Developer's Guide

March 2, 2006

Contents

The Java Ranking Service

The Java Ranking Service is an implementation of Mark Glickman's[1] glicko-2 algorithm for rating player performance, with extensions for multiplayer and team games. Using these ratings, the ranking service provides leaderboard and match-making functionality.

The purpose of this document is to instruct developers of game servers on how to incorporate the Java Ranking Service into their game. Typical use cases are:

- Creating the ranking service
- Posting results and computing ratings
- Getting leaderboard information
- Getting matches (individual players and games)

Each of these is described in detail in later sections of this document.

[1] Mark Glickman is Associate Professor in the Department of Health Services at the Boston University School of Public Health. His work on the glicko rating system is documented at http://math.bu.edu/people/mg/glicko/index.html

The JRS classes

The classes that make up the ranking service are contained in the `jrs` package.

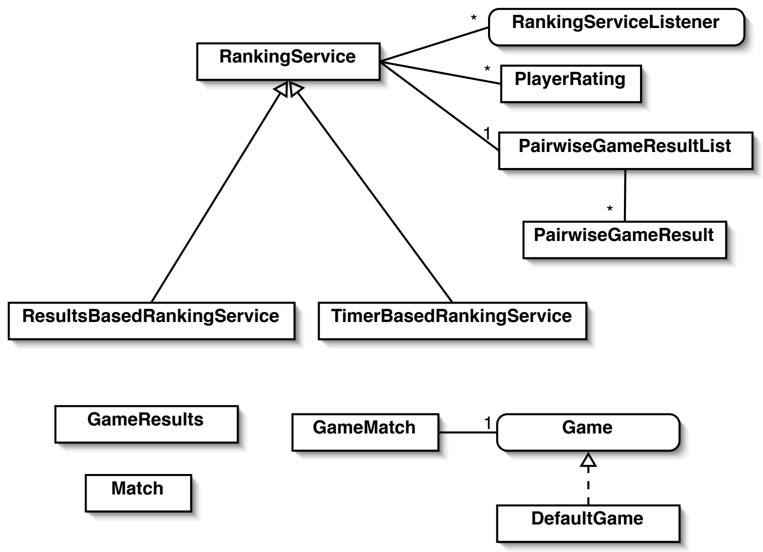


Figure 1 - The JRS classes.

At the core of the implementation is the `RankingService` class, which is responsible for calculating the ratings of the players registered with the system, and for providing methods for retrieving leaderboard and match-making information based on these ratings. The two subclasses of the `RankingService` class provide alternate strategies for determining when to update the player ratings.

The ranking service does not compute true ratings, but rather gives an estimate of the player's rating in the form of a confidence interval. The interval is specified by a *mean* rating, and a rating *deviation* which expresses the uncertainty in the rating. Additionally, ratings are assigned a volatility, which reflects how consistently the rating has changed over time. All this information is encapsulated in the `PlayerRating` class.

The ranking service uses the results of games played to compute the player ratings. Internally, the `RankingService` class uses instances of `PairwiseGameResult` to represent the results of a game, as the glicko-2 algorithm only deals with head-to-head games. The `PairwiseGameResultsList` class provides a way to group these pair-wise results into multiplayer game results, and provides methods for *down-weighting* these results so that multiplayer games do not have a disproportionately greater impact on player ratings than head-to-head games.

The results of games are passed to the ranking service as instances of the `GameResults` class. This class can be used to represent both multiplayer and team games.

Match-making information retrieved from the ranking service is encapsulated in instances of `Match` and `GameMatch`. The `Match` class represents how a player matches up against another player, while `GameMatch` represents how a player matches up against all the players in a game.

Ratings are computed at the end of rating periods. Instances of `RankingServiceListener` may be registered with the ranking service to be notified when these rating periods begin and end.

Creating the ranking service

During initialization, the game server should create an instance of the `RankingService` class.

```
RankingService rankingService = new RankingService();
```

There are two additional variants of the ranking service, `ResultsBasedRankingService` and `TimerBasedRankingService`, that differ in how they determine when to calculate player ratings. The `ResultsBasedRankingService`, like the standard `RankingService` class, is instantiated with a no-arg constructor and requires no additional initialization. The `TimerBasedRankingService`, however, requires that an update period be passed in its constructor, and that its `startUpdating()` method be called. For example, the server for the RPS sample game creates and initializes a `TimerBasedRankingService` using the following:

```
rankingService = new TimerBasedRankingService(60);
rankingService.startUpdating();
```

The next section, *Posting results and computing player ratings*, provides a more detailed discussion of how these three variants differ.

The ranking service does not persist any ratings it computes. Rather, it is left to the game server to store the list of players and their ratings, if it requires this. If previous ratings had been persisted, as is done in the RPS server, the `RankingService` object can be initialized with these ratings after it has been instantiated:

```
BufferedReader br = new BufferedReader(new FileReader(dataStorePath));
String line;
while ((line = br.readLine()) != null) {
    String[] fields = line.split("\\|");
    String playerId = fields[0];
    String password = fields[1];
    double rating = Double.parseDouble(fields[2]);
    double ratingDeviation = Double.parseDouble(fields[3]);
    double ratingVolatility = Double.parseDouble(fields[4]);
    userCredentials.put(playerId, password);
    PlayerRating playerRating =
        new PlayerRating(playerId, rating, ratingDeviation,
                         ratingVolatility);
    rankingService.registerPlayer(playerId, playerRating);
}
```

As new players are registered with the game server, they should also be registered with the ranking service

```
rankingService.registerPlayer(playerId);
```

This assigns the player the default rating values, as defined by the JRS system properties.

