Mobile Games Interoperability Forum

MGIF Platform Specification

Version 1.0

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

Important Notice

Copyright © 2002 Mobile Games Interoperability Forum. All Rights Reserved.

Implementation of all or part of any Specification may require licenses under third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a Supporter). The Sponsors of the Specification are not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS DOCUMENT AND THE INFORMATION CONTAINED HEREIN ARE PROVIDED ON AN "AS IS" BASIS WITHOUT WARRANTY OF ANY KIND AND THE ENTITIES COMPRISING THE SPONSORS AND SUPPORTERS OF THE MOBILE GAMES INTEROPERABILITY FORUM DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL THE ENTITIES COMPRISING THE SPONSORS AND SUPPORTERS OF THE MOBILE GAMES INTEROPERABILITY FORUM BE LIABLE TO ANY PARTY FOR ANY LOSS OF PROFITS, LOSS OF BUSINESS, LOSS OF USE OF DATA, INTERRUPTION OF BUSINESS, OR FOR DIRECT, INDIRECT, SPECIAL OR EXEMPLARY, INCIDENTAL, PUNITIVE OR CONSEQUENTIAL DAMAGES OF NY KIND IN CONNECTION WITH THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE. The above notice and this paragraph must be included on all copies of this document that are made.

Intellectual Property Rights have been asserted or conveyed in some manner toward these Mobile Games Interoperability Forum specifications. The intellectual property rights guidelines of the Mobile Games Interoperability Forum are defined in Section 5.1 of the Mobile Games Interoperability Forum Specification Supporter Agreement. The Mobile Games Interoperability Forum takes no position regarding the validity or scope of any intellectual property right or other rights that might be claimed to pertain to the implementation or use of the technology, or the extent to which any license under such rights might or might not be available. A public listing of all claims against the Mobile Games Interoperability Forum specifications, as well as an excerpt of Section 5.1 of the Mobile Games Interoperability Forum Specification Supporter Agreement, can be found at:

http://www.mgif.org/ipr.html

Contents

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

1. Introduction

This document specifies the requirements of an MGIF certified platform. The primary audience of this document is developers of mobile gaming platform. However, game developers will also find this document useful in determining the scope of functionality that is addressed by the current MGIF release.

The MGIF specifications address the issues of portability and interoperability in the mobile games space. The MGIF specifications will allow game developers to produce and deploy mobile games that can be more easily ported between multiple MGIF platforms and wireless networks, and played over different mobile devices. While the MGIF will not specify functionality for mobile devices, it will work with and provide input to appropriate standards bodies and forums to enable an open end-to-end games environment. The MGIF will define API specifications, it will not produce a mobile game platform.

The potential scope of a gaming platform is enormous. A pragmatic standpoint has, therefore, been taken, where initial efforts have been concentrated in those areas that are deemed to reap most benefit for the game developer. Specifically, in the first release of the API, the following areas are addressed:

- Session management: provides the identifiers that bind the user interactions into single concept of a game, provides access to the other APIs and provides the interface through which the lifecycle of game entities can be managed.
  Rationale: the core framework upon which all other API access is built.

- Connectivity: provides the communication layers, protecting the developer from the low-level implementation details of the transport mechanism.
  Rationale: network access is widely reported to cause significant rework on the part of the game developer.

- Metering: provides a standard API through which the game can inform the MGIF platform of game specific billable events.
  Rationale: relates fundamentally to how the game is paid for and so of high importance.

- Scores and Competition Management: provides the mechanisms by which the game can report and retrieve scores from the MGIF platform, so allowing competitions to be run in a unified manner.
  Rationale: the basis upon which online communities can be built in the mobile gaming arena.

- Logging: provides a standard reporting mechanism by which a game informs the MGIF platform of its status. This insulates against specific formatting requirements and through the implementation of variable logging levels, assists in the troubleshooting process.
  Rationale: by standardizing logging troubleshooting is simplified and thus operational costs reduced.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

- Timers: provides the mechanism by which a game schedules and delays activities.
  Rationale: provides unified access to time based event triggers for the game developer.

Underlying the design of the APIs discussed in this document lays an event-based mechanism. The Session API defines the core of the event model. Although not necessary, a familiarity of event based programming will significantly help in the interpretation of this document.

The MGIF framework of APIs offers no guarantees on the re-entrancy of the event handlers. Specific game platform vendors may offer tools and reentrance conditions on top of the MGIF APIs, but at this moment in time this is an implementation specific area.

This document is intended for readers familiar with the Java programming language.

In order to achieve compliance with 1.0 MGIF specification, platform support for each of the APIs listed within this document is mandatory.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

2. Session Management

2.1 Description

The Session Management APIs describe a framework and high level structure for the applications executing within the MGIF platform. This framework controls the application's lifecycle. It also facilitates managing the Actors, meaning the End-User representation within the specific Application, and provides the Application Developer with access to all other interfaces and APIs necessary to create the application. These include, the Connectivity, Logging, Scoring, etc.

MGIF APIs are event based. Each event handling is a separate transaction. There are numerous types of events: user input events, timer events, etc. The full list of possible events is found below. To connect between the game logic and the MGIF platform event handling mechanism the game logic source classes **must** implement one or more of supplied event listener interfaces. Those game logic classes should be registered in MGIF platform by the deployment process.

Game board lifecycle in MGIF platform is based on session entities. A session is defined as a series of interactions encompassing the Actor's lifetime within the specific Application Instance. `ActorSession` represents the single player role in a game board. In the real world, games may contain several players, and one single person can play several roles in different games or events in a single game board (person plays chess with himself), therefore the relation between `ActorSession` and `User` (which represents a user) is many to one. The relation between `ActorSession` and `ApplicationInstance` (which represents a single game board) may similarly be many to one. The `Application` session entity is used to represent a single type of game, registered in the MGIF platform, and defines shared functionality between all `ApplicationInstances` of the same kind. `Actor` is shared functionality between all `ActorSessions` of same `User` in the same `Application`. More information about functionally provided in each session interface is provided in the detailed documentation in JavaDoc Appendix A.

