Game Persistence 1.0 Component Specification

1. Design

The Orpheus Game Persistence component provides the Orpheus application with an interface to persistent storage of game data. The central persistence functionality is handled by a stateless session EJB, but for interoperation with the Auction Framework component the bean is wrapped in an ordinary class.

1.1 Design considerations

There are no major considerations.

1.2 Design Patterns

1.2.1 Data Transfer Object J2EE Pattern

The SlotCompletion, Game, BallColor, HostingBlock, HostingSlot, DomainTarget, and ImageInfo are custom **Data Transfer Objects**, but their goal is not to improve performance but rather to provide serializability of the data.

1.2.2 Strategy

Used extensively in this design. GameDataDAO is part of this pattern. Even the SlotCompletion, Game, BallColor, HostingBlock, HostingSlot, DomainTarget, and ImageInfo are used as strategies by the clients, EJBs, and DAOs, although we do provide specific translators for them. PuzzleData, PuzzleResource, and DownloadData are external interfaces used as strategies by the above-mentioned classes. Needless to say, the EJB interfaces also use this pattern.

1.2.3 Factory pattern

The GameDataDAOFactory uses this pattern, to some degree, to provide a single entry point for EJBs to obtain the GameDataDAO instance. In a more orthodox usage of this pattern, the factory would accept a token and retrieve a type of DAO instance based on this token. This factory has a specific method to return DAO of a single type, and they don't manufacture new DAO instances with every call. In reality, the true factory here is the ObjectFactory class.

1.2.4 DAO

The GameDataDAO class uses the **DAO** pattern to encapsulate data access in one abstracted place.

1.2.5 Template method

The abstract client CustomDownloadSource implements the work of translating between objects, but leaves the details of interfacing with the persistence to the implementations using **Template Method** pattern.

1.3 Industry Standards

JDBC, SQL Server 2000 T-SQL, EJB 2.1

1.4 Required Algorithms

There are no complex algorithms here. Implementation hints are provided in Poseidon documentation as "Implementation Notes" sections.

1.5 Component Class Overview

CustomDownloadSource
This is the download source client to the EJB layer. It implements the DonwnloadSource. It is built to work with EJBs, and this class leaves it to implementations to specify the EJBs. This is purpose of the abstract ejbXXX methods. The public methods defer to these for actual persistence calls.

LocalCustomDownloadSource
Implements the abstract ejbXXX method to work with the local game data EJB. Simply defers all calls to the EJB. It uses the ConfigManager and Object Factory to initialize the JNDI EJB reference to obtain the handle to the EJB interface itself.

RemoteCustomDownloadSource
Implements the abstract ejbXXX method to work with the remote game data EJB. Simply defers all calls to the EJB. It uses the ConfigManager and Object Factory to initialize the JNDI EJB reference to obtain the handle to the EJB interface itself.

GameDataLocalHome
The local home interface of the GameData EJB.

GameDataHome
The remote home interface of the GameData EJB.

GameDataLocal
The local component interface of the GameData EJB, which provides access to persistent information about games managed by the application.

GameData
The remote component interface of the GameData EJB, which provides access to persistent information about games managed by the application.

GameDataDAOFactory
Static factory for supplying the game data DAO instance to the EJBs. It uses synchronized lazy instantiation to get the initial instance of the DAO. Supports the creation of the GameDataDAO.

GameDataDAO
Interface specifying the methods for Game Data persistence. Supports all methods in the ejb.

SlotCompletion
Represents the recorded data about a player's completion of a hosting slot.

BallColor

A BallColor object represents a supported Barooka Ball color.

Domain
An interface representing a hosting domain within the application

DomainTarget
Represents an object to be sought by players on a host site.

Game
The Game interface represents an individual game managed by the application. It carries a unique identifier, a start date, and an end date, and can provide a BallColor representing the color associated with this game and a game name string computed based on the game id and color.

HostingBlock
Represents a 'block'; of hosting slots. Blocks serve as an organizational unit for hosting auctions, and furthermore help to obscure the specific sequence of upcoming domains, even from sponsors privy to the auction details.

HostingSlot
An interface representing the persistent information about a particular hosting slot

ImageInfo
An interface representing the stored information about an image associated with a specific domain

GameDataBean
The EJB that handles the actual client requests. It accepts all client operations, but simply delegates all operations to the GameDataDAO it obtains from the GameDataDAOFactory.

