Runtime Evolution for Online Gaming: A Case Study using JBoss and Drools

Lei Zhu
School of Computing Science
Newcastle University
NE1 7RU
+44-(0)-191-222-6053
lei.zhu1@ncl.ac.uk

Graham Morgan
School of Computing Science
Newcastle University
NE1 7RU
+44-(0)-191-222-7983
graham.morgan@ncl.ac.uk

ABSTRACT

In this paper we describe a rule based approach to online game development. Our goal is to ease the evolution of an online game by allowing far reaching change in gaming scenarios after game deployment has occurred and during game play. This is achieved by making use of a rules based engine (Drools) within the JBoss platform. We use a simple gaming scenario to demonstrate how far reaching change is possible without the difficulty of altering program code when rules are separated out from other application level logic.

Categories and Subject Descriptors

C.2.4 [Distributed Systems]: *Client/server*

General Terms

Management, Experimentation, Human Factors, Languages

Keywords

Online gaming, middleware, rules engines

1. INTRODUCTION

There are a number of commercial solutions to online gaming within which players may participate in virtual worlds that are persistent in nature. Such games are commonly termed Massively Multiplayer Online Role-Playing Games (MMORPGs), which is usually shortened to MMOs within the industry. Vendors generate revenue from such gaming environments by regular financial subscriptions made by players and/or from the value of virtual world artifacts (e.g., virtual land sales, percentage take from the inter-player trading of virtual world artifacts, sale of additional vendor created virtual world storylines and artifacts).

Key to the commercial success of an MMO is the ability to attract and then retain players. The amount of financial revenue associated to an MMO business venture is directly related to the number of regularly participating players in a virtual world: more players relates to more revenue. The number of players is commonly used as the measure on which an MMO is judged. Attracting players to an MMO is not straightforward, and requires the typical advertising campaigns one may witness for the popularizing of many products. However, the mechanisms for retaining players and ensuring the longevity of a business venture must be achieved within the MMO itself.

Player interest must be maintained over prolonged periods of time, appropriately measured in years. Therefore, an MMO must continue to provide new and challenging scenarios to encourage player participation. This can be achieved by periodically introducing new content (e.g., artifacts, rules, stories, areas) and ensuring all content exhibits a degree of persistence to provide a heightened sense of continuing community. As such, the content of a virtual world an MMO supports will, over time, increase. Such an increase in content will require additional world maintenance and, if not achieved in a highly regulated manner, may raise the probability of world failure; either in part or completely (e.g., [1] [2]).

Current engineering practices dictate that the requirement for a highly evolving, persistent, virtual world must be weighed against the requirement for a correctly functioning, always available simulation. This is mainly due to the manner in which change is afforded at the coding level via manual updates. This approach has resulted in the management of change in an ad-hoc manner and severely limits the ability to introduce far reaching change while ensuring the correctness of a simulation.

One must realize that this problem of achieving a failure free approach to runtime maintenance and adaptability for distributed applications is not solely within the domain of MMOs (e.g, [3]). However, MMOs do provide a highly visual view of the problem. In addition, MMOs have timely requirements and player-to-player interactions that add to the difficulty of application maintenance. In this respect, MMOs provide an excellent case study for attempting techniques to ease the maintenance of real-time distributed application with persistent properties [4].

In this paper we consider the difficult problem of MMO maintenance and adaptability in the context of evolving game play. That is, rather than concentrate on the management of artifacts we wish to actually change the rules that govern gaming scenarios. This approach is quite unique in this area as work to this date has been preoccupied with the management of persistent data (e.g., cataloging, inventory, retrieval).

A reason for tackling the rules as opposed to the objects contained within a simulation is that by changing rules one can create quite diverse gaming environments. Another reason is that changing the rules is considered a challenging aspect of MMO maintenance, as rules are encoded throughout the implementation (possibly throughout the program code representing many different virtual world artifacts). Successful tackling of this problem will promote the runtime diversity and longevity of MMOs.

