Game Interface Component 1.0 Specification

1. Design

The Orpheus Game Logic component provides business logic in support of game play and game data manipulation tasks performed by the Orpheus application. For the most part this involves providing `Handler` implementations conformant to the specifications of the Front Controller component version 2.1, but some operations that are triggered by internal events or that support other components will be provided by an internal API, which is the subject of this component.

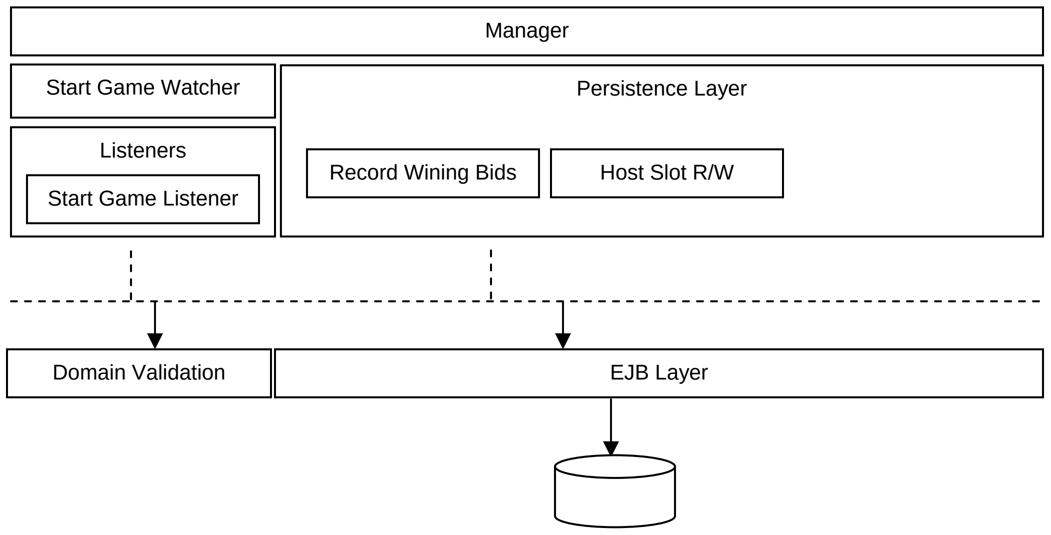
1.1 General approach

1.1.1 Anatomy of the proposed design

The basic tasks of this design can be broken up as follows:

1. We will have a general manager, which will coordinate all the efforts of this component. The user will mostly be interacting with the manager interface in a façade like manner.
2. We will create a number of internal listeners which will react to some events and which will trigger a proper behavior in the manager so that it can have 'up-to-date' information ready for the user.
3. We will have the ability to read/write to the persistence supporting the data for games in general.

The basic outline of the design is as follows:



The way that we can read the above is as follows:

1. Manager will facilitate the ability to act somewhat like a state-machine in the sense that it will 'watch' changes to its internal state (in terms of start game data) and when the time arrives for a new game to start it will trigger an event and call the registered (internal) listener
2. We will also facilitate here an indirection to the persistence layer so that we can read or write data pertinent to the task of the manager.

1.1.2 Dealing with EJB lookups and local vs. remote interfaces

When we are dealing with EJB lookup it is always preferable to use the local interface rather than the remote one. The solution here will be done as follows:

1. We will allow the manager to be configurable with a jndi name and a designation which states if it is a local interface lookup or a remote interface lookup.
2. This will be done through Object Factory and the ability to configure the constructor to accept jndi names as follows:

```
// GameDataManagerImpl constructor
GameDataManagerImpl(jndiNames String[], jndiDesignations[], . . .);

// Configuration data which names both the local and remote
// interface lookups. These are done in order from index 0..n
jndiNames[] = new String[]{“myLocalInterfaceLookupName”
                            , “myRemoteInterfaceLookupName”);
jndiDesignations[] = new String[]{JndiLookupDesignation.LOCAL
                            , JndiLookupDesignation.REMOTE);

// Will create a new manager which will lookup the necessary EJB
// in order if index. So it will first lookup the local one and if that
// fails it will then lookup the remote one.
GameDataManager manager =
        new GameDataManagerImpl(jndiNames, jndiDesignations, . . .);
```

This will allow very convenient configuration, which could only deal with remote or only with local or both, in any order the admin wants it to be.