SQLServerGameDataDAO
Implements GameDataDAO. Works with SQL Server database and most tables defined in RS 1.2.3. Most of the beans used in the methods have a mostly 1-1 correspondence with the tables. It uses ConfigManager and Object Factory to configure the connection factory, random string image, and the length of key text to generate as well as the media type. The last three are used in the recording of the host slot completion It is expected that the connection factory will use a JNDI connection provider so the Datasource is obtained from the application server.

BallColorImpl
Simple implementation of the BallColor. Represents a supported Barooka Ball color.

DomainImpl

Simple implementation of the Domain. Represents a hosting domain within the application.

DomainTargetImpl
Simple implementation of the DomainTarget. Represents an object to be sought by players on a host site.

DownloadDataImpl
Simple serializable implementation of the DownloadData. Represents a complete, downloadable entity with associated media type and (possibly) suggested file name.

GameImpl
Simple implementation of the Game. Represents an individual game managed by the application. It carries a unique identifier, a start date, and an end date, and can provide a BallColor representing the color associated with this game and a game name string computed based on the game id and color.

HostingBlockImpl
Simple implementation of the HostingBlock. Represents a 'block'; of hosting slots. Blocks serve as an organizational unit for hosting auctions, and furthermore help to obscure the specific sequence of upcoming domains, even from sponsors privy to the auction details.

HostingSlotImpl
Simple implementation of the HostingSlot. Represents the persistent information about a particular hosting slot

ImageInfoImpl
Simple implementation of the ImageInfo. Represents the stored information about an image associated with a specific domain.

SlotCompletionImpl
Simple implementation of the SlotCompletion. Represents the recorded data about a player's completion of a hosting slot.

1.6 Component Exception Definitions

This component defines six custom exceptions.

InstantiationException
Extends GameDataException. This exception is thrown by the constructors of most custom classes in this design that require configuration. The classes that use this exception include: LocalCustomDownloadSource, RemoteCustomDownloadSource, and SQLServerGameDataDAO. It is thrown if there is an error during the construction of these objects.

PersistenceException

Extends GameDataException. This exception is the base exception for persistence operations in this component. As such, it and its three subclasses are thrown by the ejbXXX method in the clients, the business methods in the EJBs, and the DAOs. In effect, the client helper method and EJB business methods act as a pass-though for these exceptions.

DuplicateEntryException
Extends PersistenceException. This is a specific persistence exception when inserting a record with a primary id that already exists. The DAO and the associated EJB and client helper methods use it.

EntryNotFoundException
Extends PersistenceException. This is a specific persistence exception when retrieving, updating, or deleting a record with a primary id that does not exist. The DAO and the associated EJB and client helper methods use it.

InvalidEntryException
Extends PersistenceException. This is a specific persistence exception when inserting a record with a primary id that is not valid, which excludes issues such as duplicate ids. The DAO and the associated EJB and client helper methods use it.

1.7 Thread Safety

Thread safety is an integral and required part of this component, and this component is indeed completely thread-safe.

Most classes are thread-safe because they are immutable. This includes the clients, DAOs, and beans. The EJBs are not thread safe, but the container assumes responsibility for this, and since these are stateless beans, they hold no state for us, so the status of their thread-safety is not a concern of ours. Transaction control, a related topic, is managed by the container.

The DAOFactory synchronizes its methods to avoid synchronization issues with lazy instantiation.

2. Environment Requirements

2.1 Environment

JDK 1.4, J2EE 1.4

2.2 TopCoder Software Components

- **Configuration Manager 2.1.5**
    - Used for configuration in constructors throughout this component to instantiate required classes and obtain other parameters.
- **Object Factory 2.0**

- Used to instantiate classes in constructors throughout this component to instantiate required classes.

- **Random String Image 1.0**
  - Provides the image generation capability for slot completion.
- **Puzzle Framework 1.0**
  - Provides the PuzzleData and PuzzleResource interface.
- **Front Controller 2.1**
  - Provides the download source and data interfaces that are implemented here.
- **Base Exception 1.0**
  - Provides the BaseException class here.
- **Command Line Utility 1.0**
  - Refer by random String image 1.0
- **Spell check 1.0**
  - Refer by random String image 1.0.
- **DBConnection Factory 1.0**
  - Provides a convenient access to Connections from a Datasource obtained from JNDI. The implementing DAO classes thus don't have to code their own access to this container resource.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

There are no third party components that need to be used directly by this component.