The MGIF platform **must** maintain persistent relations among all session objects for all existing game boards. Session objects and the relations among them may be changed automatically by an MGIF platform as a result of some event, e.g. user input can create new `ActorSession`, or as a result of game logic execution, e.g. as result of `ActorSession.delete()` call.

When an event arrives at the system, the transaction opens and a transaction context is created. Among other things context contains the target of the handling event. There are two kinds of event, the first targets `ApplicationInstance` session entity, while the second targets `ActorSession` session entity. The MGIF platform **must** create the correct context for every arriving event.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

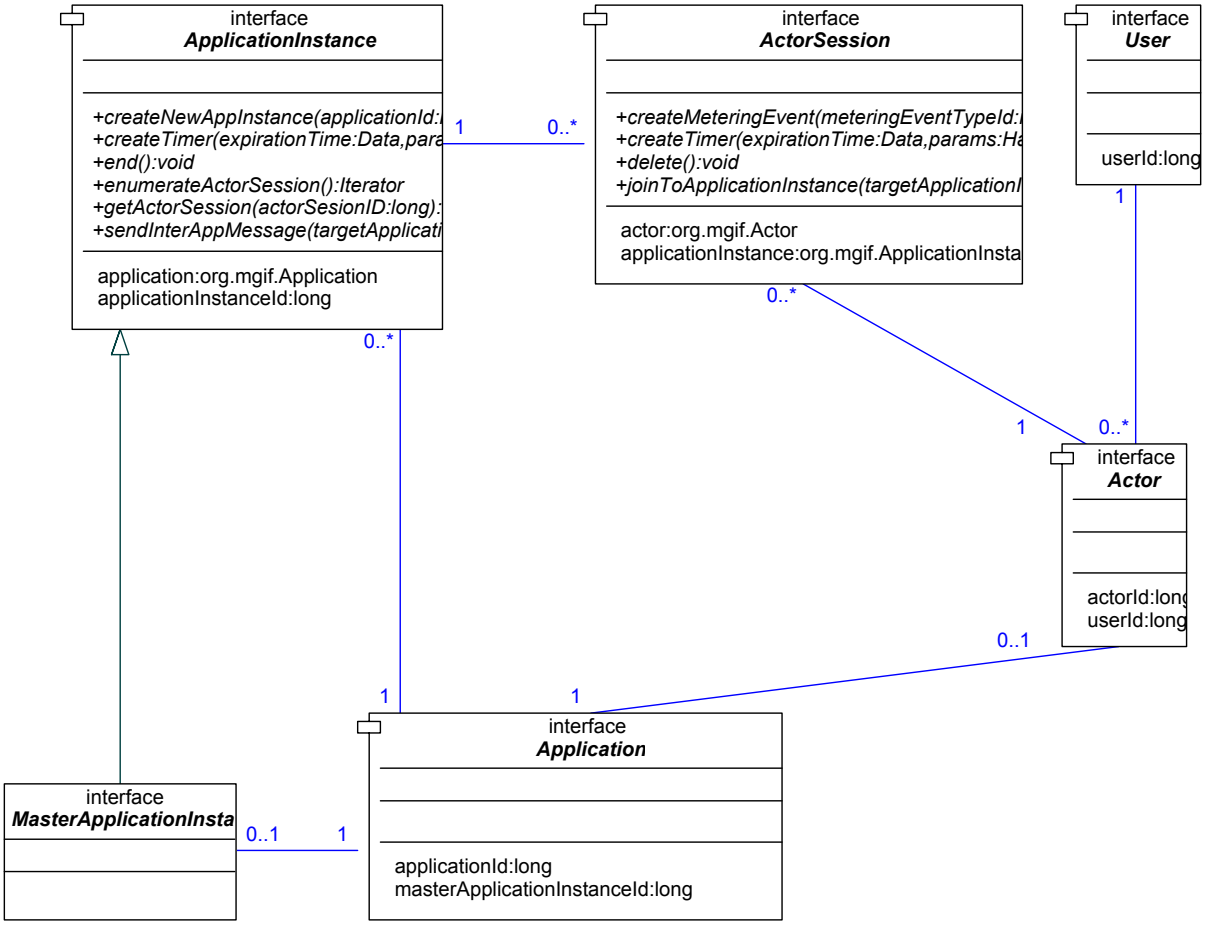


Figure 1. Relations between main session entities

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

2.1.1 Interfaces

2.1.1.1 Actor

Actor represents shared information between all `ActorSessions` in same `Application`. It contains only an `ActorID` that is used to access this shared information. This state is created when a user starts to use `Application` for first time, and exists forever.

2.1.1.2 ActorSession

An `ActorSession` object represents a specific user in the context of a particular `ApplicationInstance`. Each user can be present in multiple applications at the same time, and thus be associated with several `ActorSession` objects. There is a one-to-many relationship between the `Actor` and `ActorSession`.

An `ActorSession` is created by the MGIF platform and joined to the appropriate `ApplicationInstance`. `ActorSession` interface contains `ActorSessionID` which combined with `ApplicationInstanceID` and `ApplicationID` can be used to access `ActorSession` persistent state.

The corresponding `ActorSession` object in the game handles inputs received from a specific user.

An `ActorSession` is the actual representation of a user session in a game.

2.1.1.3 Application

An `Application` is the installed code or logic of a game. The `Application` is used for creating specific running instances: game boards. `Application` defines shared information for all `ApplicationInstances`. It also contains `ApplicationID` that is used to manage this information.

2.1.1.4 ApplicationInstance

The `ApplicationInstance` is the actual game played. It is the running instance of the `Application` object. For a game to be created there must be a new `ApplicationInstance` created to manage the actual game and the `ActorSessions` in it. In this way, multiple instances of an application can be run simultaneously, each controlling a different game and its users. A specific Tic-Tac-Toe board is an example of an `ApplicationInstance`.

Every `ApplicationInstance` contains an `ApplicationInstanceID` which may be used to access the persistent information.

`ApplicationInstance` is target for `Application` events and implements listeners for those events.

