A Pattern for Distributing Turn-Based Games

James Heliotis
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, NY 14623-5608
+1 (585) 475-6133
jeh@cs.rit.edu

Axel Schreiner
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, NY 14623-5608
+1 (585) 475-4902
ats@cs.rit.edu

ABSTRACT

A common choice of applications used in introductory computer science courses is from the domain of simple games. Games present some interesting design notions including *move*, *outcome*, *state*, and *turn*. If one focuses on the notion of a *turn* a new design is revealed that combines the familiar patterns of the Model-View-Controller architecture and Proxy when the game is played over a network.

1. INTRODUCTION

We define a *turn-based game* as one involving two or more players in which progress is indicated by discrete actions that each player takes, either sequentially or in parallel. Each of these actions has an *outcome* – an immediate impact on the state of the game. Outcomes are the agents of transition in the game state.

Along which dimensions should multiplayer games be modularized? Some answers that pop up frequently are the players, the rules, the game state, and the user presentation. In this paper we investigate turn taking as a key concept in the development of turn-based games. By treating moves as the *coin of the realm* a design is developed that aids in the distribution of the game components across a network while keeping the game's model fairly well separated from its view and controller [1].

2. DESIGN

For the remainder of this document we will discuss the specific case of two-player games. We believe that the principles apply equally well to games of more than two players, but that is a matter for further investigation.

2.1 Interface Development

We start with a simple interface `IPlayer` for classes that provide the connection between the game and one of the players.[1] The controller aspect of the interface is completely passive in that it responds only to requests from the model beneath it. For example, when the game is ready to find out a player's move, the game asks for it. As a secondary issue, this approach may require an atypical graphical user interface (GUI) implementation. Synchronized threads will interact with the state hidden by this interface. This idea will be explained later.

The current `IPlayer` interface, expressed in the Java language, is as follows:

```
public interface IPlayer< Move, Outcome > {
    void allow ();
    Move getMove ();
    void setOthersMove( Move move );
    void present();
    void outcome( Outcome outcome );
}
```

These five methods are called in a repeating cycle, but the order varies depending on the game. Here are the definitions of the methods:

`allow()`
: A new round of moves has begun. The player is now allowed to enter a new move. (It is this player's turn.)

`getMove()`
: Retrieve the move chosen by the player in this round. Block until the player has chosen a move.

`setOthersMove()`
: Learn of the move chosen by the other player.

`present()`
: The user interface may present the other player's move to this player.

`outcome()`
: Report the outcome of this round, e.g. who won or what points were gained.

2.2 Play Algorithms

One style of game playing is where each player is allowed one turn in a round and sees the other's move before he must choose his own. Tic Tac Toe works this way [2].

```
player[i].allow();
iMove = player[i].getMove();
player[j].setOthersMove( iMove );
player[j].present();

player[j].allow();
jMove = player[j].getMove();
player[i].setOthersMove( jMove );
player[i].present();

gameState = …; // compute the outcome
player[i].outcome( gameState );
player[j].outcome( gameState );
```

Figure 1: Sequential Turn Taking.

Another style would be where all players may take their turns roughly simultaneously, and may not see the other moves until all have chosen. As an example consider the game "Rock, Paper, Scissors" [3].

[1] The term `IPlayer` is used to refer to the programmatic interface for the classes that interact with the game players. When "player" refers to the actual players of the game, it will be expressed in normal font and lower case.

```
player[i].allow();
player[j].allow();

iMove = player[i].getMove();
player[j].setOthersMove( iMove );

jMove = player[j].getMove();
player[i].setOthersMove( jMove );

player[j].present();
player[i].present();

gameState = …; // compute the outcome
player[i].outcome( gameState );
player[j].outcome( gameState );
```

Figure 2: Parallel Turn Taking.

Other patterns are, of course possible.

2.3 The Referee

The other major component of this design is actually suggested by the above algorithms. Any such algorithm would be housed as a template method [4] within a class contained in the model component of the design. The class will here be called a *referee*. In a simple design, all the rules of the game can be contained in this one class. In a more complex game, a referee might only control turns, while another class could update the information associated with the current state of the game. No interface is therefore suggested for this class.

