**Design Decisions:**

To implement this solution, I made use of the Factory Method and the Abstract Factory design patterns. These fall under the creational pattern category of Gang of Four (GoF) Design Patterns.

**Factory Method:**

I defined a GameFactory abstract class. Inside the GameFactory, the CaptureGame method defines what a lottery draw looks like (WinningNumbers). The other method in our factory returns an abstract type called GetDivisions. Subclasses can define the implementation for themselves by overriding the rules of what different divisions could mean to them. This is where the Abstract Factory comes into play.

**Abstract Factory:**

To furthermore decouple the solution, I abstracted the creation of divisions by using the Abstract Factory.

To do this, I defined an interface called IDivisionAbstractFactory. This defines the interface that all concrete division factories (SaLottoDivisions) must implement, which consists of a set of methods for getting divisions. The concrete factory SALottoDivisions implements the different division families.

To get a division, the concrete game SALotto uses the concrete division factory so it never has to instantiate a division object.

This solution is highly scalable and reusable for future use. For instance, I could add another concrete game called Powerball, or the other Lotto games (lotto plus, etc). Since these games will most likely have different rules as to when someone is actually a winner, they could also make use of the GameFactory. The game factory will give them an implementation of the actual game (CaptureGame) to which they just pass through the users selected draw numbers. And then they could abstract the creation of divisions by using the IDivisionAbstractFactory.

My solution adheres to a bunch of principles that allow for flexible designs. By programming to an interface and not an implementation we keep designs loosely coupled. Both factories encapsulate the creation of objects which is a simple way to decouple code.

The intent of the Factory Method is to allow a class to defer instantiation to its subclasses. Additionnaly, the intent of the Abstract Factory is to create families of related objects without depending on concrete classed.

The dependency inversion principle helps to avoid dependencies on concrete types and to stive for abstractions.

**Unit Tests:**

The purpose of the unit tests are to act as a client and call upon the concrete SALotto game, which calls upon the GameFactory to perform the CaptureGame method and then gets divisions for the game instance by calling upon the concrete DivisionAbstractFactory.

In the LottoGameUnitTests solution I defined a class called LottoGameShould. This class is intentionally named this way to ensure that we know it is a unit test class and the test methods within this class are all the methods that the lotto game should be able to achieve / pass.

The test methods all have Fact attributes that get each division from 1 to 8, including no winner.

The Assert.Equal method verifies that the expected result and the actual result match.