The next section identifies related work and provides a clear and concise description of the problem. Section 3 identifies our approach and provides justification for our approach. Section 4 presents the implantation representing our case study and is used to exemplify the potential of our approach. Section 5 draws conclusions from our work and identifies the possibilities for future work.

2. Background

In this section we clarify the context of the work. The implementation scenario is described and what we mean by game play evolution is presented. Academic work does not directly address this problem (in the context of online gaming); therefore we present approaches taken in industry to handle game play evolution.

2.1 Implementation Scenario

Persistent virtual world implementations are server based, allowing vendors to regulate the provision of ever evolving alternate realities to maintain player interest and, most importantly, restrict participation to subscribed players. Player consoles connect to a server that provides players with access to a virtual world. Typically, a player's console holds a sub-set of game state with players informing each other of their actions via the exchange of messages between consoles. Such communication is achieved via a server, allowing the regulation of player interaction and game state to be recorded and stored onto a persistent medium if required.

To satisfy the demand for processing resources, clusters of servers are employed to cumulatively maintain game state and manage player interactions. The additional processing resources required to support an increase in player numbers is satisfied via the addition of servers to a cluster. This approach to server cluster configuration will be familiar to any developer working with scalable service solutions found in almost all Internet applications; utilize a collection of geographically co-located nodes organized into a cluster that cumulatively support online services (e.g., search engines, e-commerce, enterprise information portals). Such nodes are standard computers in their own right, and may operate as service providers independently of each other. Such computers are general purpose and not necessarily tailored for high performance multi-processor solutions, making them a cost efficient approach to server side scalability.

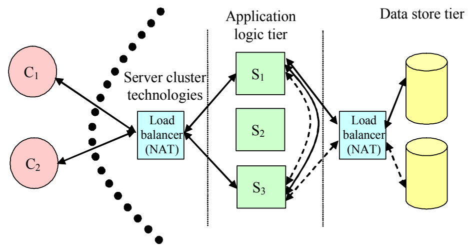

Figure 1 – Classic n-tier server side solution.

Figure 1 provides an overview of a typical server cluster solution for providing scalable online worlds. Although a simplified view, this will suffice for descriptive purposes. The load balancer ensures players are directed to an appropriate server that may satisfy their service requests (e.g., updating avatar appearance or location). The application logic is where user participation is enacted and overall governance of the virtual world occurs (e.g., avatars fighting, trading artifacts between users). Artifacts, including player's avatars, which populate the virtual world, together with their current state are stored in the data store tier and retrieved as and when required by the application tier. Updates made to persistent artifacts in the application logic tier must be registered in the data store tier to ensure continuity of the virtual world.

2.2 Game Play Evolution

Ignoring extensive philosophical arguments, it is sufficient to state for the purposes of this paper that we consider a gaming scenario to be a collection of rules, content and associated players. Content may be viewed as any number of objects to be manipulated by players while adhering to the rules governing a game. Rules are stated in such a manner as to be clearly understood within the context of a gaming environment. Our work here is based on the premise that by changing the rules of one gaming scenario, but not the content nor the number of players, we may create additional gaming scenarios.

The difficulty of evolving content may be highlighted by considering the following scenario, similar to the civilization type games from Sid Meier [4].

In a virtual world that already allows players to navigate ships between ports, we wish to evolve an economic market by introducing "trade" and "cargo". Once introduced, players will be able to trade between ports via ships carrying cargo. There is a requirement to modify the artifact ship to enable the carrying of cargo. The new concept of trade will require modification to the rules governing the virtual world itself. Ports will assume the role of trade hubs and must be enhanced to recognize their role in trading.

In our example it is not sufficient to just add content, but existing content (ships, ports) must also be altered to enhance them with the ability to participate in trade. This requires updates in the data-store tier (e.g., amount of cargo that a ship can carry) and updates in the application logic tier to enhance functionality (e.g., unload/load cargo). Furthermore, other artifacts not mentioned in our example must be designated as cargo. This in itself will require updates to other artifacts in the data-store tier (e.g., weight, size, and owner) and additions in the application logic tier (e.g., in transit, set owner, and change value). Finally, the concept of trade itself is quite fundamental and not easily captured within one single artifact, requiring recognition in the rules governing a virtual world (e.g., supply, wealth, and exchange).

