JDNC DataBinding Architecture Whitepaper
Part I

Preamble
This document is the first rough draft, so any comments are welcome!

Introduction
The goal of the JDNC project is to significantly reduce the effort and expertise required to build rich, data-centric, Java desktop clients. These clients are representative of what enterprise developers typically build, such as SQL database frontends, forms-based workflow, data visualization applications, J2EE-based network services, Webservices, and the like.

JDNC leverages the power of J2SE and Swing while providing a higher level API, as well as an optional XML markup language, which enables common user-interface functionality to be constructed more quickly, without requiring extensive Swing or GUI programming skill. Additionally, JDNC simplifies the task of connecting a rich client to a J2EE backend, including JDBC and WebServices.

It is the purpose of this document to describe the design, architecture, goals, and rationale behind the JDNC DataSet API. After reading this paper, you should have a thourough knowledge of how to use the JDNC DataSet classes to connect and manage the backend portion of the data binding architecture. For a complete analysis and description of the frontend portion (ie: how to bind GUI components to the classes described in this document), see the second paper in this two part series.

This paper begins by describing some common use cases. These will be used throughout the paper as rationale behind various design decisions. The intent is that they will also be useful in helping developers understand how to get their own applications running with JDNC.

Use Cases
The Adventure Builder blueprints application for WebServices will be used as the use case for this paper. AdventureBuilder is a fictional application that contains information on various vacation packages such as “Maui Survival Adventure” and “Mountain Climbing Adventure”.

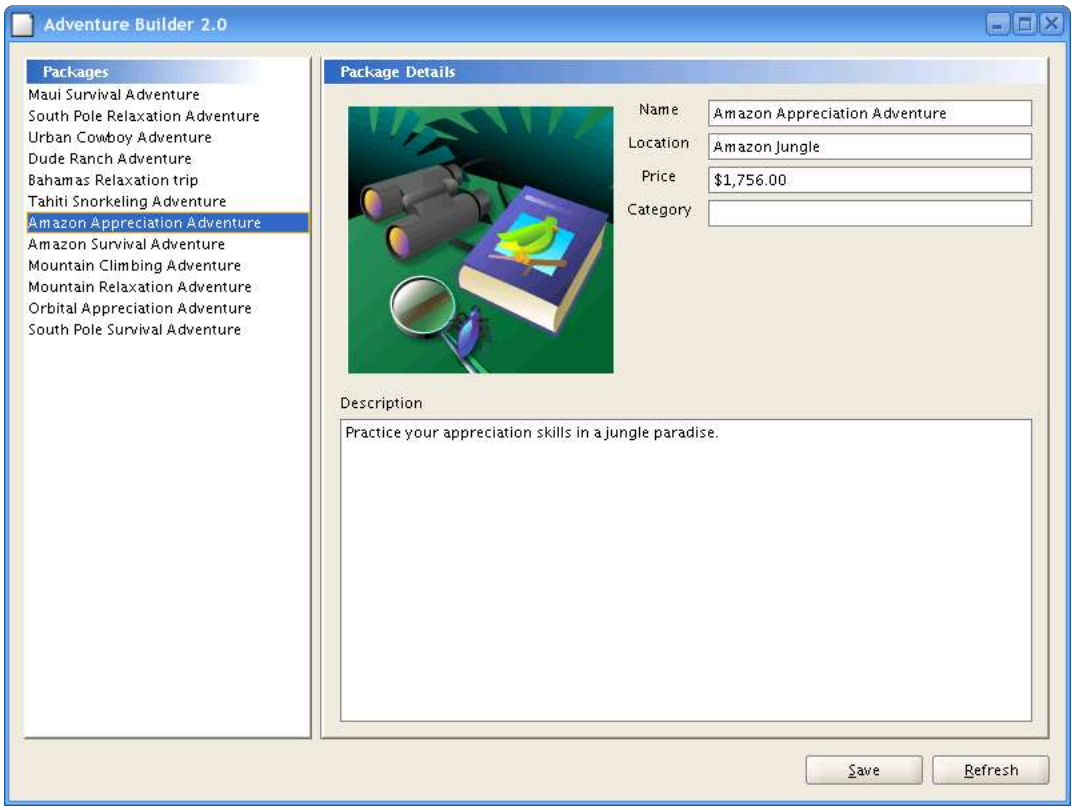
The app is comprised of the following tables:

Table Name	Description	Sample
Category	Categories for the types of available adventures	“Island Adventures”,“Mountain Adventures”
Lodging	Available Hotels/lodging for the various adventures	“Budget Hotel”, “Jungle Tent Hotel”, “Moutain Cave Hotel”
Package	Adventure Packages. Aggregate of category, lodging, price	“Maui Survival Adventure”
Activity	Some activity available on an adventure	“Snorkeling”, “Mountain Biking”
ActivityList	List of activities available on each Adventure	“Snorkeling” and “Surfing” could be for the “Maui Survival Adventure”
Transportation	Transportation options. Related only superficially to destination	“Mountain Airlines, Airbus A320”
SignOn	Login information for app user	
Account	Customer account	

Use Case #1: EJB Frontend
3 tiered
EJB middle tier
Wants a rich client that can be installed on the clients computer. This client needs to interact seemlessly with the EJB tier.

Use Case #2: Direct Database access
Adventure Inc. corporate needs to design and deploy several applications in house for reporting, accounting, and management purposes. These apps connect directly to the database, bypassing the web tier to reduce needless resource contention. These apps are behind the corporate firewall.

Use Case #3: WebService Frontend
This is an alternative to Use Case #1. Rather than accessing the EJB's directly, they will be hidden behind a webservice facade, and accessed via this facade.