The JRS System properties

The following table lists properties that the ranking service uses to tune the glicko-2 algorithm. Typically, these are specified in the game server's initialization code, prior to instantiating the `RankingService`. The Analyzer tool (see Appendix A) allows you to experiment with different values for these properties and see their impact on the ratings calculations.

Property	**Default Value**	**Description**
`jrs.defaultRating`	1500	The rating assigned to new players.
`jrs.defaultRatingDeviation`	350	The rating deviation assigned to new players.
`jrs.defaultRatingVolatility`	0.06	The rating volatility assigned to new players.
`jrs.drawThreshhold`	0	If the difference between two players' scores is less than or equal to this amount, the game will be considered a draw.

Property	Default Value	Description
`jrs.glicko2SystemConstant`	1.0	The glicko-2 system constant that constrains the change in volatility over time. Reasonable values range from 0.3 to 1.2
`jrs.downweightResults`	true	If true, the ratings computed by the glicko-2 algorithm will be *down-weighted* in multiplayer and team games to have the same impact on player ratings as head-to-head games.
`jrs.performanceVarianceAroundPlayerSkill`	50	Used in calculating the probability of a draw during match-making. This value can be determined empirically using the *Matching Calculator* in the Analyzer tool.

Table 1 - System properties of the JRS.

Posting results and computing player ratings

As games are completed, the game server must pass the results to the Java Ranking Service so that it may update the ratings of the player's involved. This is done by calling the `postResults()` method and passing it an instance of `GameResults`.

```
GameResults gameResults = new GameResults();
if (isTeamGame()) {
    Iterator teamNames = teamScores.keySet().iterator();
    while (teamNames.hasNext()) {
        String teamName = (String)teamNames.next();
        Iterator playerIds =
            getTeam(teamName).getPlayerIds().iterator();
        while (playerIds.hasNext()) {
            String playerId = (String)playerIds.next();
            double playerScore =
                ((Integer)playerScores.get(playerId)).doubleValue();
            gameResults.addPlayerResults(teamName, playerId,
                                         playerScore);
        }
        double teamScore =
            ((Integer)teamScores.get(teamName)).doubleValue();
        gameResults.setTeamResults(teamName, teamScore);
    }
}
else {
    Iterator playerIds = playerScores.keySet().iterator();
    while (playerIds.hasNext()) {
        String playerId = (String)playerIds.next();
        double playerScore =
            ((Integer)playerScores.get(playerId)).doubleValue();
        gameResults.addPlayerResults(playerId, playerScore);
    }
}
rankingService.postResults(gameResults);
```

The `GameResults` class encapsulates the score of each player in the game, and, in the case of a team game, the score of each team. Typically, an instance is created, then one of its `addPlayerResults()` methods is called for each player in the game. Which method to use depends on whether or not teams are involved. Use

```
void GameResults.addPlayerResults(Object playerId, double score)
```

to record a player's results in a mutliplayer game, and use

```
void GameResults.addPlayerResults(Object teamId, Object playerId,
                                  double score)
```

to record a player's results in a team game.

Note that, when computing player ratings, only the relative score matters. That is, if player A has a score of 1 and player B has a score of 0, player A is ranked simply as having defeated player B. Had player A scored 10, the ranking would be the same.

Also, note that individual scores in team games are not considered when computing player ratings. Rather, only the team result is used. This means that if player A scores 10 points and player B scores 5 points, but player A's team only scores 12 points where player B's team scores 20, then B is ranked as having defeated player A. By default, the team result is set to be the sum of the individual scores of each of the team's members.

However, this can be overridden with the `setTeamResult()` method after all individual scores have be recorded to the `GameResults` object.

As described in the previous section, there are variants of the ranking service that use different strategies to determine when to calculate the player ratings. According to the documentation for the glicko-2 algorithm, it is recommended that ratings be computed after each player has played 15 games, on average. The game server may try to determine this on its own. If so, it should use an instance of `RankingService`, and call its `endPeriod()` method after an average of 15 games have been played by each player in the system. Alternatively, if an instance of `ResultsBasedRankingService` is used, it will track the average number of games played based on the results passed to its `postResults()` method.

One of the features of the glicko-2 algorithm is that it increments a rating's uncertainty (ie. rating deviation) as time passes. However, it can only make this adjustment when new ratings are computed, which occurs at the end of a rating period. With `RankingService` and `ResultsBasedRankingService`, the rating period can continue on indefinitely if no games are played. While, in practice, this is not likely to occur, a third variant is provided to resolve this issue. The `TimerBasedRankingService` ends the rating period after a specified amount of time, whether an average of 15 games has been played or not. With this implementation, the game server administrator must tune the update period such that, on average, each player plays 15 games during each rating period.