One may envisage creating a game that will allow this progression of evolution to unfold in a natural seeming manner. In fact, the Civilization series of games base their popularity on just such a progression, with players directing the game through a finite choice of human discovery and endeavor. However, in the context of persistent virtual worlds populated by many players the problem intensifies and becomes a challenging research oriented issue:

1) Consistency of the virtual world must be maintained for all players (this is a shared experience)
2) The evolution of game play may not have been realized at game design/implementation time.

When game play scenarios change they must change in a consistent manner, that is, players participating must be presented with the same shared experience. Not to achieve this would be to engineer an unfair scenario. Unfortunately, unlike Sid Meier's Civilization, the world's we are concerned with are persistent and the manner with which game play may change may only be determined once the virtual world has gone live (not pre-computed and finite – too limiting of scope). Therefore, no forward planning for a particular type of game play scenario is feasible.

In our example we identified changes required to content and rules. Changing content, although difficult and still a challenging research problem may be achieved. However, changing the rules is a challenge that our example clearly shows to be almost impossible if such rules are not easily accessible, even though rule change may bring about the most significant game play evolution.

2.3 Industrial Approaches

Faced with the problem of content management, vendors are restricted to manual updates by their own developers or by players themselves. Vendors providing MMOs may have good reasons to manage content, for the purposes of coherent storylines and directing the overall look and feel of a gaming scenario. However, this is a burdensome task when millions of artifacts exist. Therefore, an alternative approach has arisen where players are encouraged to create such content, albeit at the expense of a vendor's ability to direct gaming scenarios.

When vendors manage content the use of client side updates coupled with additions at the server side is common (e.g., [5]). Updates to client software (given or sold to a player to afford access to a vendor's MMO) may be an additional revenue stream for a vendor. Such updates are achieved by the vendor releasing "expansion packs" (software updates) which the player must purchase to participate in new gaming scenarios. To ensure existing players may continue to participate without "expansion packs" the vendor isolates new scenarios from existing content. This is achieved by adding a new area to a virtual world. In reality, existing content is not evolved, but increased in the form of additional areas.

Second Life [6], by Linden Lab, allows player content creation with a financial revenue model based on real-estate and trading: the main type of revenue for Linden Lab relates to the purchase of land and paying of ground rent. An innovative aspect of Second Life is the ability players have for creating content. Such content may then be traded between players. No client side updates are required to access new content. A scripting language allows artifacts to be instilled with behavior, allowing players to provide their own virtual world scenarios. This approach provides Second Life players with the most powerful content creation tool available today for MMOs with players providing a wealth of content.

Existing approaches to vendor and player derived content evolution can't realize our trading example as existing content cannot be changed appropriately to accommodate new content. In Second Life, propagation of change from one artifact to another is limited and inhibited between artifacts belonging to different owners. Even using such an inhibitive approach Second Life has been plagued by problems (failure of simulation due to erroneous scripts [1]). The more controlled approach used in vendor driven content change is viewed as a safer option (but failures still happen [2]). However, this safety has come at the expense of limiting existing content updates to simple bug fixes and only allowing new content distinct from existing content. The thought that the actual rules governing a gaming scenario may be altered has not been considered (or if it has, has never achieved fruition). Emphasis in current approaches has been placed on content creation and minor modification. Fundamentally, all existing approaches severely limit content evolution in favor of safety and the programming burden is immense. Rules are left alone.

3. Experiment

In this section we describe our case study in which we attempt to demonstrate that rule change may be possible if achieved with the appropriate toolset. We attempt our work using well known software commonly used in scalable enterprise solutions over the Internet. This way, our approach fits the diagram in figure 1 and should be applicable for online game development [8].