An `ActorSession` is the actual representation of a user session in a game.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

2.1.1.5 ApplicationInstance - ActorSession Relations

Optional for version 1.0 of MGIF platform:

An `ApplicationInstance` may manage several actors simultaneously.

A user may be represented simultaneously in several `ApplicationInstances`. A user may simultaneously play in all those game boards.

Note: The decision as to whether a user may play simultaneously in several instances of the same game is a commercial production decision of the operator. The application cannot make any assumptions to that effect.

2.1.1.6 MasterApplicationInstance

`MasterApplicationInstance` defines one special application instance, which is used to manage events and information shared for regular application instances. This instance is the target for special management events, e.g. declarative timers.

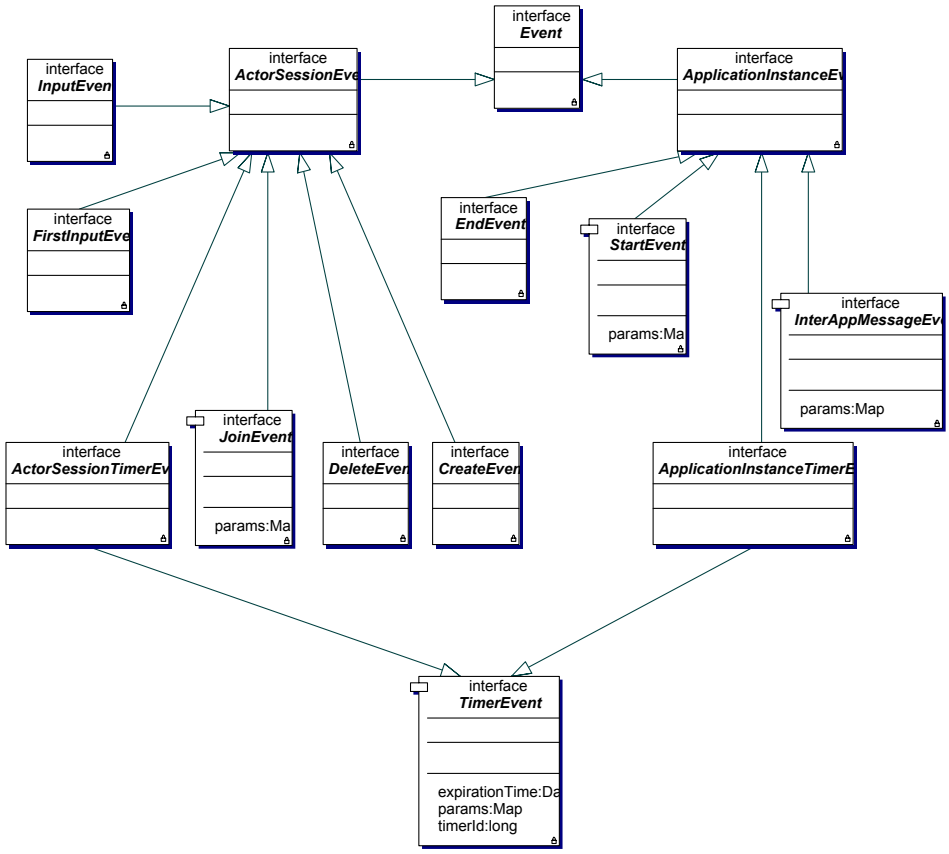
2.1.1.7 User

Users are subscribers who have cellular accounts and may access the MGIF platform. The term “user” represents a specific person connecting to the system via a cellular phone or another communication device. The user is an object that exists independently of any game. The user places requests to the system to play a particular game. Normally a user is identified with a SIM (Subscriber Identity Module) or User Name and Password. Each user has an internal unique `UserID` on the platform.

When the user is connected to an `Application`, an instance of the `ActorSession` class is created, by the MGIF platform, for the user, and joined to that `Application` in the form of an `ActorSession` object. It is possible for multiple `ActorSession` objects to exist simultaneously for a particular user when each `ActorSession` object is attached to an `ApplicationInstance` but controlled by the same user.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

Events List



2.2.1 Events

2.2.1.1 Actor Session

Base interface for all events that have actor as a target

2.2.1.2 Actor Session Timer

Indicates timer event for Actor object

2.2.1.3 Application Instance

Base interface for all events that have application instance object as target.

2.2.1.4 Application Instance Timer

Indicates timer event for Application Instance object

2.2.1.5 Create

Indicates start of lifecycle of Actor Session object.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

2.2.1.6 Delete

indicates end of lifecycle of Actor Session object.

2.2.1.7 Delivery

Base interface for delivery reports.

2.2.1.8 ActorSessionDelivery

Event created as a result of delivery report to ActorSession object.

2.2.1.9 ApplicationInstanceDelivery

Event created as a result of delivery report to ApplicationInstance object.

2.2.1.10 End

Indicates end of lifecycle of Application Instance object.

2.2.1.11 Event

Base interface for all possible events

2.2.1.12 Input

Base class for all input events.

2.2.1.13 AsyncInputEvent

Event created as a result of input to ActorSession object

2.2.1.14 SyncInputEvent

Event created as a result of request/response input to ActorSession object

2.2.1.15 Inter App Message

Indictes message sent to Application Instance object by some other Application Instance object.

2.2.1.16 Join

Indicates request for join of user from some Actor Session object to another freshly created Actor Session object. This event immediately follows CreateEvent.

2.2.1.17 Start

2.2.1.18 Timer

Base class for al timer events.

2.3 Action Listeners List

2.3.1.1 OnActorSessionDelivery

This interface declares that implementing ActorSession class is ready to listen for incoming delivery reports.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

2.3.1.2 OnActorSessionJoin

This interface declares that implementing ActorSession class is ready to listen for JoinEvent from some other ApplicationInstance and implements a hook to deal with transferred event.

2.3.1.3 OnActorSessionTimer

This interface declares that implementing ActorSession class is ready to listen for TimerEvents implements a hook to deal with transferred event. To create TimerEvent use ActorSession.createTimer() call.

2.3.1.4 OnApplicationInstanceDelivery

This interface declares that implementing ApplicationInstance class is ready to listen for incoming delivery reports.