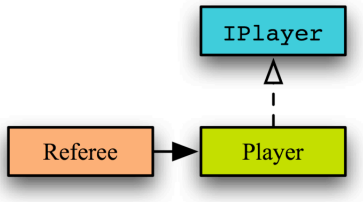

Figure 3: Class Diagram.

A simple game would likely have three objects created – two that implement the `IPlayer` interface and a Referee instance that makes calls on the `IPlayers`.

A simple game's object diagram would typically look like Figure 4 wherein we use the game of Tic Tac Toe as an example.

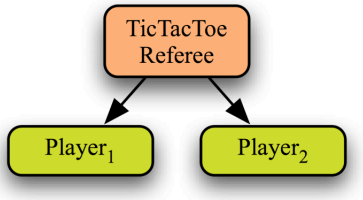

Figure 4: Object Diagram for Tic Tac Toe.

If the interface between the human player and the `IPlayer` object is provided by an event-driven GUI, then additional scaffolding is inserted in the form of a synchronizing *cell* (a monitored single-element queue) between the `IPlayer` object and the graphical elements. In this way the referee may still make calls on the `IPlayer` objects (*pull*) rather than the other way around (*push*).

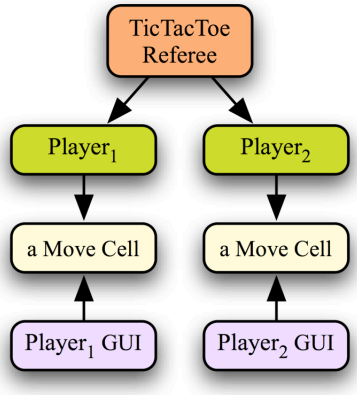

Figure 5: Object Diagram with GUI.

It might be worth noting to students that the view and controller in an MVC architecture are far more than the objects instantiated from a GUI library. They contain all the application-independent and application-aware components that deal with actor-system communication but have no knowledge of the rules (*business model*) of the application.

3. NETWORKING AND PROXIES

How does the design change when the (human) players are sitting at separate hosts in a network? Perhaps the most obvious answer is to put one `IPlayer` object on the second host and replace it with a proxy object [4] on the first host.

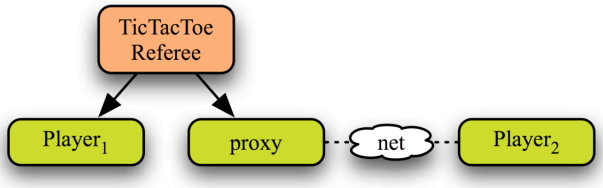

Figure 6: A Proxy for the Second Player.

Although workable, this seems rather unbalanced and it forces the two players to do drastically different things to start up the game. In addition, the students might need to know how to deploy a distributed server, because objects that must communicate to their application via proxy are often embedded in a server-style application.

Another design choice that is often made is to treat the referee as a game server and place it on a separate host. Players would run a player program that would connect to the service. All player moves would now in essence have to travel across the network twice.

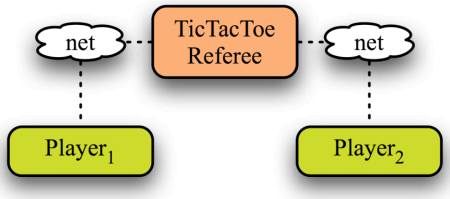

Figure 7: Referee as a Server.

If, however, instruction focuses on discernable design patterns, this is starting to be a significant problem. The `IPlayer` objects can no longer be labeled as pure controllers, nor the referee as a pure model; they are both also involved in network communication. Certainly one can argue that proper use of proxies can once again separate the concepts, but we fear this demands more "abstraction maturity" on the part of the students.

Our approach involved a discovery. If two `IPlayer` objects are connected as peers across a network where each object acts as a proxy for the other, the system can be designed as shown in Figure 8. We will call these dual local/proxy `IPlayer` objects "dual players".