3.1 Rule Engine

In recent years engineers of e-commerce solutions have begun to make use of rule based approaches in the construction of their applications. A number of server side middleware products now include rule based tools as part of their application development support. As business practices are well attuned to operating within particular parameters governed by rules, efforts to construct software tools and techniques to ease the development of business oriented applications by allowing rules to be clearly stated have preoccupied a number of researchers (e.g., [7]). By separating the business logic from other aspects of application implementation one may alter business rules without a requirement to manually update a number of code fragments within the application tier of the server side. In such applications, the rules become a clearly identifiable (and manipulative) aspect of an overall application. This has proved successful in the development process as rules that were not determined accurately at design time could be tailored (or even created) even after a system has gone live.

Initially, rule based software tools originate from work carried out in the artificial intelligence (AI) research community. Work carried out in Expert Systems may be considered rule based with such research eventually taking a number of directions, most prominently helping to create the *Business Rule Management Systems* that are the subject of this paper. There are a number of rule based systems available for programmers to make use of, but of most interest to MMO developers are those found in distributed systems middleware solutions (e.g., JavaEE, .NET). For this reason we chose the JBoss platform and Drools engine [8] (others are available, but this choice was made due to our familiarity with JBoss).

3.2 Approach

As a first step towards evaluating the appropriateness of utilizing a rules engine for use within gaming environments we consider a very simple scenario. A board game for two players that resembles the well known game of "noughts and crosses" (also known as "tic-tac-toe") is used as the demonstrator. This game was chosen as the rules may be varied any number of ways to create distinct gaming scenarios.

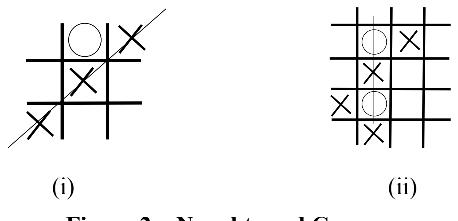

Figure 2 – Naughts and Crosses

In figure 2 there are two gaming scenarios presented. In 2.i there is the standard game where there are nine possible positions to place two types of game play pieces. Two players take it in turn to place their pieces on the board. A player wins if they achieve three of their own pieces in a row. In 2.ii the game is varied. Players still take turns, but now the board is bigger and the winning line achieved by spelling the word "OXO".

Possibilities for constructing varying different gaming scenarios are, in theoretical terms, infinite. However, assuming no change to player numbers then variance of grid size, winning word length, winning word pattern and player turns presents a finite set of changeable parameter. Our challenge is:

1) Separate rules from the implementation code
2) Implement using JBoss (Java) and Drools (rules)
3) Vary game by changing rules
4) Never alter Java code

After starting with the most simple scenario (figure 2.i), a player may instruct, by specifying rules arbitrarily, what gaming scenario they desire (even mid game). As this is a JavaEE implementation, the resultant game is easily prepared for online access.

3.3 Implementation

The game is implemented in three distinct parts as shown in figure 3. A client is built using the Java language. The client serves solely as an interface to the game and allows players to place their pieces on the game board. The client has no rules governing the game encoded within it, however, the client may realize what constitutes an appropriate placement of a piece on the board (i.e., piece must reside within an empty space). The initialization of the client (determining board size) is achieved with a request to the server. Once the game is being played the client can take requests from players. The client will not allow players to go out of turn.

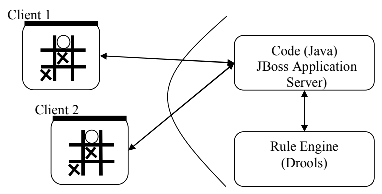

Figure 3 – Overview of implementation

Once a turn has occurred the client informs the server where a piece has been placed. A model representing the gaming arena is maintained by the server with information received from clients used to update this model. The model reflects what the players see, but has the ability to evaluate each move using the rule engine. The server communicates all moves to the rule engine. Once the rule engine has evaluated a move and indicated an outcome, this outcome is used to update the model held by the server. Once the model has been updated the server transmits the updated state to the clients. A number of outcomes are possible from a client move: *invalid* – the move is deleted and state rolls back; *progression* – the move progresses the game state; *winning* – the move wins the game; *finish* – no more moves possible.