2.3.1.5 OnApplicationInstanceTimer

This interface declares that implementing ApplicationInstance class is ready to listen for TimerEvents implements a hook to deal with transferred event. To create TimerEvent use ApplicationInstance.createTimer() call.

2.3.1.6 OnAsyncInput

This interface declares that implementing ActorSession class is ready to listen for asynchronous input event and implements a hook to deal with transferred event.

2.3.1.7 OnCreate

This interface declares that implementing ActorSession class is ready to listen for CreateEvent and implements a hook to deal with transferred event. This event is automatically created when ActorSession created, and this is first event that should be handled by ActorSession.

2.3.1.8 OnDelete

This interface declares that implementing ActorSession class is ready to listen for CreateEvent and implements a hook to deal with transferred event. This event is automatically created when ActorSession created, and this is first event that should be handled by ActorSession.

2.3.1.9 OnEnd

This interface declares that implementing ApplicationInstance class is ready to listen for EndEvent and implements a hook to deal with transferred event. This event is automatically created when ApplicationInstance deleted, and this is last event that should be handled by ApplicationInstance.

2.3.1.10 OnFirstAsyncInput

This interface declares that implementing ActorSession class is ready to listen for first asynchronous input event and implements a hook to deal with transferred event.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

2.3.1.11 OnFirstSyncInput

This interface declares that implementing ActorSession class is ready to listen for first synchronous input event and implements a hook to deal with transferred event.

2.3.1.12 OnInterAppMessage

This interface declares that implementing ApplicationInstance class is ready to listen for InterAppMessageEvent and implements a hook to deal with transferred event.

2.3.1.13 OnStart

This interface declares that implementing ApplicationInstance class is ready to listen for StartEvent and implements a hook to deal with transferred event. This event is automatically created when ApplicationInstance created, and this is first event that should be handled by ApplicationInstance.

2.3.1.14 OnSyncInput

This interface declares that implementing ActorSession class is ready to listen for synchronous input event and implements a hook to deal with transferred event.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

3. Connectivity

3.1 Description

The purpose of the connectivity APIs is to enable communication between the application and the clients. The connectivity APIs specify how the requests from clients are exposed to the applications, and how applications generate responses to the clients.

The communication models required by different application types can be categorized into four modes:

- client pull
- client push
- application pull
- application push

This version of the connectivity APIs only addresses messaging and browser clients. Subsequent versions will include executable clients.

3.2 Content

This section illustrates the components that collectively comprise the Connectivity APIs. This API comprises of three parts:

- synchronous communication
- asynchronous communication
- transfer, dealing with functionality common to each of the above types of communications

The Session APIs provide listener hooks for both synchronous and asynchronous communication. An application serving requests from both communication types can simply provide implementation for the both `onSynchInput`() and `onAsynchInput`() methods. The protocol used by the client determines which method the platform calls whenever a new request is received from a client. The routing of requests to the correct application instance is implementation specific.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

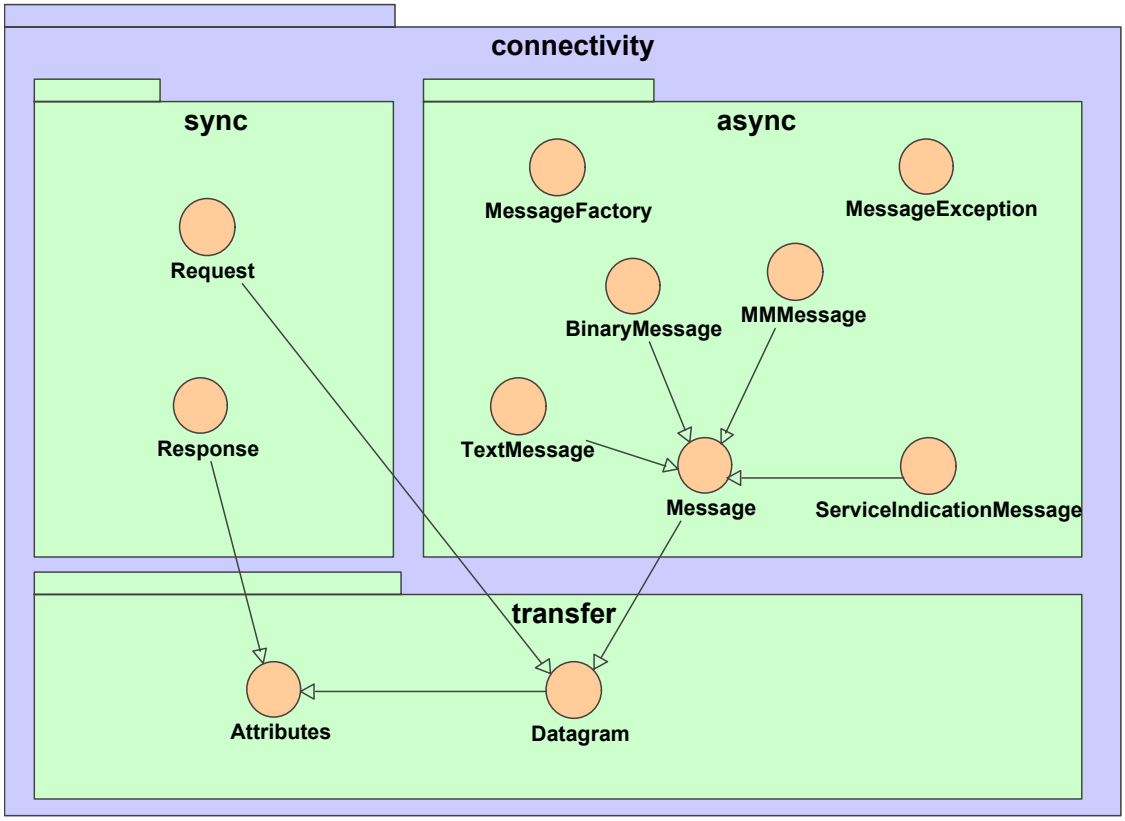


Figure 2. Connectivity package

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

3.3 Async Package

The `async` package manages the asynchronous communication.