Getting leaderboard information

The leaderboard is simply a list of each player's rating, ordered from highest to lowest. To obtain this list, first get the list of players registered with the ranking service, then retrieve each player's `PlayerRating` object and add it to a list. The `PlayerRating` class implements `Comparable`, so if a sorted collection such as `TreeSet` is used, the player ratings will automatically be sorted from highest to lowest.

```
TreeSet rpsPlayerRatings = new TreeSet();
Iterator playerIds = rankingService.getPlayers().iterator();
while (playerIds.hasNext()) {
    String playerId = (String)playerIds.next();
    PlayerRating jrsPlayerRating =
        rankingService.getPlayerRating(playerId);
    RPSPlayerRating rpsPlayerRating =
        new RPSPlayerRating(playerId,
                            jrsPlayerRating.getRating(),
                            jrsPlayerRating.getRatingDeviation(),
                            jrsPlayerRating.getRatingVolatility());
    rpsPlayerRatings.add(rpsPlayerRating);
}
```

Note that the code sample above comes from the RPS sample game, and that `RPSPlayerRating` is just a subclass of `PlayerRating` that adds additional methods that are needed to serialize the objects for sending to the RPS client. Since `RPSPlayerRating` is a subclass of `PlayerRating`, it is also `Comparable`, so adding it to a sorted set has the desired effect.

Getting matches (individual players and games)

The ranking service can be used to find players or games that are a good match for a specific player. A match is considered good if there is a high probability of a draw between the players involved.

Matching against individual players

A list of players that match up well against a given player is useful for arranging head-to-head games. There are four methods for getting a list of individual matches:

```
Set<Match> RankingService.getMatches(Object playerId)
Set<Match> RankingService.getMatches(Object playerId, int numMatches)
Set<Match> RankingService.getMatches(Object playerId, Set playerList)
Set<Match> RankingService.getMatches(Object playerId, int numMatches,
                                      Set playerList)
```

Note: The JRS is JDK 1.4 compatible, so it does not use generics, as shown here. These are included for clarity in the documentation only.

The first method returns a set of `Match` objects representing the best match-ups for the specified player from the list of all registered players. The second method is similar, except that it allows you to limit the number of matches returned. Note that these are the top matches.

The third and fourth methods are similar to the first and second, except that they allow you to specify the list of players to consider, rather than drawing from the pool of all registered players.

The `Match` objects that are returned provide the unique ID of the opponent, his rating and rating deviation, and the probability of a draw between the player and the opponent. The `Match` class implements `Comparable`, so if `Match` objects are added to a sorted collection, such as a `TreeSet`, they will be ordered from best to worst match.

Matching against games

Finding games that match-up well against a specified player is useful when listing games a player may join. The `RankingService` class defines two methods for obtaining a list of games that match a specified player:

```
LinkedHashSet<Game> RankingService.orderbyBestMatch(Object playerId,
                                                    Set<Game> games)
LinkedHashSet<Game> RankingService.orderbyBestMatch(Object playerId,
                                                    Map playerLists)
```

> *Note: The JRS is JDK 1.4 compatible, so it does not use generics, as shown here. These are included for clarity in the documentation only.*

Both methods return a set of `Game` objects ordered from best to worst match against the specified player. However, they differ in their specification of the list of games to consider for matching. In the first method, a set of `Game` objects is passed in. This method should be used if the class the game server uses to represent a game can implement the `jrs.Game` interface, which is defined as:

```
public interface Game {

    /** Get the ID of the game. This should uniquely identify the
      * game to an instance of the ranking service.
      */
    public Object getId();

    /** The ids of the players participating in the game.
      *
      * @return
      *     A Set of Objects representing the ids of the partipants.
      */
    public Set getParticipantIds();
}
```

If the game server does not represent games as a single class, or if implementing the `Game` interface is not possible for some other reason, then the second method should be used. In this case, the game server must create a `Map` of `Objects` uniquely identifying a game to a `Set` of `Objects` uniquely identifying the players in that game. For example:

```
HashMap playerLists = new HashMap();
Iterator games = getGames().iterator();
while (games.hasNext()) {
    MyGame game = (MyGame)games.next();
    HashSet playerList = new HashSet();
    Iterator players = game.getPlayers().iterator();
    while (players.hasNext()) {
        MyPlayer player = (MyPlayer)players.next();
        playerList.add(player.getId());
    }
    playerLists.put(game.getId(), playerList);
}
Set gamesToJoin =
    rankingService.orderByBestMatch(playerId, playerLists);
```

When this version of the `orderByBestMatch()` method is used, the set of objects returned are instances of `DefaultGame`, which is just a minimal implementation of the `Game` interface.

Like the `Match` objects returned by the individual match-making methods, a `GameMatch` object that provides details of the match-up can be obtained from the ranking service using the `getGameMatchDetails()` method:

```
GameMatch RankingService.getGameMatchDetails(Object playerId,
                                             Game game)
```

The `GameMatch` class implements `Comparable`, so instances of it may be inserted into a sorted collection to order them from best to worst match.

The JRS Package

The JRS package is a zipped archive of a specific release of the JRS. When unzipped, it has the directory structure listed in the table below.

`docs/`	Documentation for the JRS and associated tools and libraries. This includes JavaDoc for the JRS classes and the Developer Guide..
`exe/`	Scripts for launching the tools and sample applications.
`lib/`	The JRS jar files, and any third-party jar files required by the JRS.
`rps/`	The workspace for the sample game.
`rps/lib/`	Libraries used by RPS.
`rps/src/java/`	Java source files and other files that are often packaged with the generated class files (eg. .gif, .properties, and package.html files)
`rps/src/web/`	Files that would appear in a web archive, such as html and jsp pages.
`rps/src/web/WEB-INF/`	Deployment descriptors and other files that would appear in a web archive's WEB-INF directory.
`rps/build.xml`	The build file for the rps workspace.
`rps/build.properties`	Defines properties used during the build and deployment of the rps web application.