3.4 Programming

With any technology there is a learning curve. Although the authors are proficient in programming languages and middleware technologies (e.g., JavaEE, .NET), this was the first time a rules engine had been used. However, the rules engine proved to be reasonably intuitive to use once syntax and semantics had been mastered. Therefore, we assume any programmer should be able to develop simple rules with minimal effort after a number of hours study and utilize the advanced features of the rules engine within a reasonable amount of time. As of writing this paper, we do not profess to be experts in creating rules using Drools; others may see shortcomings in our descriptions. However, we are able to use Drools sufficiently well for our purposes.

For comparison, we now consider a rules based approach using the rule engine and one where the rules are encoded within Java. We consider the programming task of identifying if the winning scenario of four pieces of the same type occurs in a line (column, row or diagonal). We assume the grid is 4x4. The process is simply a search for a particular pattern. There are more efficient ways for searching, but we simply identified a brute force approach for clarity.

```
public boolean isGameWon (int player) {
// check all columns whether 4 same
symbols exist
for (int x = 0; x < 4; x++)
   {
     int count = 0;
    for (int y = 0; y < 4; y++)
       {
          if (data[x][y] == player)
             count++;
          }
    if (count ==4) return true;
   }
   // repeat for row and diagonal
```

Figure 4 – Expected (part) code for implementing rule

Figure 4 shows part of the Java code used to identify a winning row of 4 pieces the same. Two further pieces of code were written to determine the identification in the other directions (row and diagonal). In a typical game engine such code would be optimized and such optimization may become tightly coupled to other aspects of the game implementation. For example, one would not search all the game board each play but concentrate only on those areas that could be affected by the last move being assessed and consider only the player who has just completed their move.

```
rule "Win with the catercorner2"
   when
       $mainStatus :ArrayofSymbolStatus(
            playerholder != -1,
            $x : x,
            $y : y,
            $playerholder : playerholder)
     $alarms : ArrayList
     (
        size >=
(RuleCommon.SUC_NUM_OF_ROWS_COLS - 1)
     )
     from collect(
       ArrayofSymbolStatus(
             playerholder ==
$playerholder,
            x >= ($x -
RuleCommon.SUC_NUM_OF_ROWS_COLS + 1),
          x < $x,
          $x1 : x,
          y == ($y - $x1 + $x)))
    then
     ruleCommon.wonGame();

end
```

Figure 5 – Implementation of rule in rule engine

Figure 5 shows the equivalent rule as shown in figure 4, but this time represented in the rule engine format. The first obvious note to take is that the complete rule is not as substantial as the whole

Java equivalent (only a third of the code is shown in figure 4). In addition, the rule captures the notion of identifying a winning scenario in a manner independent from the Java approach.

Optimization of the rule is not possible as one would optimize the Java implementation. However, this is not a drawback. The rule engine abstracts the concern of optimization away from the developer and uses well known optimization techniques to search the memory space of the problem. This has been an area of research for a number of years; it is wise to make use of it in such an engine.

So far we have considered a winning rule that may be tailored, irrelevant which language it is written in: with a little foresight we can structure the java code so all it takes is parameter values, and not a change to the code, to test for winning lines of arbitrary length. A more problematic scenario is when the winning line pattern changes. Consider the rule described in figure 6.

```
rule "Win with the same column"
     when
          $mainStatus : ArrayofSymbolStatus
(playerholder != -1, $x : x, $y : y, $playerholder :
playerholder)
          $alarms1 : ArrayList($size1 : size)
               from collect(ArrayofSymbolStatus
(playerholder != -1,
               playerholder == $playerholder,
               x == $x,
               y > $y,
               eval((y - $y) % 2 == 0),
               y <= ($y +
RuleCommon.SUC_NUM_OF_ROWS_COLS - 1)))
          $alarms2 : ArrayList($size2 : size)
               from collect(ArrayofSymbolStatus
(playerholder != -1,
               playerholder != $playerholder,
               x == $x,
               y > $y,
               eval((y - $y) % 2 == 1),
               y <= ($y +
RuleCommon.SUC_NUM_OF_ROWS_COLS - 1)))
          eval(($size1 + $size2) ==
(RuleCommon.SUC_NUM_OF_ROWS_COLS - 1))
     then
               ruleCommon.wonGame();
end
```