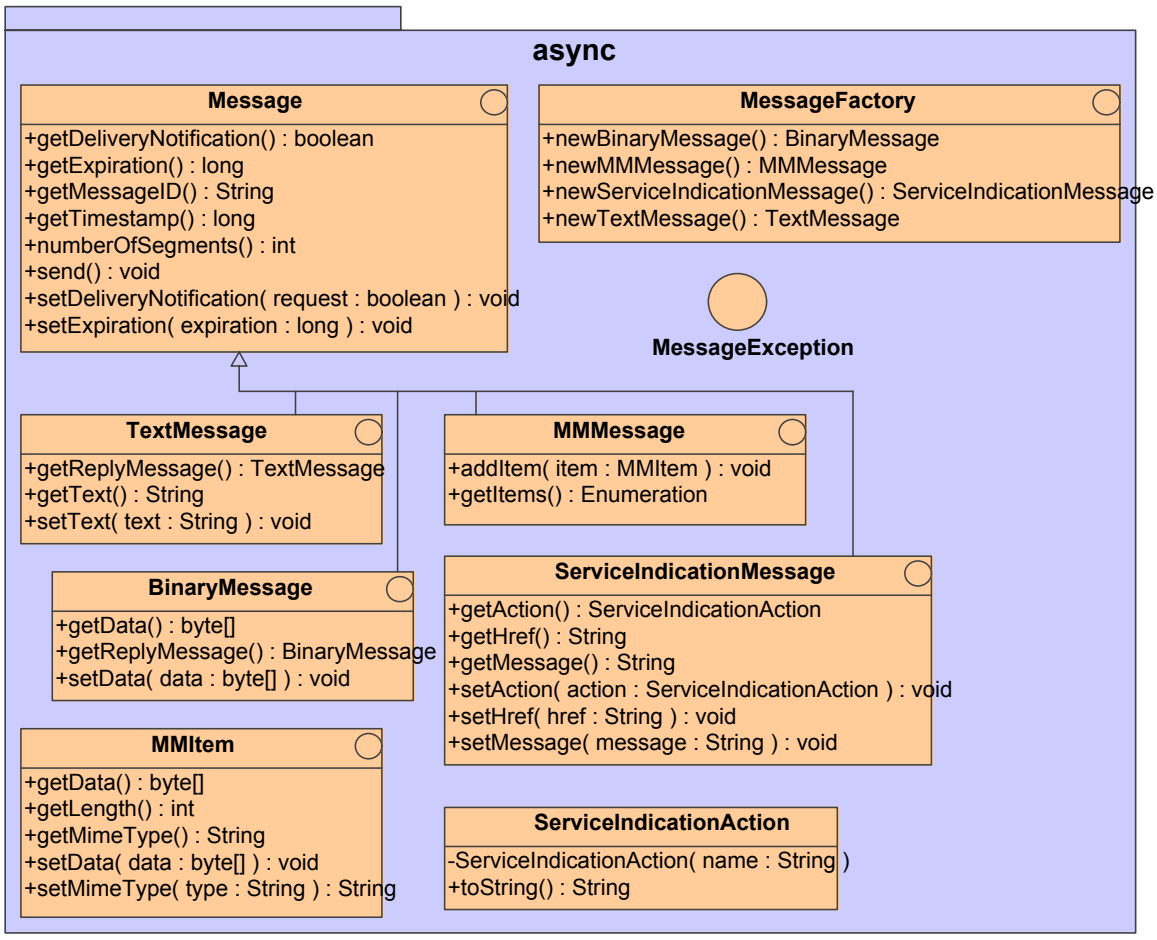


Figure 3. Async package

The asynchronous communication uses messages. The base interface for all messages is `org.mgif.connectivity.async.Message`. The interface provides basic common methods for handling messages.

3.3.1 Mobile originated messages

Incoming messages are delivered to the application via the `OnFirstAsyncInput` and `OnAsyncInput` listeners.

3.3.2 Mobile terminated messages

Message interfaces contain a `getReplyMessage()` method that provides an easy way to generate a response to an MO message. The `MessageFactory` is used for pushing an MT message to a user. The `MessageFactory` is obtained from the `ActorSession`.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

Sample code snippet in Java (receiving and responding to an asynchronous client request):

```
 import org.mgif.connectivity.async.*;

 ...
  public void onAsyncInput(AsyncInputEvent event)
  {
    TextMessage textMessage = (TextMessage)
event.getMessage();
    if (textMessage.getText().equals("Hello MGIF!"))
    {
      TextMessage replyMessage
        = textMessage.getReplyMessage();
      replyMessage.setText("Hello!");
      replyMessage.send();
    }
  }
```

The basic message interface is inherited to produce service specific message types.

- `org.mgif.connectivity.async.TextMessage` is an interface for handling messages containing only textual data.
- `org.mgif.connectivity.async.BinaryMessage` is an interface for handling messages containing binary data, e.g. operator logos.
- `org.mgif.connectivity.async.ServiceIndicationMessage` is an interface for handling WAP push.
- `org.mgif.connectivity.async.MMMessage` is an interface for handling Multimedia messages

For details please refer to the Javadoc of the corresponding interfaces.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

3.3.3 Sync package

The sync package handles the synchronous communication.