Table 2 - Directory structure of the jrs package.

To use the JRS in your game server, add the `jrs.jar` file from the `lib` directory into your classpath. Documentation for the classes in this jar file can be found in the `docs/api` directory.

The JRS package also includes an Analyzer tool which can be used to test the behaviour of the JRS in various scenarios. A script to launch this tool can be found in the `exe` directory. See Appendix A for details about how to use the tool.

Lastly, the JRS package includes the source code for a sample game, which you may review to see how the JRS was intended to be incorporated into a game server. The next section describes how to build and deploy the sample game. Appendix B describes the design of the game.

Building and deploying the sample game

The `rps` directory in the JRS package contains a workspace for the sample game. Within this directory, there is a `src` directory containing the java source files and other files needed to distribute the game as a web application.

The build.xml file is an ant build file that defines targets for building and deploying the web application. The two most useful targets are `deploy-rps` and `run-rps`.

The `deploy-rps` target rebuilds the `rps.war` file, if necessary, and deploys it to the web container specified in the `build.properties` file. To specify a web container, uncomment the appropriate block of properties in the `build.properties` file and set them to reflect your environment.

For example, to deploy to the Sun Java Systems Application Server, your `build.properties` file should define properties similar to the following:

```
as.home=/opt/as8
as.admin.user=admin
as.admin.passwordfile=/opt/as8/config/password_file.txt
as.admin.host=localhost
as.admin.port=4848
```

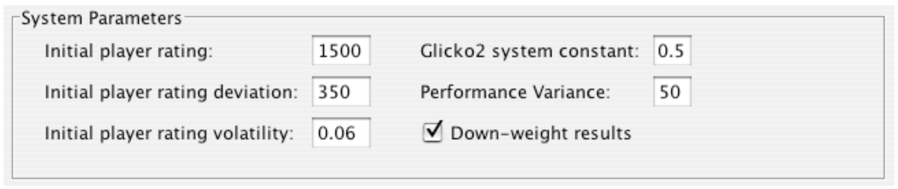
If no web-container properties are specified in the `build.properties` file, the `deploy-rps` target will still build the `rps.war` file, but it will only print a message instructing you to manually deploy the web archive.

The `run-rps` target invokes the `deploy-rps` target. Then, if the `browserCommand` property is specified in the `build.properties` file, a new browser window is opened and the client applet is loaded. If this property is not set, the build output will list the URL to use to launch the client.

Appendix A: The Analyzer

The Analyzer tool is intended to be used to test the behaviour of the glicko-2 algorithm and to tune the system properties that affect ratings calculations. It does this by simulating rating calculations for either randomly generated game results, or for explicitly specified, custom game results.

In either scenario, you may specify the system properties that affect how the glicko-2 algorithm calculates ratings using the fields in the top panel of the Analyzer tool's main window.



System Parameters

Initial player rating:	1500	Glicko2 system constant:	0.5
Initial player rating deviation:	350	Performance Variance:	50
Initial player rating volatility:	0.06	✓ Down-weight results	

Figure 2 - The system parameters.

There are three fields for specifying the initial values to assign to a player's rating variables when the player is first registered with the system. The values shown are those recommended in the glicko-2 documentation, however you can adjust these to see what affect they have on rating computation.

The Glicko2 system constant constrains the change in rating volatility over time. It corresponds to the `jrs.glicko2SystemConstant` property. Typically, values for this field range from 0.3 (small change) to 1.2 (large change).

The performance variance field is used in calculating the probability of a draw during match-making. This value can be determined empirically using the Matching Calculator, shown in figure 3. To use the matching calculator, open its window by selecting Matching Calculator... from the File menu. In the Beta field, enter a value for the performance variance, then specify a rating and rating deviation for each player. When the Calculate button is clicked, the probability that the two players would draw is computed. The goal is to determine a performance variance (or Beta) that yields reasonable probabilities for a range of different ratings match-ups.

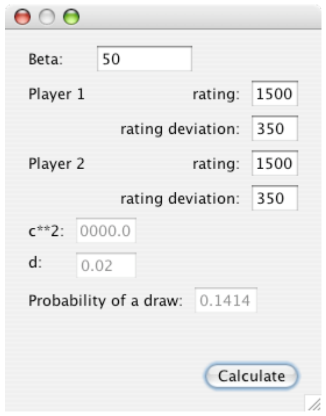

Beta: 50
Player 1 rating: 1500
rating deviation: 350
Player 2 rating: 1500
rating deviation: 350
c**2: 0000.0
d: 0.02
Probability of a draw: 0.1414

