GAME – CIRCUS OF PLATES

Computer and Systems Engineering Dept.
Alexandria University , Egypt.
Second Year Projects, Fall Semester
Object Oriented Design (CS203)

ASSIGNMENT REPORT

FY 2013

By: 1)Bahie Eldeen Samir [16]

2) Mohamed Elfeki[60]

ContentsOverview **Error! Bookmark not defined.**

GAME – CIRCUS OF PLATES

Game Description

QUICK REVIEW

Circus of plates is a two player-game in which every player carries two stacks of plates, and there are several colored plates falling down and he tries to catch them, if he manages to collect three consecutive plates of the same color, then they are vanished and his score increases, the player who gets more score at less time wins. You are free to put rules to handle if the two players stand at the same place; also you can modify the rules of ending the game.

OUR IMPLEMENTATION

In order not to go further with useful details we determined to go through to reach the core of this Assignment. We neglected the details that not very important for the core of this course. We assumed that the Course of Object Oriented Design will not care about the other stick of the player or the 4 shelves, and in order to make a shortcut to the core of this course we concentrated about the design and the direct implementation of the High –level design. We made the necessary designs for this project which will described afterwards.

DESIGN ISSUE

In order to reach the meaning of this assignment we made our whole concern to the design , starting with the pre-implementation design which was the final design but forgot some small details, which indicate that we made a very good work in our first design to get over the expected obstacles that met any other team.

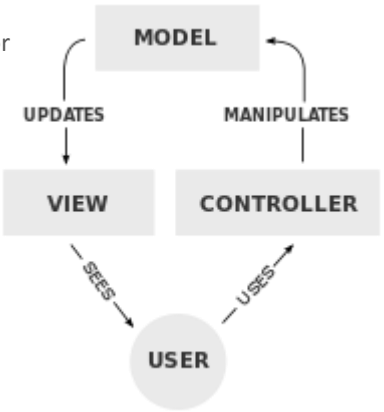
Object Oriented Design

On our Implementation of the game we concerned mostly with the design issues that can obstacle the software engineers while developing some applications or interfaces. As you may think the most difficult problem was making the desired compatibility between different design patterns to reach the goal of implementing the pre-studied patterns in our course. This obstacle was not that easy especially that there is no specific single implementation of each design. On the following lines, We will try to describe shortly the design patterns of our implementation, how we implemented them, and how we connected them together with a short terms.

1. **MVC** (Model- View Controller)

According to *Wikipedia* **Model–view–controller** (**MVC**) is a software pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. The central component, the *model*, consists of application data, business rules, logic, and functions. A *view* can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The third part, the *controller*, accepts input and converts it to commands for the model or view.

As mentioned before, this design is used as a core design , So **our implementation** based on the discrimination between different attitudes or perspectives of the Program, So we made a class called “Control” that was the single operating class for other, the only one has the main method that creates the whole program, which can manipulate the data in the model and in the same time show this modifications on the view. This class besides creation of the program it has multiple thread to still operating the game while the user is playing.



2. Object Pool

The **object pool pattern** is a software creational design pattern that uses a set of initialized objects kept ready to use, rather than allocating and destroying them on demand. A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished, it returns the object, which is a specific type of factory object, to the pool rather than destroying it.

Object pooling can offer a significant performance boost in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instances in use at any one time is low. The pooled object is obtained in predictable time when creation of the new objects (especially over network) may take variable time.

The Creation of the objects was a main role of the game as it creates shapes continuously. The shapes can not be instantiated each time I need a new shape because it make a hug overloading on java virtual machine and consequently the behavior of the program. So we used the object pool to reuse the objects every time we need to re instantiate it.

3. Iterator

Iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

For example, the hypothetical algorithm *Search For Element* can be implemented generally using a specified type of iterator rather than implementing it as a container-specific algorithm. This allows *Search For Element* to be used on any container that supports the required type of iterator.

The Iteration was a continuous role of the game and it is the base of all game development as the game is required to still open for the user who is gaming. So the iteration on the generic data structure was used to create objects continuously as the result set contains it and to print it as well.

The iterator is used mostly on the creation of the shapes on the game and on the creation of the shapes on saving an loading and of course on printing on the GUI.