3. Installation and Configuration

3.1 Package Names

com.orpheus.game.persistence
com.orpheus.game.persistence.entities
com.orpheus.game.persistence.ejb
com.orpheus.game.persistence.dao

3.2 Configuration Parameters

3.2.1 CustomDownloadSource subclasses

Parameter	Description	Details
jndiEjbReference	The JNDI reference for the EJB. Required	Example: “java:comp/env/ejb/GameDataLocal”

3.2.2 DAOFactory

Parameter	Description	Sample Values
specNamespace	Namespace to use with the ConfigManagerSpecificationFactory. Required	Example: “com.topcoder.specify”
gameDataDAO	Key for the GameDataDAO to pass to ObjectFactory. Required	Valid key

3.2.3 SQLServerGameDataDAO

Parameter	Description	Sample Values
specNamespace	Namespace to use with the ConfigManagerSpecificationFactory. Required	Example: “com.topcoder.specify”
factoryKey	Key for the DB Connection Factory to pass to ObjectFactory. Required.	Valid key
name	Name of the connection to the persistence to get from the DB Connection Factory. Optional.	“myConnection” Will use the factory’s default connection, if available, if name not given.
randomStringImageKey	Key for the RandomStringImage to pass to ObjectFactory. Required.	Valid key
keyLength	Length of text to generate. A positive number. Required	3
mediaType	Non null/empty media type. Required	“text/plain”

3.3 Dependencies Configuration

3.3.1 DBConnectionFactory, ConfigManager, ObjectFactory

The developer should refer to the component specification of these components to configure them. The later two are used extensively.

3.3.2 DDL for tables

Please see sections 1.2.3 and 1.2.5 of the Requirements Specification.

3.3.3 EJB deployment descriptor

This is provided in the ejb-jar.xml file in the /docs/mappings directory.

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

Both client and EJBs must be deployed in a container that supports EJB 2.1 standard. The client can be deployed in a thinner environment as long as it has access to the necessary J2EE packages and has access to the Home and Remote interfaces and implementations in the classpath. If using local access, it must be collocated with the EJBs. Overall, this just corresponds to standard EJB usage.

The tables defined in the Requirements Specification must exist, and the database must be running. The ObjectFactory and ConfigManager components must be configured properly.

4.3 Demo

See 4.2 above on the required steps to set up the environment. The first part of the demo will only focus on the one public API, which is the custom download source. We will also show the GameData EJBs in action, as this component does present them as the public API for game data manipulation.

In both cases, we will only focus on the remote implementations, with the knowledge that usage of the local implementation is the same. We will show method calls directly.

4.3.1 Typical DownloadSource client usage

```
// create remote instance with a namespace
CustomDownloadSource client = new RemoteCustomDownloadSource
("myNamespace");

// we are requesting some download data
String id = valid download id
DownloadData data = client.getDownloadData(id);
```