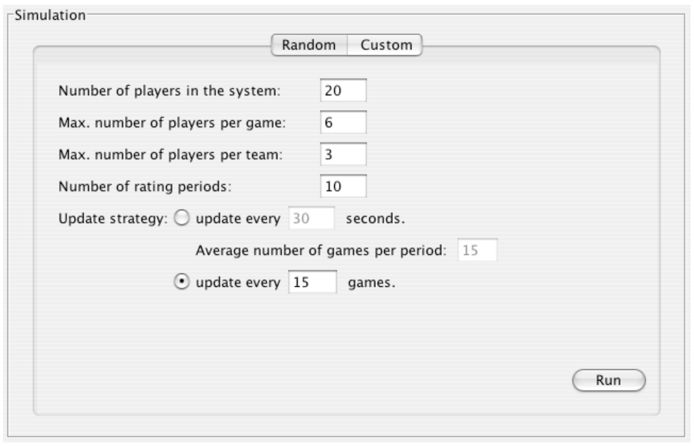
Figure 3 - The matching calculator

For example, if player 1 had a rating of 1500 and a rating deviation of 50, and player 2 had a rating of 1300 and a rating deviation of 50, then the probability of a draw should be quite low, so the choice of Beta should reflect that. However, if player 1 had the same rating and rating deviation as before, but player 2 had a rating of 1500 and a rating deviation of 50 as well, then the probability of a draw should be quite high. Again, the choice of Beta should reflect that.

Finally, the down-weight results checkbox enables the down-weighting of the impact results from multiplayer and team games have on the ratings calculations. This is an attempt to accommodate the fact that the glicko-2 algorithm was designed for head-to-head games only, and the extensions for multiplayer and team games tend to give more weight to games with more than two players. For example, if a player with a rating of 1500 were to defeat a player with a rating of 1600, that player may increase his rating to 1520. However, if that player were to defeat 3 other players in one game, each rated at 1600, his rating may increase to 1560. In some types of games, like death-matches, this can be misleading, because some of the other players may have had a hand in lowering the scores of the other players. However, in a multiplayer car racing game, finishing first may truly indicate that that player is better than all the others in the game. Given that there are many types of games, the down-weighting of multiplayer results been made optional. By default, it is enabled, but it can be disabled using this checkbox, or in the ranking service, by setting the `jrs.downweightResults` property to `false`.

Random Simulations

A random simulation is useful for seeing the effects of the system variables over several rating periods because you do not have to enter game results for every game in every period.



Number of players in the system: 20
Max. number of players per game: 6
Max. number of players per team: 3
Number of rating periods: 10
Update strategy: ○ update every 30 seconds.
Average number of games per period: 15
◉ update every 15 games.

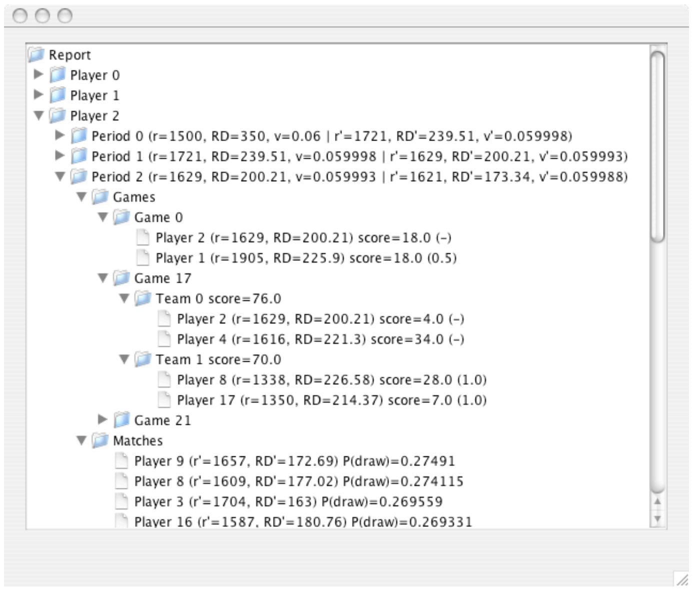
Figure 4 - Setting up a random report.

Using the Random Simulation tab in the main window, enter the number of players to be registered with the ranking system. Each player will be assigned the same rating and rating deviation when they are registered. These values are taken from the initial values specifed in the System Parameters panel described earlier.

In the random simulation, the games are defined randomly as well. The number of players in each game will be between 2 and the value specified for the maximum number of players per game. These players will also be randomly selected from the pool of registered players. If the maximum number of players per team is set to 0, then no team games will be generated. However, if this field is not 0, then teams of size 1 to the maximum will be created. Note that a game with teams of size 1 are the same as a multiplayer game, and are thus represented as such in the simulation report. Also, note that if the size of the teams is not an even multiple of the number of players in the game, then no teams will be formed. Rather, a multiplayer game will be simulated.

The update strategy radio buttons allow you to specify when rating periods will end. This is analogous to the different strategies employed by the `ResultsBasedRankingService` and `TimerBasedRankingService` classes, as described in the first section of this document.

Once the simulation parameters are setup, click the Run button to run the simulation and generate the simulation report. A typical report is shown in the figure below.



```
Report
▶ Player 0
▶ Player 1
▼ Player 2
    ▶ Period 0 (r=1500, RD=350, v=0.06 | r'=1721, RD'=239.51, v'=0.059998)
    ▶ Period 1 (r=1721, RD=239.51, v=0.059998 | r'=1629, RD'=200.21, v'=0.059993)
    ▼ Period 2 (r=1629, RD=200.21, v=0.059993 | r'=1621, RD'=173.34, v'=0.059988)
        ▼ Games
            ▼ Game 0
                  Player 2 (r=1629, RD=200.21) score=18.0 (-)
                  Player 1 (r=1905, RD=225.9) score=18.0 (0.5)
            ▼ Game 17
                ▼ Team 0 score=76.0
                      Player 2 (r=1629, RD=200.21) score=4.0 (-)
                      Player 4 (r=1616, RD=221.3) score=34.0 (-)
                ▼ Team 1 score=70.0
                      Player 8 (r=1338, RD=226.58) score=28.0 (1.0)
                      Player 17 (r=1350, RD=214.37) score=7.0 (1.0)
            ▶ Game 21
        ▼ Matches
              Player 9 (r'=1657, RD'=172.69) P(draw)=0.27491
              Player 8 (r'=1609, RD'=177.02) P(draw)=0.274115
              Player 3 (r'=1704, RD'=163) P(draw)=0.269559
              Player 16 (r'=1587, RD'=180.76) P(draw)=0.269331
```

Figure 5 - Report from a random simulation.