4. Dynamic Linkage

The static linking model is simple to implement and use. This is its main advantage. However, it's pretty limiting. When you statically link a program, the resulting executable file must contain all the data and code that the program may ever need. Consequently, it can become ridiculously large.

In addition, changing or extending a program's behavior entails recompilation and relinking. Customers must therefore close the application before they install a new version of the executable whenever you supply an update. This may not sound like a serious problem until you think of your favorite Web browser, word processor, or operating system. Would you agree to reinstall any of these from scratch whenever a new version, bug fix, or update arrives?

Dynamic linking offers several advantages over static linking: code sharing, automatic updates, and security.

We used it to invoke the classes of the shapes that are used in our model.

5. Snapshot

The **memento pattern** is a software design pattern that provides the ability to restore an object to its previous state.

The memento pattern is implemented with three objects: the *originator*, a *caretaker* and a *memento*. The originator is some object that has an internal state. The caretaker is going to do something to the originator, but wants to be able to undo the change. The caretaker first asks the originator for a memento object. Then it does whatever operation (or sequence of operations) it was going to do. To roll back to the state before the operations, it returns the memento object to the originator. The memento object itself is an opaque object (one which the caretaker cannot, or should not, change). When using this pattern, care should be taken if the originator may change other objects or resources - the memento pattern operates on a single object.

Classic examples of the memento pattern include the seed of a pseudorandom number generator (it will always produce the same sequence thereafter when initialized with the seed state) and the state in a finite state machine.

In our implementation we used it to set the coordinates of each shape while he is moving and when saving.

6. State and Observer

The **state pattern**, which closely resembles Strategy Pattern, is a behavioral software design pattern, also known as the **objects for states pattern**. This pattern is used in computer programming to encapsulate varying behavior for the same routine based on an object's state object. This can be a cleaner way for an object to change its behavior at runtime without resorting to large monolithic conditional statements

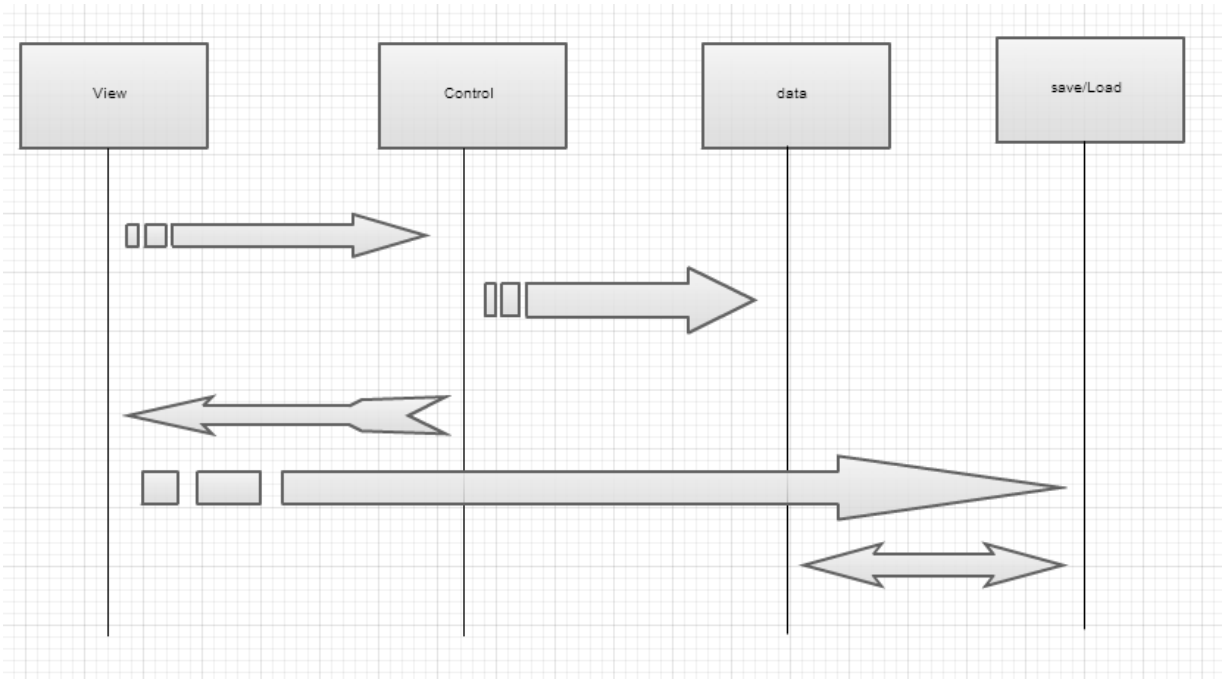
In our implementation we used them to define the current state of the shape and whether to move right or left or whatever.

