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 the instance was not previously created, the method will call a private constructor that will in fact, create the unique singleton instance. Each class will have a protected static instance of the class that will work as the singleton.

Singleton class was created to ensure that only one instance of the class could be created. In version 1, it was possible for a player to create, for example, several stocks in order to improve his chances of winning the game. Utilizing singletons is a way to further certify the game is being played the way it should.

Additionally, another improvement from v1, is the use of lazy initialization. The table and especially the stock (one of the most expensive initialization in this game), would not be initialized until it is required.

One last improvement from v1 is the now uncomplicated way of sub-classing. Below, it is provided a sample code of how it is done (and how easy it is). All the benefits of sub-classing can now be achieved with ease.

Stock:

As there should be only one stock for each game, the singleton design pattern was implemented in the class Stock.

Table:

As there should be only one table for each game, the singleton design pattern was implemented in the class Table. 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. 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(){ }
}
```

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.

The main improvement over v1 is the fact that the game would not start until a valid client makes a request, thus all of the resources that are required for the making of the game would not be used until a valid player is logged.

A secondary improvement is that, in v2 only valid players can actually play the game. This may be a very desirable attribute that is thought to improve the game greatly.

Finally, since the proxy is already created, scalability is enhanced. If, in the future, some extra functionalities are wanted, before the actual start of the game, the coding would not be as difficult.

Iterator:

Applied in Player.

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

Iterator were implemented in v2 to improve the accessing of elements. The principal benefit of implementing iterators is the amelioration of the extendability o the code. For now, there is no technical improvement over the way the list in Player was being accessed (normal way to access a list of certain elements). The benefit will come, if in the future, the player's hand is desired to be made up of more than one class (a game with regular dominoes and princess dominoes, at the same time). Iterators make the code less sensitive to changes in the data structures.

Another benefit of the iterators in v2, is that they improve readability of the code, and also offer additional restriction on access.

Finally, the use of iterators is key if true generic programming is wanted.

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.

The main benefit of generics is that they greatly increase re-usability. With generics, for example, different domino chips can be created and easily used. It makes the code remarkably more resilant to change. The previous are reasons why generics were implemented in v2.

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.

A few exception were implement in v2 to raise the code's resilience. Certain scenarios were disruption of the normal flow of the program may occur were thought, and exceptions were coded to handle them.

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 & Notes:

Most design patterns were implemented to increase the maintainability and extendability of the code.

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.

For each of the classes's changes, a considerable amount of hours were dedicated to research the pattern's benefits, restrictions and implementation, and afterwards in deliberating if the amount of time required to implement them was surpassed by the benefits they provided.

Per each pattern chosen, some additional hours were invested in the implementation of the pattern and re-factorization of the whole code.

An average of 16h were devoted to documentation.