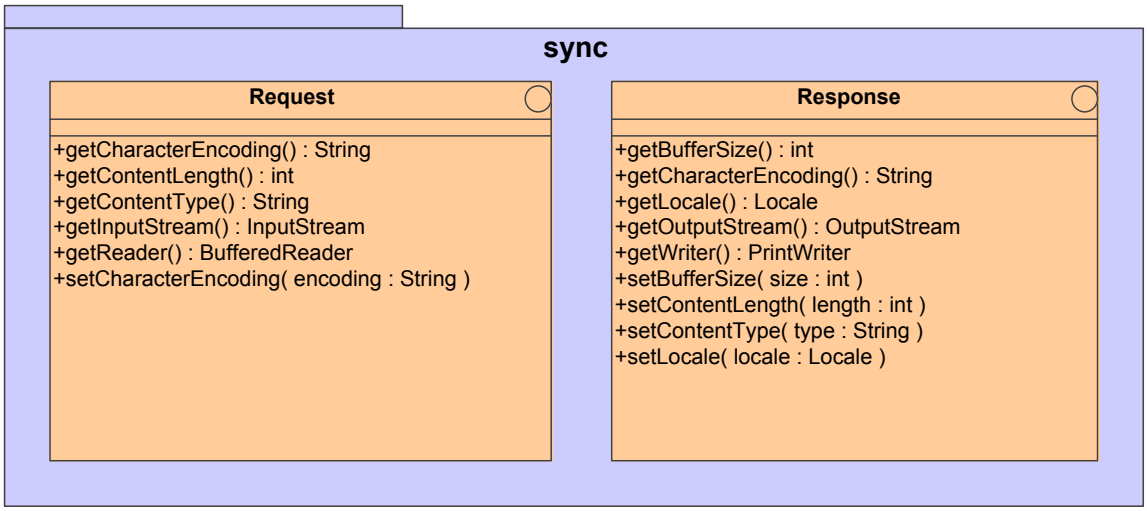


Figure 4. Sync package

Synchronous communication uses the request/reply paradigm. This is realized by using the `org.mgif.connectivity.sync.Request` and `org.mgif.connectivity.sync.Response` interfaces. Both interfaces contain methods for getting and setting attributes, as well as other methods for dealing with the actual content of the request or response.

Sample code snippet in Java (receiving and responding to an synchronous client request):

```
import org.mgif.connectivity.sync.*;

...

 public void onSyncInput(SyncInputEvent event)
 {
   Request request = event.getRequest();
   if (request.getContentType().equals("text/html"))
   {
     Response response = event.getResponse();
     response.getWriter().println("<html>");
     response.getWriter().println("<h1>Hello</h1>");
     response.getWriter().println("</html>");
   }
 }
```

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

3.3.4 Transfer package

The transfer package contains the interfaces extended by both the async and sync packages.

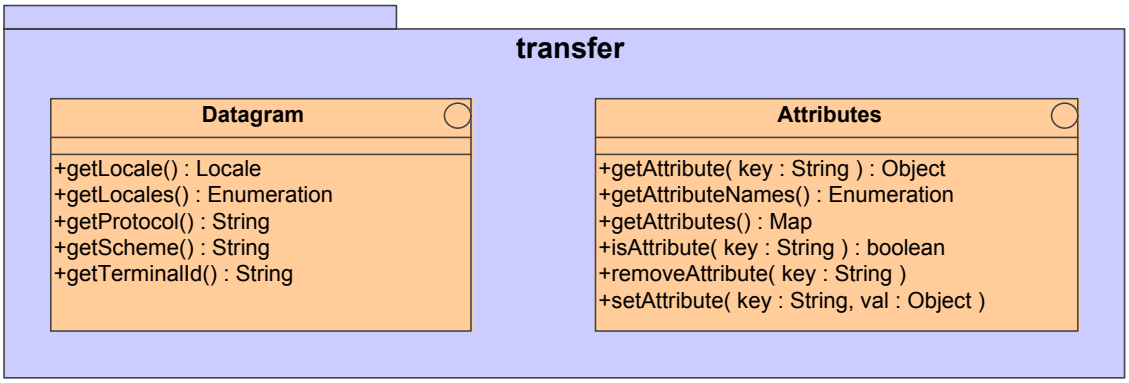


Figure 5. Transfer package

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

4. Metering

4.1 Description

An MGIF platform **may** produce metering events for potential use in different billing scenarios. Some metering events are traffic related, e.g. game session duration, MT SMS message, and some are application specific, e.g. moving to the next level in a game.

An MGIF platform **may** use metering events for post-paid, pre-paid and subscription scenarios.

The scope of these APIs is to address the generation of game specific metering events.

4.2 Traffic Based Events

An MGIF platform **may** implement advanced, flexible metering services. A platform **may** provide metering of:

- Session duration
- Data transfer
- MT and MO message
- Subscription/Pre-Paid Billing authorization

The application developer is not responsible for adding code for metering of traffic related events. Such metering is handled transparently by the MGIF platform.

4.3 Game Specific Events

The MGIF Metering APIs are used to programmatically commit metering events. Each metering event is created using a metering event type `ID`. All metering event types **must** be configured prior to use. Processes or tools for configuration are undefined. An MGIF Platform vendor may choose the mechanism for configuring these events. Each metering event type **must** have an integer `ID`. Metering events are created in the scope of an `ActorSession`.

Sample code snippet in Java (metering event creation):

```
import org.mgif.*;
import org.mgif.metering.*;

...

// Create a metering event for moving to the next level
// (ID=17)
// An event type with ID=17 must have been configured
```

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

```
MeteringEvent event =
ActorSession.createMeteringEvent(17);
event.raise();
```

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

5. Score and Competition Management

5.1 Description

These APIs provide the mechanisms for recording and retrieving scores. This allows various forms of competition to be provided by either the platform or the individual game application.

5.1.1 Scoring

The MGIF platform **must** support the following scoring models for applications. An application **must** state the scoring model it will use for each of the scores it records, and **must** reliably record these scores for each player for each session following the stated model.

Each model may be further configured according to scoring information, e.g. May more than one score be stored for a particular user? Or are scores "better" when higher or lower?

5.1.1.1 No Scoring

No scores are recorded and no score table information is available. No further configuration required. This is often used in games in which there is no simple score available. Such games need to implement their own internal competition mechanisms as they will not be able to assume support from the platform.

5.1.1.2 Simple Numeric Scores

The game service records a simple numeric score for each user. This model **must** be configured according to scoring information:

- Is "better" higher or lower?
- The number of scores that may be recorded per user.
- The total number of scores that are to be recorded.

The game application **must** record a score at least once for each player in each game session.

5.1.1.3 Cumulative Numeric Scores

The user builds up a score over a number of sessions. This scoring model **must** be configured according to scoring information:

- Is "better" higher or lower?
- The total number of scores that are to be recorded.