Beside working on that Design Patterns

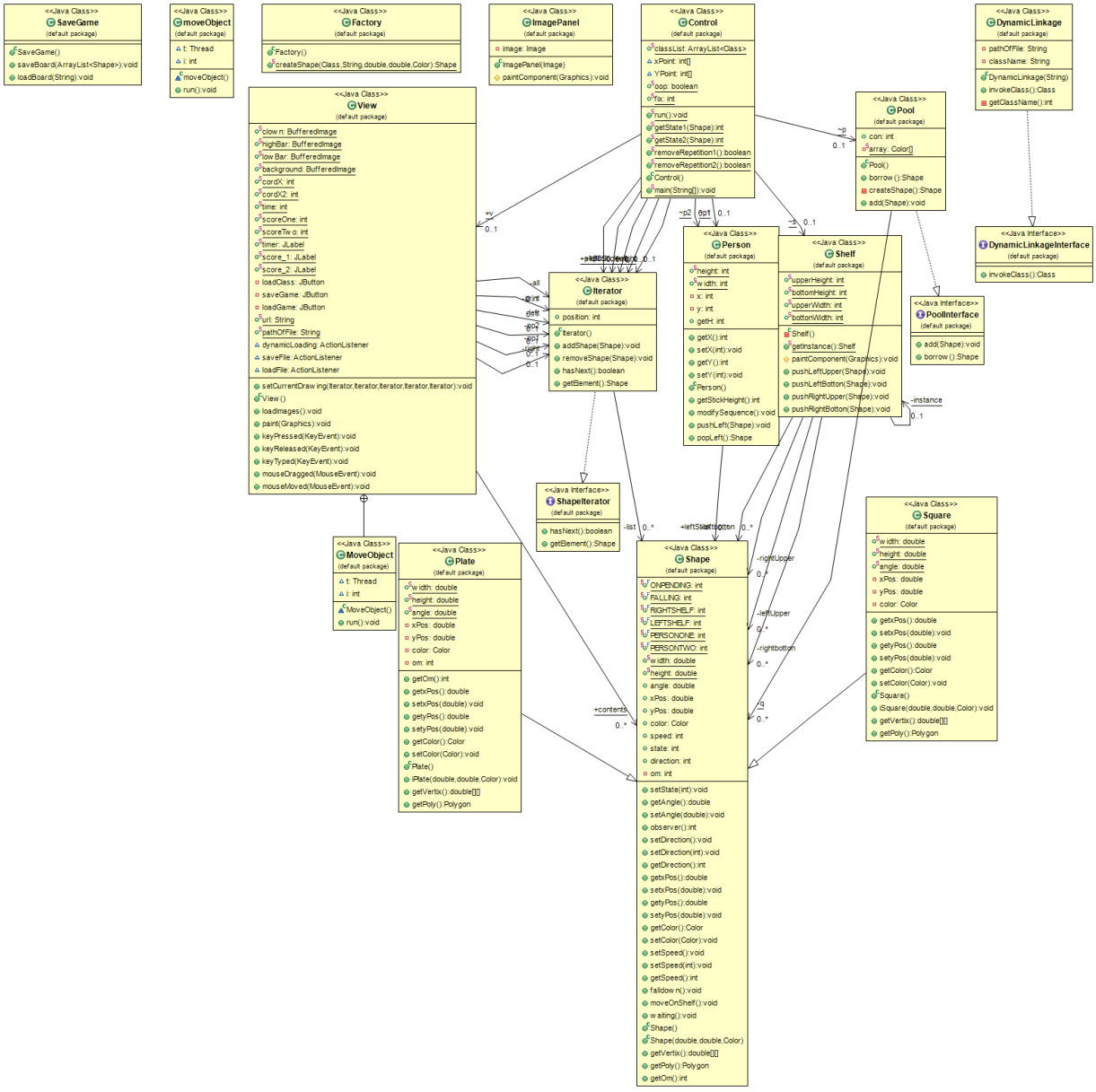
We used **Skelton** on single creation and the **Factory** for multiple instantiation of the objects.