1.1.3 Dealing with new games being added and with game start notification

Most of this component’s functionality revolves around delegating to persistence, but we need to do two additional things that go beyond simple delegation. We will need two worker threads that do the following:

1. We will have a configurable (i.e. sleep interval) worker thread, which will periodically check if new games have been added to persistence. This is important from the perspective of up-to-date game data. This thread will simply wake up and get the list of currently supported games and will update the current list.
2. We will also have a configurable (i.e. sleep interval) worker thread, which will wake up periodically to see if a game should be started. This will work on the assumption that all games are sorted on their start times and this thread when it wakes up will simply go through the list and for any game that is past current time but is not started.

1.2 Design Patterns

The **Facade** pattern is used with the `GameDataManager` implementation classes. Implementation swill hide any persistence manipulations, any persistence lookup[and instantiation, as well as any notification or listener activity.
The **Observer/Observable** pattern is used with processing the worker threads for both the new game discovery and the start game arrival time.
Also the **Template Method** pattern is used with eth AbstractGameDataManager class that allows the user to place their own implementations of listeners etc…

1.3 Industry Standards

• EJB 2.1, J2EE 1.4

1.4 Required Algorithms

There are no complicated algorithms involved. Please consult the method documentation in Poseidon for further details.

1.5 Component Class Overview

1.5.1 com.orpheus.game

GameDataManager <<Interface>>
Defines the behavior of objects that manage Orpheus game data. This interface essentially sets with two aspects of the game data interaction. On the one hand we need to ensure that the public API is properly implemented (which is not more than testing upcoming domain for validity, persisting winning bids, and advancing hosting slots) and on the other hand we also have this contract specifying that we should be able to track the assigned start dates of all games that have not yet started.
It is expected that the implementation swill be thread-safe.

BaseGameDataManager <<abstract class>>
This is a base abstraction of the manager. It adds implementations of listener methods (the manager becomes listener) as well as direct game data manipulation for start game functionality (i.e. monitoring when a game starts, as well as monitoring when new games become available)
The added methods are thread-safe.
Implementations must be thread-safe.

GameDataManagerImpl <concrete class>>
This is an implementation of the base manager abstract class. It overrides the key three methods and uses either the local or remote ejb interface to enact persistence functionality. It provides persistence based mechanism for testing whether an upcoming domain is ready to begin hosting a specific slot, provides a mechanism for programmatically advancing to the next hosting slot, and finally it provides an implementation of functionality for recording the IDs of the winning bids for the slots of a specified hosting block.
It also defines two threads to act as poll thread workers to act as notifiers of when new game data becomes available or when a game is ready to be started.
Implementation is thread-safe.

JndiLookupDesignation <<Concrete class>>
This is an enumeration style class, which lists possible JNDI lookup designations. Currently we can only lookup local or remote interfaces.
This is thread-safe by its read-only virtue.

NewGameAvailableNotifier <<Concrete class>>
This is a simple class, which runs as a worker thread and checks, at specified intervals, if new game data is available on the server. This class is 'smart' enough to work with both local and remote ejb interfaces transparently.
This class accepts a registered listener, which gets notified whenever a new game has indeed been found on the server. This is achieved by keeping a cache of 'known' games, which gets updated whenever a new game is discovered. The memory consumption for this should not be very large since we are only caching ids and not the whole game data.
This is an internal worker thread class so it is guaranteed to be only executed by a single thread (i.e. this is created in the manager's constructor)

GameStartNotifier <<Concrete class>>
This is a simple class, which runs as a worker thread and checks, at specified intervals, if games (which have not yet started) should be started. This class accepts a registered listener, which gets notified whenever a game has been started.
This is an internal worker thread class so it is guaranteed to be only executed by a single thread (i.e. this is created in the manager's constructor) Note also that we do not have to worry about concurrency issues as far as the list of games is concerned since we are working on a copy so the original could actually be concurrently updated.

GameStartListener <<interface>>
This interface is an observer of a status change for games that go from not started to started.
Implementations do not have to be thread-safe since a single thread is guaranteed to trigger this notification.

NewGameAvailableListener <<interface>>
This interface is an observer of new games that have been added to the server. We only consider new games to be ones that have not been started yet.
Implementations do not have to be thread-safe since a single thread is guaranteed to trigger this notification.

GameDataUtility <<concrete class>>
This is a simple utility class, which will aggregate some useful functionality. The current functionality is basically a factory like method for easy creation of configured managers. It is based on ObjectFactory configuration.
Since there is not state this class is thread-safe.

