Domino Game
Version: 2.0
Authors: Andrés González & Abelardo Valiño

Design Patterns applied:

Singleton:

Applied in Table and Stock.

Singletons were made by making a public method that will return the unique instance of the class. If it was not previously created, the method will call a private constructor. Each class will have a protected static instance of the class.

Table:

As there should be only one table for each game, the singleton design pattern was implemented in the class Table to assure the condition. The code is the following:

```
protected static Table tab = null;

public static Table getTable()
{
        if(tab == null)
        {
                tab = new Table();
        }
        return tab;
}
```

Singleton was implemented in this particular manner, to allow easy sub-classing and thus, to support scalability. In case a sub-class is wanted, in a different packet, the following code can be employed:

```
import softwareGame.Table;

public class SingletonB extends Table
{
        public static Table getInstance()
        {
                if(instance == null)
                        instance = new SingletonB();
                return instance;
        }
        private SingletonB(){ }
}
```

Stock:

As there should be only one stock for each game, the singleton design pattern was implemented in the class Stock to assure the condition. Both code for the implementation of the new singleton class and a possible sub-class, is similar to the previous Table code.

Proxy:

New class ProxyGame was created. A client class Client was also created.

Proxy was implemented in a fashion so that the actual game is not instantiated until a valid

user wants to play.

The proxy will provide an additional functionality to the game. It will only allow the player to play the game, if a valid user name and password is given. Proxy works as an wrapper for the actual game, and it was devised so that the client (where the main function is present), will call the proxy and it will first be asked for a valid user name and afterwards for the corresponding password.

The valid users lists and their passwords are lists hard-coded in the constructor of the ProxyGame class.

Iterator:

Applied in Player.

The iterator design pattern was implemented as a way to access the domino's lists.

Player:

There may be different type of dominoes (for now, regular and “princess”). An iterator is applied to allow this two, but also to add generality to the code for future uses.

Generics:

Applied to Domino, Stock, Table, Player, ProxyGame, Game, Princess.

It was applied in the attributes of the previous classes. It provides a certain degree of generality that allows the code to add new features in a smoother manner.

In order to exemplify the generality, the class Princess was created. With it, a new functionality was added to the game. The game now gives the valid user the possibility for two types of game (different dominoes). At the beginning of the game, after introducing both a valid user name and it's password, the user may choose between regular domino chips and or princess kind.

Exceptions:

Applied to Game and ProxyGame.

Various exceptions were implemented to further improve the code. Specifically, within the treatAnswer method in the class Game, exceptions were used to manage when the player puts a domino in the left/right side of the table. If the player tries to put a domino in the wrong side, an exception will be thrown and the catch will try to put the domino in the other side of the table.

Other changes:

Syntax changes were made in most of the classes to make them more readable.

Accessibility of several attributes and methods in different classes, were changed.