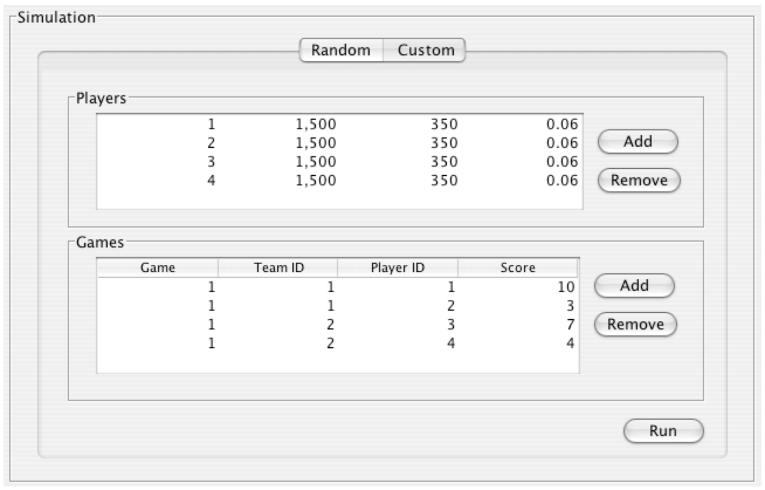
The simulation report is presented as a tree structure. There is a node for each player that was registered with the ranking service. Under each player node is a period node that lists the player's rating, rating deviation and rating volatility both at the beginning (r, RD, v) and ending (r', RD', v') of the rating period.

Within each period node is a games node, which contains a subnode for each game the player participated in, and a matches node, which contains a list of subnodes representing players that matched up well against the player by the end of the period. This list is ordered from best match to worst (up to 10 matches).

The game nodes contain different subnodes depending on whether they represent multi-player or team games. In either case, there are leaf nodes representing the players in the game, their rating at the time the game was played, and their individual score. There is also a number or a dash referenced in parentheses. This represents the *pairwise result* the "top node" player will be awarded for his performance against the player listed on the leaf node. If he (or in the case of a team game, his team) scored higher than the other player (or the other player's team), he will receive 1 point. If he or his team scored lower, then he will receive 0 points. And, if the difference between his score and the other's is less than the draw threshold, then he will receive 0.5 points. If the "leaf node" player and the "top node" player are the same player, or if they are on the same team, then a dash (ie. not applicable) will be shown.

Custom Simulations

Custom simulations are useful for seeing the results of specific scenarios, such as how a rating of 1500 with a rating deviation of 350 would change if the player defeated another player with a rating of 1600 and a rating deviation of 100. The custom simulator also supports simulating multiplayer and team games, so more complex scenarios can also be tested.



1	1,500	350	0.06
2	1,500	350	0.06
3	1,500	350	0.06
4	1,500	350	0.06

Game	Team ID	Player ID	Score
1	1	1	10
1	1	2	3
1	2	3	7
1	2	4	4

Figure 6 - Setting up a custom simulation.

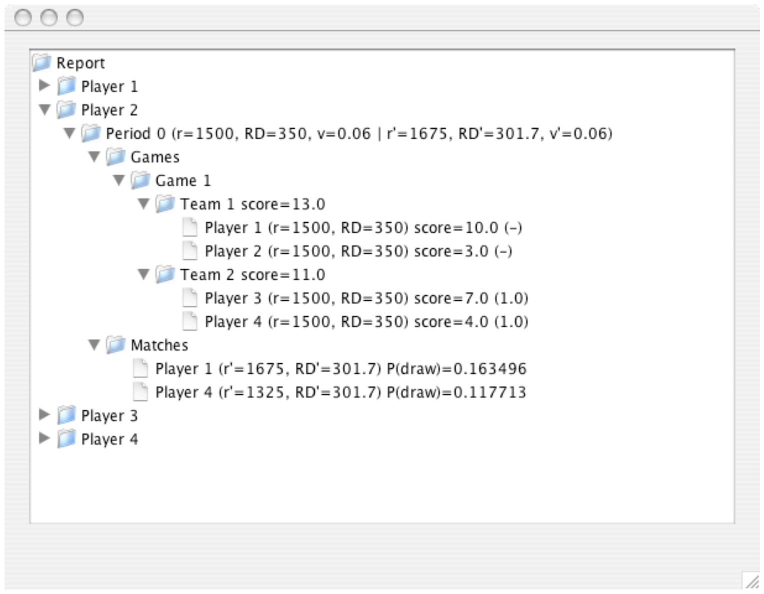
To setup a custom simulation, use the Custom tab in the main window of the Analyzer tool. This panel contains two tables: one that lists the players to be registered with the ranking service, and one that lists the games the players will participate in. Note that the custom simulation cannot simulate more than one rating period, so the results of all games listed will be posted to the same rating period.