In this case the game application first retrieves the user's current score, alters it based on the outcome of the session and then records the updated score.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

5.1.1.4 Rank

This scoring model is provided to support services that maintain their own score table internally by some mechanism not supported within the MGIF specification. The game service simply calls setScore with the players new rank as the value – all lower ranked positions are moved down to make room. This scoring model **must** be configured according to scoring information:

- The total number of scores that are to be recorded.

5.1.1.5 Combined

An application can specify any number of score tables, each using one of the model described above.

5.1.2 Competitions

Competition management is normally a feature of the game server and not the individual game, consequently there is no specific API provided to support competitions. For a games service to be useable for competitions it **must** record a score for every user in every game session. Such scores **must** always be reliably comparable.

For every such games service the game platform **may** provide facilities for competitions running over various time periods with the winner selected in a variety of ways – in part depending on the nature of the scoring used in the game service in question.

If a game service wishes to implement a service specific competition system it may do so using a combination of the Score Management API and Scheduling API. This is discouraged as it is likely to duplicate platform functionality.

5.2 Content

5.2.1 Scores

All scores returned from the various interfaces defined below return objects implementing the `org.mgif.score.Score` interface. This is a simple bean style interface allowing access to the score value, the rank it represents from where the score was retrieved, if applicable, when the score was achieved and who achieved it.

Score values are always represented by the `int` type. If a game requires fractional scores it should scale these to produce an integer representation with an appropriate number of significant digits.

5.2.2 Multiple Score Tables

An application can specify any number of score tables within reason. These are referred to via a table number. All of the APIs in this chapter are provided in two forms. The first form does not specify a table number and operates on the first or only table. The second form allows the table number to be specified.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

5.2.3 Recording Scores

All score recording is done via an object implementing the `org.mgif.score.ScoreManager` interface that can be retrieved from the `ActorSession`.

The `setScore` method allows a score value to be set for the session. Normally it is assumed the score was achieved at the time the `setScore()` method was called. Optionally the time/date it was achieved may be provided explicitly.

The `getScore()` method retrieves the last score set for this session, or from a previous session if the cumulative model is in use.

5.2.4 Retrieving Past Scores

All score retrieval is done via an object implementing the `org.mgif.score.ScoreTableManager` interface which can be retrieved from the `ApplicationInstance`.

The simplest method is `getScoreAt()` which retrieves a single score from a particular rank. This will return null if no score is available at that rank.

The other methods all return an array of `Score` objects that may vary in size between 0 and the requested number of scores if the scores requested are available or are not in the table in question.

The top scores in the table can be retrieved by using the `getTopScores()` method.

The scores around a particular rank, or the rank of the highest score of some specific `Actor`, may be retrieved by using the `getScoresAround()` method.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

6. Timers

6.1 Description

Applications that need to delay or schedule activities for a later time should use the timer service provided by the MGIF platform. The `Timer` service provides scheduling and notification of timers. An MGIF platform **may** provide additional unspecified services, e.g. load balancing and persistence of timers.

Timers can be created either programmatically, i.e. by calling a method on an interface, or declaratively, i.e. via some implementation specific mechanism for configuring timers.

6.2 Programmatic Timers

There are two types of programmatic timers, `ActorSession` timers and `ApplicationInstance` timers.

`ActorSession` timers are created and notified in the scope of an `ActorSession`. The `createTimer()` method on the `ActorSession` interface is used for creating these timers. When such a timer expires the method `onActorSessionTimer()` on the `OnActorSessionTimer` interface is invoked.

`ApplicationInstance` timers are created and notified in the scope of an `ApplicationInstance`. The `createTimer()` method on the `ApplicationInstance` interface is used for creating these timers. When such a timer expires the method `onApplicationInstanceTimer()` on the `OnApplicationInstanceTimer` interface is invoked.

Sample code snippet in Java (create timer)

```
import org.mgif.*;

...
// Create a timer in ActorSession scope
Hashtable params = new Hashtable();
params.put("myParam", "myValue");
myActorSession.createTimer(new Date((new
Date()).getTime() + 60*60*1000), params);

...
```

Sample code snippet in Java (timer notification)

```
import org.mgif.*;
import org.mgif.listener*;

public class MyTask implements OnActorSessionTimer {
```

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

```
  public void onActorSessionTimer(ActorSessionTimer
event) {
    // This method is called when the timer expires
    Map params = event.getParams();
  }
}
```

6.3 Declarative Timers

Declarative timers are created via an implementation specific mechanism, e.g. a configuration file or administration tool. . When such a timer expires the method `onApplicationInstanceTimer()` on the `OnApplicationInstanceTimer` interface is invoked. All declarative timers are notified for the `MasterApplicationInstance` entity.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

7. Logging

7.1 Description

An MGIF platform **may** provide logging of events of all client requests and MT SMS messages, timer events, etc. For debugging, troubleshooting, monitoring and other purposes it can be important for the application developer to be able to write information to an application log. The logging APIs of the MGIF platform specification is designed for these purposes.

7.1.1 Logger Interface

The `Logger` interface from MGIF logging APIs, can be used to add information to the log. The `ActorSession` and `ApplicationInstance` interfaces can be used to retrieve a `Logger`.

Sample code snippet in Java

```
import org.mgif.util.logging.*;

...

Logger logger = myActorSession.getLogger();
logger.fine("Moving to the next level.");
// Move to the next level
...
if (successful) {
   Log.info("The move to the next level was
successful.");
} else {
   Log.warning("The move to the next level was
       unsuccessful.");
```

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

8. Future API Expansion

The MGIF is committed to the improvement of the implementation of games in the mobile environment. To this end further APIs may be added to the specification as and when they are required. Two areas that have been identified for immediate attention:

- Game Administration – Tools for game deployment configuration and administration.
- Executable client applications – which although not a specific API as such, entails the broadening of scope of MGIF to encompass games where some of the game logic does not reside on the server.

