DESIGN APPROACH

I used Factory Design pattern in my implementation to cater for cases in the future where a different type of Game is implemented. This way, the factory class can be used to create a new instance of the game and the actual type of game will be determined at runtime. Also implemented Singleton Design pattern so that only one instance of the factory exists in the JVM at a time.

I used SOLID Design principles to ensure that each class has only responsibility to avoid situations where
changes in one functionality affects other functionalities/objects. This approach ensures that if changes are required in future, unexpected side-effects are averted which makes it easier to implement new functionality. Also using Liskov Substitution principle, any method in the SALottoGame object can be called on the superclass object without throwing an exception, as seen in unit tests. The open for extension, closed for modification is applied in the Game interface whereby any new game (e.g. PowerballGame) can implement Game and provide its own implementation to the predefined methods including any other additional methods without requiring any modification to existing code.

The architecture is based on the building blocks of DDD. This ensures that most of the entities are based on objects in any type of lottery game which also supports alteration and improvement on a regular basis.

UNIT TESTS

I implemented the unit tests by testing that the right values are returned. Scenarios include :

1. When a valid game name is given to the factory, it should return a game.
2. When an invalid game name is given to the factory, it should throw and exception.
3. When a set of invalid winning numbers (numbers outside the scope of SA Lotto) are being captured, it should throw an exception.
4. When a set of invalid entry numbers (numbers outside the scope of SA Lotto) are being submitted for the look up, it should throw an exception.
5. When given a winning draw, it should be stored in the current instance of the game.
6. When a set of winning entry numbers are given, it should return the right division.
7. When a set of winning entry numbers that include the bonus number are given, it should return the right division.
8. When a set of losing entry numbers are given, it should return a loss.

ADAPTING TO NEW REQUIREMENTS

If in the future, a Powerball game needs to be implemented, then a new domain object can be created to extend the abstract super class Game and

then provide its own implementation of captureDraw. Also, a service layer can be defined to implement the DivisionService interface so it can provide it's own implementation of getting the winning divisions.

SHORTCOMINGS

One shortcoming of the implementation is that the Game object knows/controls too much in that it encapsulates too much of the business logic, like getting the divisions, which it should rather delegate to some other service.