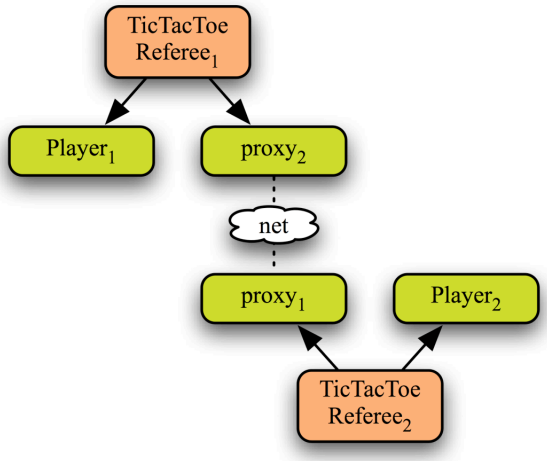

Figure 8: Replicated Referee and Dual Players.

There are now two referees running in sync. Each one thinks it is talking to two local players. When one referee talks to its dual player, its peer on the other side talks to its own referee using the opposite part of the `IPlayer` interface. Two possible timing scenarios are shown in Figures 9 and 10.

As an aside, an additional design element also reveals itself. Both

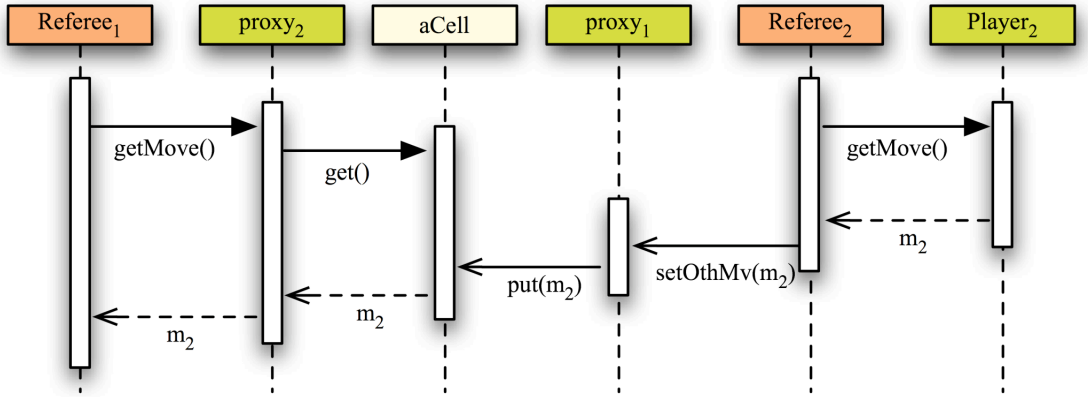

Figure 9: Sequence Diagram for Serial Turn Taking.