4.3.2 *Typical gameData usage*

This demo shows usage of the EJB. We will demonstrate the use of the bean via its remote interface.

```
        //obtain remote interface: the details are
omitted.
        GameData gameDataAdmin = getRemoteEJB();

        //we might begin by creating some games,
        //first creating the block that the game use
        HostingBlock[] blocks =
TestHelper.getBlocks();
        Game game =
TestHelper.getGameImplInstance((BallColor)
colors.get(0), keyCount, gameStartDate, blocks);

        //create game now
        Game createdGame =
gameDataAdmin.createGame(game);

        //at this point, the game is created and id
was assigned
        //we create some slots for a block and bids
        long blockId =
createdGame.getBlocks()[0].getId().longValue();

        //now create a domain
        ImageInfo[] images =
TestHelper.getImages((BallColor[]) colors.toArray(new
BallColor[0]));
        Domain toCreate =
TestHelper.getDomain(sponsorId, "domainName", new
Boolean(true), images);
        Domain createdDomain =
gameDataAdmin.createDomain(toCreate);

        //get the already existed createdDomain's
image id
        long imageId =
createdDomain.getImages()[0].getId().longValue();

        //persist a bid record for testing
        long[] bidIds = TestHelper.persistBid(blockId,
imageId, 1, new Date());

        //create slots
```

```
        HostingSlot[] createdSlots =
gameDataAdmin.createSlots(blockId, bidIds);

        //we create a hosting block
        int slotMaxHostingTime = 1000;

gameDataAdmin.addBlock(createdGame.getId().longValue()
, 1000);

        //the download id that can be used
        long downloadId = ((BallColor)
this.colors.get(0)).getImageId();

        //persist the puzzle
        long puzzleId =
TestHelper.persistPuzzle(downloadId, "puzzleName");

        //persist the slot for testing.//TODO, it
would be removed as it should be created in the
createSlot method
        int sequenceNumber = 1;
        Date hostingStart = new Date();
        long slotId =
TestHelper.persistSlot(bidIds[0], sequenceNumber,
hostingStart, puzzleId, downloadId);

        //we get a game
        Game getGame =
gameDataAdmin.getGame(createdGame.getId().longValue())
;
        //we get a block
        blockId =
getGame.getBlocks()[0].getId().longValue();

        HostingBlock block =
gameDataAdmin.getBlock(blockId);

        //we get a slot
        HostingSlot slot =
gameDataAdmin.getSlot(slotId);

        //we get a download object
        DownloadData data =
gameDataAdmin.getDownloadData(downloadId);

        //we get a domain
```

```
        long domainId =
createdDomain.getId().longValue();
        Domain domain =
gameDataAdmin.getDomain(domainId);

        //we get key texts for a player
        long[] slotIds = new long[] { slotId };
        String[] keys =
gameDataAdmin.getKeysForPlayer(playerId, slotIds);

        //we get puzzle data
        PuzzleData puzzle =
gameDataAdmin.getPuzzle(puzzleId);

        //persist the download

TestHelper.persistPluginDownload("pluginName");
        //we record a plugin download count increase
        String pluginName = "pluginName";

gameDataAdmin.recordPluginDownload(pluginName);

        //we record a plugin download count increase

gameDataAdmin.recordPluginDownload(pluginName);

        //we record player registration in a game
        long gameId = createdGame.getId().longValue();
        gameDataAdmin.recordRegistration(playerId,
gameId);

        // we record slot completion for a player
        Date date = new Date();
        gameDataAdmin.recordSlotCompletion(playerId,
slotId, date);

        //we record game completion by a player
        gameDataAdmin.recordGameCompletion(playerId,
gameId);

        //we add a new binary object in the database
        String name = "name";
        String mediaType = "image/png";
        byte[] content = "content".getBytes();
        gameDataAdmin.recordBinaryObject(name,
mediaType, content);
```

```
        //update slots
        DomainTarget[] targets =
slot.getDomainTargets();
        targets[0] = new DomainTargetImpl(null,
targets[0].getSequenceNumber(),
targets[0].getUriPath(),
                targets[0].getIdentifierText(),
targets[0].getIdentifierHash(),
targets[0].getClueImageId());

        HostingSlot toUpdate = new
HostingSlotImpl(slot.getId(), slot.getDomain(),
slot.getImageId(),
                slot.getBrainTeaserIds(),
slot.getPuzzleId(), slot.getSequenceNumber() + 1,
targets,
                slot.getWinningBid(),
slot.getHostingStart(), slot.getHostingEnd());

        HostingSlot[] slots = new HostingSlot[] {
toUpdate };

        //update the slot in ejb
        HostingSlot[] updatedSlots =
gameDataAdmin.updateSlots(slots);

        //update domain
        Domain toUpdatedDomain = new
DomainImpl(createdDomain.getId(), sponsorId,
"domainName", new Boolean(false),
                images);
        gameDataAdmin.updateDomain(domain);

        //find active domains
        Domain[] domains =
gameDataAdmin.findActiveDomains();

        // find games for player in domain
        Game[] games =
gameDataAdmin.findGamesByDomain("domainName",
playerId);

        //find completed slots
        HostingSlot[] completedSlots =
gameDataAdmin.findCompletedSlots(gameId);

        //find slot completions for game and slot
```

```
        SlotCompletion[] slotCompletions =
gameDataAdmin.findSlotCompletions(gameId, slotId);

        //find games
        Boolean isStarted = new Boolean(true);
        Boolean isEnded = null;
        Game[] findGames =
gameDataAdmin.findGames(isStarted, isEnded);

        //find games that the player is registered in
        long[] gameIds =
gameDataAdmin.findGameRegistrations(playerId);

        //find a sponsor's domains
        Domain[] sponsorDomain =
gameDataAdmin.findDomainsForSponsor(sponsorId);

        //find first slot this player did not finish
        gameDataAdmin.findSlotForDomain(gameId,
playerId, "domainName");

        //get all ball colors
        BallColor[] colors =
gameDataAdmin.findAllBallColors();
```

5. Future Enhancements

None at this time.