To add players to the player list, click the Add button next to the table. This will append a row with a new, unique player id, and values for rating, rating deviation and rating volatility as specified in the System Parameters panel. Once added, the rating values may be edited in place by double-clicking the cell. To remove players from the list, select the row or rows and click the Remove button.

To add games to the game list, click the Add button next to the table. This will append a row to the table which, like the player's table, can be edited in place by double-clicking the cell whose value you want to change. Note that the Team ID field is not optional. If you want to simulate a multiplayer game, use a different value for each row in the Team

ID column. By convention, you can use the player's id as the team id. The Score column contains the player's individual score. For team games, the team score will be the sum of the individual scores of each player on the team.

To run the custom simulation, click the Run button. A window similar to the one depicected below will appear showing the simulation report.



```
Report
▶ Player 1
▼ Player 2
  ▼ Period 0 (r=1500, RD=350, v=0.06 | r'=1675, RD'=301.7, v'=0.06)
    ▼ Games
      ▼ Game 1
        ▼ Team 1 score=13.0
            Player 1 (r=1500, RD=350) score=10.0 (-)
            Player 2 (r=1500, RD=350) score=3.0 (-)
        ▼ Team 2 score=11.0
            Player 3 (r=1500, RD=350) score=7.0 (1.0)
            Player 4 (r=1500, RD=350) score=4.0 (1.0)
    ▼ Matches
        Player 1 (r'=1675, RD'=301.7) P(draw)=0.163496
        Player 4 (r'=1325, RD'=301.7) P(draw)=0.117713
▶ Player 3
▶ Player 4
```

Figure 7 - Report from a custom simulation.

The structure of this report is identical to that of the random simulation report.

Appendix B: RPS - The sample game

A sample game is supplied with the Java Ranking Service to help developers of game servers understand how the ranking service API is intended to be used. The game is the classic Rock, Paper, Scissors game, with extensions for multiplayer and team games. The design goal of the game is to be representative of a typical multiplayer, client-server game, but also be very simple so as not to distract the developer from its primary purpose of exercising the JRS.

The key classes of the game are shown in the class diagram below.

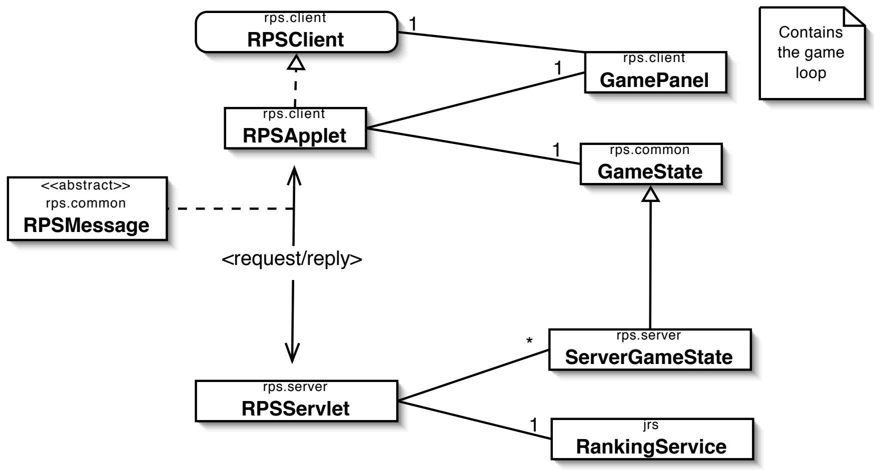


Figure 8 - Key classes of the RPS application

The server is implemented as a servlet, and is shown in the diagram as the `RPSServlet` class. This class manages an instance of the `RankingService` class, and uses this instance to post results, and to get player ratings and match making information. The `RPSServlet` class also manages a subclass of the `GameState` class, which encapsulates the state of a game being played. The subclass, `ServerGameState`, adds additional, server-side logic to the `GameState` class, such as computing scoring, checking for end-game conditions, and other things not suitable to leave to the client.

The game client is implemented as an applet (`RPSApplet`) and implements the `RPSClient` interface. The applet also manages an instance of the `GameState` class. This instance is associated with an instance of `ServerGameState` managed by the server, and is kept in sync with that instance by exchanging messages (ie. instances of `RPSMessage`) with the server. Being a servlet/applet architecture, a request/reply messaging protocol over HTTP is used.

The applet also manages a number of panels within a card layout. These panels correspond roughly to the various states the player may be in, which are shown more formally in the state diagram below.