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

Appendix A JavaDocs

The related JavaDocs for the MGIF Platform Specification are freely available at:

http://www.mgif.org/docs/MGIF_JavaDocs_v1.0.zip

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

Appendix B Sample Code

```
package org.mgif.examples.RPS;

import org.mgif.listener.OnFirstAsyncInput;
import org.mgif.listener.OnAsyncInput;
import org.mgif.listener.OnActorSessionTimer;
import org.mgif.listener.OnDelete;
import org.mgif.event.AsyncInputEvent;
import org.mgif.event.DeleteEvent;
import org.mgif.event.ActorSessionTimerEvent;
import org.mgif.ActorSession;
import org.mgif.util.logging.Logger;
import org.mgif.score.Score;
import org.mgif.score.ScoreManager;
import org.mgif.connectivity.async.MessageFactory;
import org.mgif.connectivity.async.TextMessage;
import org.mgif.connectivity.async.Message;

import java.util.Map;
import java.util.Date;
import java.util.HashMap;

/**
 * A very simple Rock-Paper-Scissors text message game written to the MGIF v1 API
 * The game automatically ends after 24 hours.
 */
public class RockPaperScissors implements OnFirstAsyncInput, OnAsyncInput,
OnActorSessionTimer, OnDelete
{
    /**
     * Number of milliseconds on 24 hours!
     */
    private static final int ONE_DAY = 1000*60*60*24;

    /**
     * The three possible values to play!
     */
    private static final int ROCK = 0;
    private static final int PAPER = 1;
    private static final int SCISSORS = 2;

    /**
     * Handle a first input in a session.
     */
    public void onFirstAsyncInput(AsyncInputEvent event)
    {
        Message mo = event.getMessage();
        ActorSession actorSession = event.getActorSession();
        Logger logger = actorSession.getLogger();

        // Create an MT message to send back out.
        TextMessage mt;
        if (mo instanceof TextMessage) {
            logger.info("RPS: Session started for "+mo.getOriginator()+" by a text
message.");
            mt = ((TextMessage)mo).getReplyMessage();
        } else {
            logger.info("RPS: Session started for "+mo.getOriginator()+" by a non-text
message.");
            mt = actorSession.getMessageFactory().newTextMessage();
        }

        // Fill in the MT and send it.
        mt.setText("Welcome to rock paper scissors - take your pick of r,p, or s");
        mt.send();

        // Create the timer which will end the game.
        Date oneDayInTheFuture = new Date((new Date()).getTime() + ONE_DAY);
        Map args = new HashMap();
        actorSession.createTimer(oneDayInTheFuture, args);
```

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

```
    }

    /**
     * Handle all subsequent inputs in a session.
     */
    public void onAsyncInput(AsyncInputEvent event)
    {
        Message message = event.getMessage();
        ActorSession actorSession = event.getActorSession();

        if (message instanceof TextMessage) {

            TextMessage mo = (TextMessage)message;

            // Try to act on the content of the MO.
            Result result = null;
            if (mo.getText().equalsIgnoreCase("r")) {
                result = takeTurn(ROCK);
            } else if (mo.getText().equalsIgnoreCase("p")) {
                result = takeTurn(PAPER);
            } else if (mo.getText().equalsIgnoreCase("s")) {
                result = takeTurn(SCISSORS);
            }

            TextMessage mt = mo.getReplyMessage();
            if (result!=null) {
                // We have a result so send out an appropriate MT.

                if (result.isUserWin()) {
                    // If the user won then update their score.
                    ScoreManager scoreManager = actorSession.getScoreManager();
                    Score currentScore = scoreManager.getScore();
                    scoreManager.setScore(currentScore.getValue()+1);
                }

                mt.setText(result.getDescription());

            } else {
                // We didn't understand the MO so send an error as out MT.
                mt.setText("I don't understand. Please choose one of r, p or s!");
            }
            mt.send();

        } else {
            // If the MO wasn't a text message push out an error.
            TextMessage mt = actorSession.getMessageFactory().newTextMessage();
            mt.setText("Rock-Paper-Scissors can only respond to plain text messages -
sorry!");
            mt.send();
        }
    }

    /**
     * Handle timer events.
     */
    public void onActorSessionTimer(ActorSessionTimerEvent event)
    {
        ActorSession actorSession = event.getActorSession();
        MessageFactory messageFactory = actorSession.getMessageFactory();

        // Let the user know the time has run out!
        TextMessage mt = messageFactory.newTextMessage();
        mt.setText("Your game time has run out!");
        mt.send();

        // Close the session.
        actorSession.delete();
    }

    /**
     * Handle session end.
     */
    public void onDelete(DeleteEvent event)
    {
        ActorSession actorSession = event.getActorSession();
```

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.

```
        ScoreManager scoreManager = actorSession.getScoreManager();
        MessageFactory messageFactory = actorSession.getMessageFactory();
        Logger logger = actorSession.getLogger();

        // Send the user a summary of the session.
        TextMessage mt = messageFactory.newTextMessage();
        mt.setText("Your final score was "+scoreManager.getScore());
        mt.send();

        logger.info("RPS: Session finished for "+mt.getDestination());
    }

    /**
     * Table of results against user and cpu choices.
     */
    private static Result results[] = {
        new Result(false, "Rock draws with rock."),
        new Result(false, "Paper wraps rock - you lose."),
        new Result(true, "Rock blunts scissors - you win!"),
        new Result(true, "Paper wraps rock - you win!"),
        new Result(false, "Paper draws with paper."),
        new Result(false, "Scissors cut paper - you lose."),
        new Result(false, "Rock blunts scissors - you lose."),
        new Result(true, "Scissors cut paper - you win!"),
        new Result(false, "Scissors draws with scissors.")
    };

    /**
     * The actual game "logic" - table driven.
     */
    private Result takeTurn(int userChoice)
    {
        int cpuChoice = (int)(Math.random()*3);
        boolean userWin = false;
        String description = null;

        Result result = results[userChoice*3+cpuChoice];

        return result;
    }

    /**
     * Simple bean representing the result of a round.
     */
    private static class Result
    {
        private boolean userWin;
        private String description;

        public Result(boolean userWin, String description)
        {
            this.userWin = userWin;
            this.description = description;
        }

        public boolean isUserWin()
        {
            return userWin;
        }

        public String getDescription()
        {
            return description;
        }
    }
}
```

Copyright © 2002 Mobile Games Interoperability Forum. All rights reserved.