DIAGRAMS

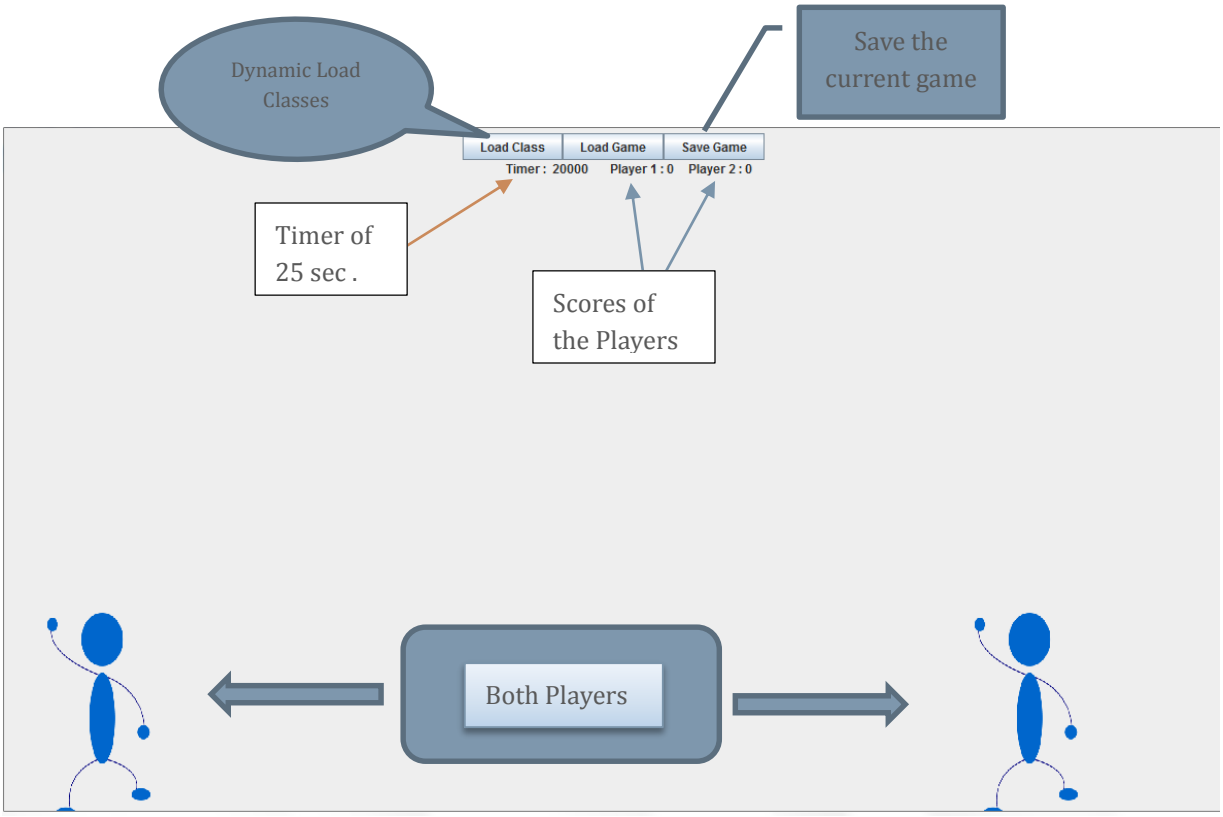
1-Sequance Diagram



2-Class Diagram



User Guide:



Our game is very simple, even the children can easily play it.

1. Load the specific Shapes using the Dynamic load classes
2. Try to catch three plates of the same color up to gather to increase your score but be aware that if you collected more than 15 shape on not the same color you lose the game, so be careful.
3. The person who can collect the maximum three points in the same time wins the game. The game duration is 25 seconds.
4. You can control one of them using the left and right arrows on the keyboard and the other using the direction of the mouse cursor.