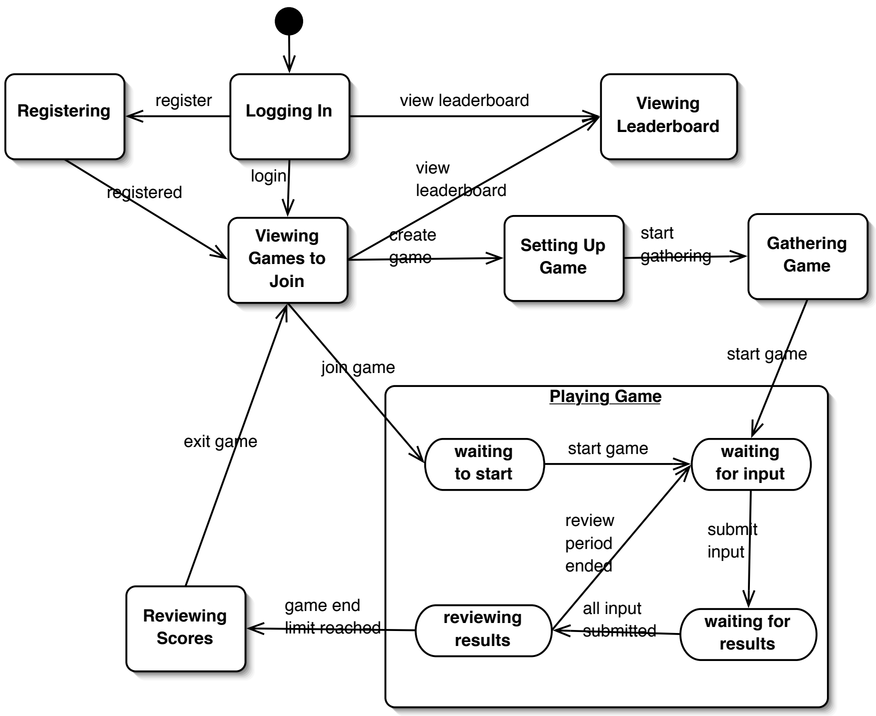


Figure 9 - RPS state diagram

When the applet starts up, players are first presented with the `LoginPanel`. From there, they may login, register as a new player, or view the leaderboard. Once logged in (either directly or indirectly via the registration process), they are presented with list of games being gathered. From here, they may join one of the games listed, create a new game, or view the leaderboard. When joining a game, they move to the `GamePanel` and wait for the host to start the game. If they choose to create a game, they are presented with a panel that allows them to specify the details of the game, such as how many players may participate, whether there are teams or not, and under what conditions the game will end. Once the game is setup, the player is shown the `GatherPanel`, which lists the players that have joined the game. When all players are gathered, the host may then request that the game be started, and all players move to the *waiting for input* state. Here the game waits for the player to enter a choice (ie. rock, paper or scissors). Once a choice has been made, the player moves to the *waiting for results* state. When all players have made their choice, the results of the hand are shown and they all move to the *review results* state. After a few seconds, the players are moved to the *waiting for input* state again, or, if the game is over, to the *Reviewing Scores* state. Players remain in this state until they quit the game, at which time they go back to the `JoinPanel`.

Appendix C: The JRS Workspace

While the JRS package contains all that developers need to use the Java Ranking Service in their game servers, they may find it useful to have the jrs workspace available as well. Having access to the source code makes debugging easier, enhances clarity of the API, and allows developers to fix bugs and add extensions to the service.

This section describes how to checkout the jrs workspace, how it is structured, and how it is built.

Checking out the workspace

The JRS workspace is currently hosted at dev.java.net in a cvs repository. Instructions for setting up a cvs client can be found at:

https://jrs.dev.java.net/servlets/ProjectSource

There are no modules in JRS, so you should simply checkout the entire `jrs` repository.

Each release of the JRS is tagged with a tag name of the form *M-m-stage*, where *M* and *m* are the major and minor release numbers, and *stage* is one of `alpha#`, `beta#`, `rc#` or `fcs`. The # is a sequential number. You should specify the tag name of the release you are working with when performing the checkout.

For example, if you downloaded the `jrs-1.0-beta2.zip` package, you would want to checkout the version of the files tagged with `jrs-1-0-beta2`.

Structure of the workspace

The JRS workspace has the following structure:

`build/`	A directory containing artifacts generated during the build process.
`build.xml`	The `ant` build file.

`dist/jrs-<version>.zip`	A zipped distributable package containing the contents of the `dist/jrs-<version>/` directory. See the JRS Package section for details about the contents of this archive.
`docs/`	Reference documentation for the JRS project.
`lib/`	Third-party libraries used by the JRS.
`src/docs/`	Documentation to be included in the distribution package.
`src/exe/`	Scripts for launching the tools and sample applications.
`src/java/`	Java source files and other files that are often packaged with the generated class files (eg. .gif, .properties, and package.html files)
`src/test/`	JUnit source files. The directory structure under this directory should match the directory structure under the `src/java/` directory.
`src/rps/`	The workspace for the sample game. The contents of this directory will be copied into the distribution package. See the JRS Package section for details about the contents of this directory.

Table 3 - Directory structure of the jrs workspace.

Building the workspace

To build the workspace, your environment must be setup with JDK 1.4 or later, and ant 1.6 or later.

The build file (`build.xml`) defines the following targets:

Target	Description
clean	Remove the `build` and `dist` directories.
compile	Compile all java source files.
docs	Generate JavaDoc for the JRS classes.
jar	Create the `jrs.jar`, `jrs-analyzer.jar`, and `rps-applet.jar` files.
war	Create the `rps.war` file.
run-unit-tests	Run the JRS unit tests.

Target	Description
test	Setup the environment to include the `junit.jar` file in the `CLASSPATH`, then run the unit tests.
dist	Create a distributable directory.
package	Create a zipped archive of the distribution directory.
run-analyzer	Launch the Analyzer tool.
run-rps	Deploy the `rps.war` file to a web container. This target simply invokes the run-rps target defined in the `build.xml` file of the rps workspace. See the JRS Package section for more detail on how this target should be used.

Table 4 - The build targets.