Figure 6 – Variable pattern winning rule

In figure 6 a rule has been designed to test for winning lines that spell alternate pieces. A substantial change to the java code shown in figure 4 would be required for this scenario. One may assume that a programmer, through good programming practices, could have foreseen the need to identify variable sized grids and winning line lengths. However, the prospect of the winning pattern changing formats may not have occurred to them.

The rule shown in figure 6 was created and deployed after the game had been executed on the platform. There was no need to alter any programming code of the server and the client and the game changed without problems during runtime.

4. Conclusions

The work presented here, although in its earliest stages, demonstrates that rule engines can ease game evolution. This is achieved by allowing developers to safely upgrade, delete or create rules governing a simulation during runtime. There is no need to alter actual program code. The separation of rules from other aspects of implementation has proved beneficial in this respect. To achieve a similar evolution in game play program code would have to be changed during runtime, something that is considered difficult to achieve in a safe manner. In essence, modern rule based approaches used extensively in the e-commerce engineering industry warrant further investigation by MMO developers.

An interesting aspect of the work carried out is the realization that optimization of rule execution is now removed from the ad-hoc approaches of game developers to the rule engines themselves. One may argue that optimization achieved by a programmer specifically with the problem in hand may return more optimum solutions. However, this assumes that the game play scenario has actually been thought of during the initial construction of program code. Optimization while updating existing code during runtime is difficult for any programmer to achieve safely and correctly.

A difficulty encountered during development was the identification of rules. Initial implementations resulted in applications where the rules were partly in the rule engine and partly in the client and/or server. This was only discovered when game play was changed, and a realization that what was actually part of the Java code was inhibiting change as it was rule dependent. An interesting avenue for future work would be to provide some tool to aid in determining how rule dependent a piece of code is.

Changing the underlying rules governing a virtual world is considered the most challenging aspect of game evolution in MMOs (as opposed to simply adding content). This is highlighted by our example in 2.2. However, when structured appropriately this may appear to be a much more straightforward (or at least attainable) goal than once thought.

5. REFERENCES

[1] Linden Lab, “Security and Second Life”, *http://blog.secondlife.com/2006/10/09/security-and-second-life/,* viewed August 2008

[2] CNet, “World of Warcraft' battles server problems”, *http://news.com.com/World+of+Warcraft+battles+server+problems/2100-1043_3-6063990.html* as viewed August 2008

[3] Milazzo, M., Pappalardo, G., Tramontana, E., and Ursino, G. 2005. Handling run-time updates in distributed applications. *In Proceedings of the 2005 ACM Symposium on Applied Computing*, Santa Fe, New Mexico, March 13 - 17, 2005, New York, NY, 1375-1380

[4] Fraxis Games, “Civilization”, http*://www.civilization.com/* as viewed September 2008

[5] Blizzard Entertainment, “WoW: Burning Crusades”, *http://www.worldofwarcraft.com/burningcrusade/*, viewed August 2008

[6] Linden Lab, “Second Life”, *http://secondlife.com/*, viewed August 2008

[7] Oracle, “Oracle Business Rules: Technical Overview”, available from Oracle Website: *http://www.oracle.com*

[8] Lu, F., Parkin, S., and Morgan, G. 2006, “Load balancing for massively multiplayer online games”, *In Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support For Games*, Singapore, October 2006, NetGames '06. ACM, New York, NY

[9] Mark Proctor et al. Drools Documentation. JBoss. *http://labs.jboss.com/drools/documentation.html*