The Big Picture

There are a series of common problems that all client side applications face. One of these problems is data binding. Data binding is a term used to describe automatic coupling between a user interface component and some data in the application domain. An example could be *binding* a JTextField to the "firstName" field of a "Customer" object. Such a binding would automatically load the JTextField with the value of the firstName field from the Customer object. Changes made to the value of this JTextField would subsequently be pushed back into the firstName field.

Traditionally, developers of Swing clients have had to write the data binding code by hand. The JDNC DataBinding architecture is designed to relieve developers of this repetitive boilerplate coding.

In addition, JDNC provides a DataSet API that is used in conjunction with the DataBinding API to simplify the task of connecting and binding to tabular data stores, such as SQL databases. In addition, the DataSet API can be used with WebServices, EJBs, or other JavaBean oriented technologies. One benefit of having a unified architecture is that it allows the developer to migrate from one data store to another without changing much code. However, it is possible that the DataSet API will prove to be most useful when used with SQL databases, while other architectures might be more useful with WebServices, EJB's, etc.

Master/Detail

One of the key scenarios at the heart of many of the design decisions is Master/Detail. Master/Detail is an arrangement where the “current” record(s) in the Master produces a specific set of “Detail” records. This very naturally maps to the behavior of a database.

For example, our Adventure database contains the tables “package” and “category”. The package table contains all of the travel packages in the database, and the category table contains all of the categories in the database. Each record in the package table contains a link (catid) referencing the category to which the package belongs. For example, the record in the package table corrosponding to the “Maui Survival Adventure” contains a reference to the category record named “Island Adventure”.

A sample query that returns all of the packages in the database would be “select * from package”. If instead I had a specific package with an id of 'PACK-1' that I was interested in, I would use “select * from package where packageid = 'PACK-1'”. If I wanted the category for this package, I'd use “select * from category where catid='ISLAND'”.

This is the basis behind master/detail in an application. With a database backed DataSet, you have to set up relationships between tables like Package and Category as a master/detail. If I wanted a JList to display the packages, but a JtextField to display the category name for the selected package, then every time a package is selected I need to refresh the category DataSet and reload the JTextField.

Where does the logic go that ties these two user interface components together? Where does one put their database queries? How does the detail know what the current row is in the master JList? What does it mean to “know what the current row is”?

The JDNC DataBinding architecture provides a mechanism for handling the master/detail relationships in a natural manner. See the description of the DataSet below for a complete description of how the DataSet implements this functionality.

DataStoreConnection
The DataStoreConnection abstracts the connection with a backend data store from the rest of the DataSet API. Example backend data stores include an EJB middle tier, an RDBMS, a WebService, or even a File System. The DataStoreConnection also handles background threading tasks necessary for maintaining a responsive GUI.

The DataStoreConnection maintains a “connected” property. Changes to this property can be observed via a PropertyChangeListener. DataSets register a listener on this property. When a DataStoreConnection's “connected” property changes from “false” to “true”, the DataSet will refresh itself. When the property changes from “true” to “false”, then the DataSet flushes its data.

DataSet
The DataSet interface represents a tabular set of data, much like the Swing TableModel. It could be backed by a collection of JavaBeans, a RowSet, or some other custom data structure that can be mapped to two dimensions. The DataSet contains methods for participating in Master/Detail relationships. Because it implements the DataModel interface, it can be bound to GUI components. The DataSet also has a selection model,

allowing more than one record in the DataSet to be “current” at the same time.

The DataSet also contains MetaData associated with each field. The MetaData describes aspects of the field, such as it's name, type, and edit constraints. This class my be used when such information isn't encapsulated in the DataSet itself and thus must be represented in an external object. MetaData is intended only for holding state about a DataSet field element and is not intended for implementing application semantics.

MetaData can contain custom properties as well. These properties are based on the traditional key/value paradygm. An example of useful custom properties might be hints for GUI rendering. Keys in this map should be of the form:

com.mydomain.packagename.PropertyName

Master
The Master interface contains the methods necessary to describe the Master portion of a Master/Detail relationship. While DataSet extends Master, it is possible that some custom third party component could also implement the Master interface, and participate in a Master/Detail relationship.

Detail

Task
A Task is something that you want to execute asynchronously on a background thread, using the DataStoreConnection. Usually these are things that are invoked from the UI (such as saving customer data, or refreshing, or starting a streaming video, or some such thing).

RDBM systems generally support several types of operations including stored procedures, Data Manipulation Language SQL statements, and Data Definition Language SQL statements. These operations can more generally be broken down into two different types: operations that return data, and operations that don't.

For example, delete, update, and insert SQL statements generally return a count indicating the number of records affected. Select SQL statements return some set of results.

In JDNC, there are two different types of Tasks corrosponding to these two different types of SQL operations. The first is the QueryTask, the second is the ProcedureTask. As will be seen, this paradygm works in the world of JavaBeans as well.

The QueryTask is used for fetching information from the data store and loading a DataSet with that data. A QueryTask knows how to get data, and how to update the data store with any changes to that data. In the simplist case, a JDBCQueryTask might ask for the table name. It would then return the set of all records in the table. It could automatically generate the update, delete and insert SQL statements based on the metadata for the query.

Each DataSet contains a QueryTask property named “query”. In this way, the DataSet knows how to refresh itself, if necessary, and how to persist changes to the data store. It simply asks its DataStoreConnection to execute the given QueryTask.

ProcedureTasks can be defined and used from any module. For example, some ActionListener code might call a stored procedure on the RDBMS whenever the action is fired. This work would be done by calling the DataStoreConnection, and would happen on a background thread.