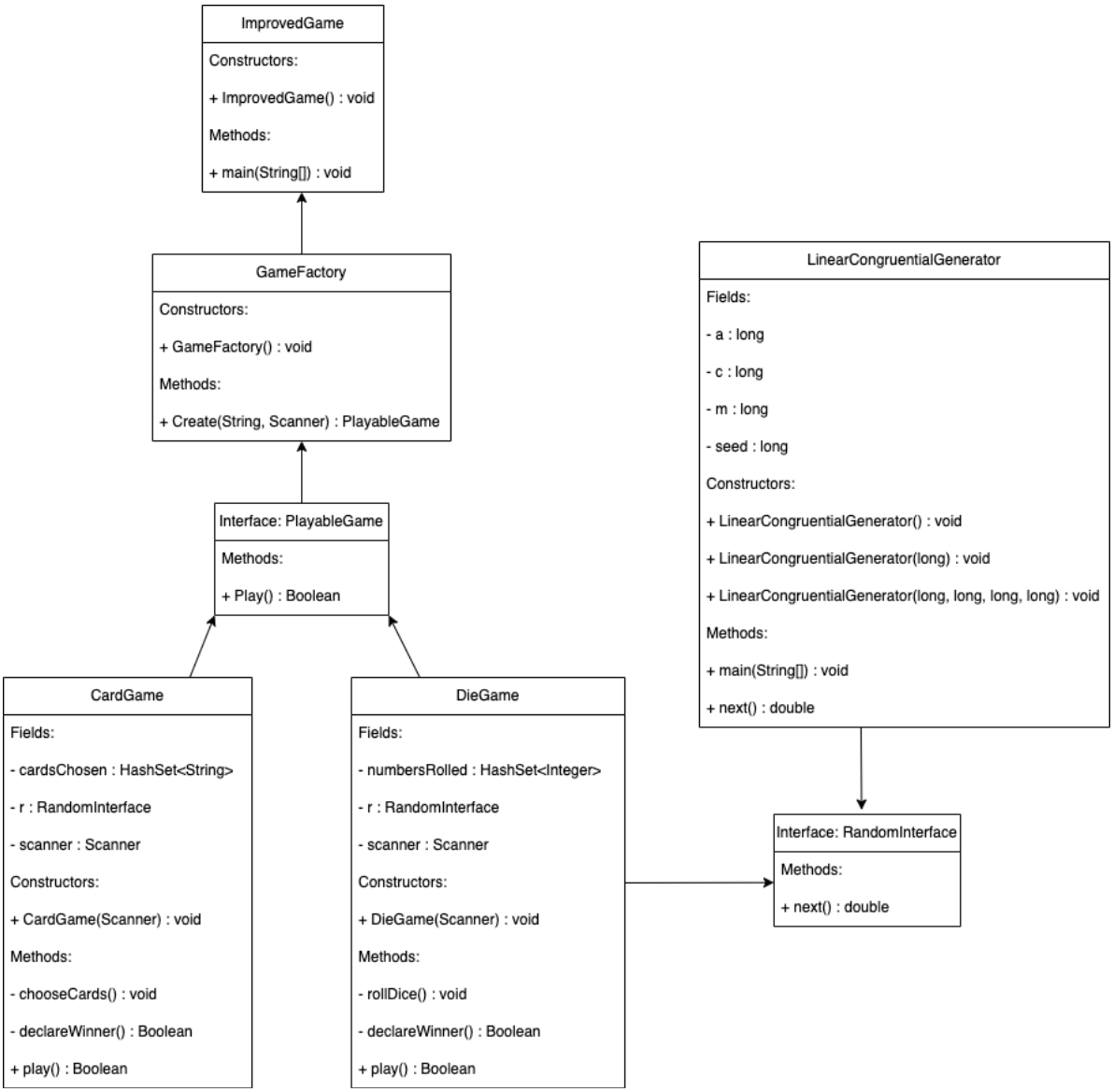
CardGame: This class implements the PlayableGame interface and represents the card game. It uses the ThreadSafeCardDeck class from Task 2 to select two random cards and determine the winner based on whether an Ace card is drawn or not.

DieGame: This class implements the PlayableGame interface and represents the die game. It uses the LinearCongruentialGenerator class to roll two dice and determine the winner based on whether a 1 is rolled or not.

By separating out the game implementations and using the factory pattern, a low coupling between the classes is achieved, since the ImprovedGame class does not depend on the specific game being played. The PlayableGame interface also helps to maximise cohesion by

defining the common method that each game must implement. Additionally, the use of the ThreadSafeCardDeck class helps to minimise coupling and improve thread safety in the card game implementation.

ii.



iii. Some of the changes made to improve the programming style include:

Minimising coupling and maximising cohesion: The classes were designed to be loosely coupled and have high cohesion, meaning that each class has a single responsibility and is designed to be easily reusable in other contexts.

Separating the two game implementations: The original program had two games implemented in a single class, which made it difficult to maintain and extend. The improved version separates the two games into their own classes, each implementing a common interface (PlayableGame) to allow for easy interchangeability.

Use of design patterns: The GameFactory class uses the Factory design pattern to create instances of PlayableGame objects. This allows for easy extension of the game system with new games, without having to modify the GameFactory class.

Use of inheritance: The CardGame and DieGame classes inherit from the PlayableGame interface, allowing them to share common methods and attributes.

Improved code readability: The code has been formatted for readability, with consistent indentation, variable naming conventions, and commenting to explain code blocks and design decisions. I have included descriptive Javadoc's and written the code to adhere to standard Java coding conventions.

4.
i. The program implements the Dining Philosophers problem. The problem involves a group of philosophers who share a round table with one fork between each pair of adjacent philosophers. Each philosopher must alternate between thinking and eating, but can only eat if they have both the left and right forks.

The Philosopher class represents a philosopher and implements the Runnable interface. Each philosopher has two Fork objects, one on the left and one on the right. The Philosopher class has a run() method which implements the logic of the philosopher's behaviour. The method repeatedly loops through the following steps:

Think for a random amount of time.
Pick up the left fork.
Pick up the right fork.
Eat for a random amount of time.
Put down the right fork.
Put down the left fork.
When a philosopher wants to pick up a fork, it checks the inUse variable of the corresponding fork object to see if it is available. If the fork is available, the philosopher sets inUse to true to indicate that it is being used. Once the philosopher is done with the fork, it sets inUse back to false to indicate that it is available again.

ii. The liveness hazard that causes the execution to freeze is deadlock. Deadlock occurs when each philosopher picks up the fork on their left side and waits indefinitely for the fork on their right side to become available. This scenario creates a circular dependency, and the program will freeze because none of the philosophers can proceed.

iv. If one philosopher picks up their right fork first, it will break the circular dependency and allow the program to continue. However, this does not guarantee that all philosophers will get a fair share of opportunities to eat. One reason why this might happen is because of starvation.

Starvation occurs when a philosopher is continually passed over in favour of other philosophers who have picked up their right fork first. In the case where the first philosopher picks up their right fork first, the philosopher to their left may experience

starvation if other philosophers keep picking up the fork to their right before they have a chance to do so.

Since the order in which philosophers pick up forks is randomised, it is possible for some philosophers to experience starvation while others get more opportunities to eat.