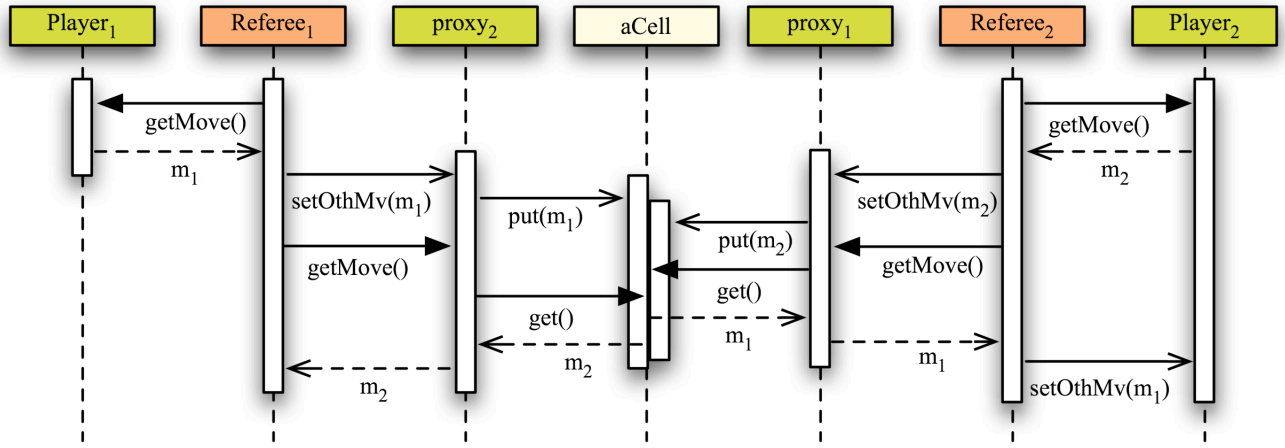

Figure 10: Sequence Diagram for Parallel Turn Taking.

dual players are connected to common cells that synchronize communication of moves. Figure 9 illustrates one turn in sequential turn taking and suggests that a single cell is sufficient for synchronization — it will be used for each direction in turn. Parallel turn taking, however, requires two cells because it cannot be predicted when the moves will arrive to be communicated to the other side.

There are pedagogical repercussions to this design. The cell can be realized in a concurrent or in a distributed environment, using a monitor, a simple multi-threaded message exchange service that uses strings or serialized objects à la C#/Java, a remote object invocation service, or even a web service if it includes a way to realize asynchronous calls [5].

4. DISCUSSION SUGGESTIONS

Our design has some consequences that can be used for class discussion.

One observes that the referee objects on the two different hosts are actually running the same code in approximate synchronization with each other. At the end of any round, one should be able to query either one and get the exact same game state. Under what conditions, if any, could the two sides get out of sync? What would the consequences be?

Like many designs, the methods in the `IPlayer` interface should be called only in certain orders. Can those orders be articulated? What could be done to enforce them?

These two dual player objects "linked at the hip" (through the network) do not represent the same player at the two ends of the link. Is this too confusing? Is there a potential for a designer to "hook up" the system improperly?

A fourth possible discussion point comes from presenting the overall architecture of the design to the students and having them experiment with different interface designs for `IPlayer`. The criterion would be that the interface must be appropriate and sufficient for both a completely local object (player communicates directly with the object through a UI) and for a dual player.

5. CONCLUSIONS

We often tell our students that object-oriented design is very straightforward; the objects we see in the requirements appear quite naturally in our design. However, this does not mean that the best solution is always the most obvious one.

In this case, the obvious solution for a distributed turn-taking game is to use the apparent symmetry of UI-Referee-UI and place each of these on a separate machine.[2] Although this works, if one adopts the standard practice that the component at the user end is a client, and clients *pull* from the server (Referee), the entire algorithmic design is inverted from the original one-machine design. Reuse doesn't happen.

If one pays attention to the wisdom of our ancestors, i.e. follows design patterns, it becomes clear that each Player must be represented on the other side of a network by a proxy. At that point a new symmetry falls into place. There can be referees on both sides, each one playing a local player against a proxy. Now the referee is exactly the same as the original local one; it contains a template method that is clueless as to how the `IPlayer` interface is implemented.

What follows then is that `IPlayer` must be carefully designed to accommodate both a local and a proxy player, and that the server must be multi-threaded so that deadlock is avoided. Thankfully, all that is needed for the latter requirement is a cell in each direction, which can be implemented in many ways, even as an asynchronous web service.

6. REFERENCES

[1] A. Goldberg, D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley 1983.

[2] Adit Software, *The History of Noughts and Crosses*, <http://www.adit.co.uk/html/noughts_and_crosses.html>, checked August 25, 2006.

[3] World RPS Society, <http://www.worldrps.com/>, checked August 24, 2006.

[4] E. Gamma et al., *Design Patterns*, Addison-Wesley 1995.

[5] M. Powell, *Asynchronous Web Service Calls over HTTP with the .NET Framework*, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnservice/html/service09032002.asp>, checked August 24, 2006.

[2] The first player's UI and the Referee could share a machine instead.