1.6 Component Exception Definitions

1.6.1 Custom Exceptions

GameDataException
This is not really a custom exception created in this component but rather a required exception dictated by the RS API interface definition. This is essentially an exception indicating a failure to modify game data.
It will be invoked for the following issues:

1. When we try to persist (Read/Write) data from and to the persistence supporting EJB
2. When we try to lookup/create an EJB (local or remote) and we fail

GameDataManagerConfigurationException
This exception will be thrown if manager related configuration data is missing or incorrect, or if the actual mechanism for configuration reading fails.

1.6.2 System and general exceptions

In general this component will check for empty string input or null reference parameter input. It will also check of arrays have any null references or empty strings.

IllegalArgumentException:
This exception is used for invalid arguments. Invalid arguments in this design are empty strings provided as jndi lookup keys for example. This exception is also used

when checking in all classes for null arguments and arrays/lists with null elements. The exact details are documented for each method in its documentation.

1.7 Thread Safety

Implementation is thread-safe. It is quite obvious that the manager will be used by many concurrent threads and thus we must ensure proper thread-safety as well as some degree of atomicity of transactions.
Since the current implementation of the manager uses EJBs directly for its main functionality and since the container will ensure one thread per ejb we do not have to do anything special with reference to that.
On the other hand other functionality which involves updating start game information is achieved by locking/synchronizing on subsequent data structures (such as a `notYetStartedGames` list for proper synchronization of relevant data)

2. Environment Requirements

2.1 Environment

- Development language: Java1.4
- Compile target: Java1.4

2.2 TopCoder Software Components

- **Orpheus Game Persistence 1.0**
  Used to deal with persistence aspects that this component builds on.
- **Site Validator 1.0**
  Used to test subsequent domains (see requirement 4.3.4) to make sure we are dealing with a valid domain(s).
- **Object Factory 2.01**
  Used to implement a utility lookup for a manager creation. This is just a convenience to the user so that they do not have to implement Object Factory lookup functionality.
- **Configuration Manager 2.1.5**
  Used to store configuration parameters for this component.
- **Type Safe Enum 1.0**
  Used to create enumerations or this component.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.orpheus.game

3.2 Configuration Parameters

3.2.1 *GameDataManager configuration for Config manager*

Parameter	Description	Values
jndi_names	These are 1 or more jndi names to be used when looking up the ejb interface. These are comma delimited values, where the first element is the jndi name and the second element is a designation of either “Local” or “Remote” ***Required***	“jndiname , Local” to identify what type of lookup we are doing with the jndi name (Local)
new_game_poll_interval	Number of milliseconds to wait between polling the server for new games. ***Required***	60000
game_started_poll_interval	Number of milliseconds to wait between polling for starting games. ***Required***	1000

3.2.2 *GameDataManager configuration for ObjectFactory*

Parameter	Description	Values
game_Data_manager_key	This is a key that maps Object Factory configuration of a GameDataManager instance. Please consult the specific GameDataManagerImpl constructors for the type for data needed for configuration. ***Required***	“myManager”

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

None

4.3 Demo

Here we will demonstrate the general usage of the manager while assuming the following general configuration information:

The worker thread that checks the new game arrival is configured to look at every 1 min (60*1000 ms)
The worker thread that checks which games need to be started is configured to look at every 1 sec (1000 ms)

We will also assume that we have currently 3 games (not started yet) with eth following start times:

Game 1(10034)	1:00:00 PM
Game 2(10035)	1:01:00 PM
Game 3(10036)	1:01:00 PM

We will also assume that it is currently about 12:59:00

We will also assume that a local ejb interface is available and will have it configured under the JNDI lookup name of “localGamePersistenceJNDILookupName”
Finally we will assume that we have the proper ObjectFactory key configured (“games_manager”) so that we can use this key to instantiate the manager instance through OF configuration (here we assume that the “localGamePersistenceJNDILookupName” JNDI name is used with “Local” designation.

4.3.1 Creating the manager

```
// Directly
GameDataManager manager = new GameDataManagerImpl();
// through Object factory utility
GameDataManager manager =
               GameDataUtility.GetConfiguredGameDataManagerInstance(“games_manager”);
```

4.3.2 Record Winning Bids

```
manager.recordWinningBids(1000, new long[]{1, 2, 3, 4})
```

4.3.3 Advance Hosting Slot

```
manager.advanceHostingSlot(10034);
```

4.3.4 Test Upcoming Domain

```
boolean success = Manager.testUpcomingDomain(slot);
```

4.3.5 Assuming that some time has passed and game get started

```
// We should now see that the 10035 and the 10036 games have been started.
```

5. Future Enhancements

- None at this time