Software Tree
Simplify Data Integration

N J D X 1.5
.NET Jet Database Exchange

USER MANUAL

.NET Jet Database Exchange (NJDX^{TM})

Version 1.5

NJDX User Manual

Software Tree, Inc.
2953 Bunker Hill Lane, Suite 400
Santa Clara, CA 95054
Phone: 408-282-3606
Fax: 408-282-3501
http://www.softwaretree.com

Contents

Introduction

.NET Jet Database Exchange (NJDX) technology provides a software infrastructure for bridging the gap between object-oriented .NET programs and relational SQL databases. NJDX is based on JDX, a flexible and market-proven object-relational mapping technology for Java/J2EE world. NJDX inherits all the great features of JDX and blends seamlessly into the .NET infrastructure including all CLR based languages (e.g., Visual C# .NET, Visual Basic .NET, Visual J# .NET). In this manual, NJDX and JDX may be used interchangeably.

OOP and Business Applications

Object-oriented programming (OOP) has become the dominant programming paradigm. In OOP languages (e.g. C#, Java, C++), a class encapsulates the structure and behavior of objects of a certain type. Business (domain model) objects are easier to represent as instances of classes. Examples of business objects are: Quotations, Purchase orders, Customer info, Invoices, Patient records, Drug info, Configuration info, Product info, Pricing info, Sales transactions, etc. The applications that create and manipulate these business objects can come and go. So, one common need for virtually all business applications is the persistence (stable storage) of business objects. The options for persistent storage are file systems, object-oriented databases and relational SQL databases (RDBMS). For simple, non-mission critical applications requiring low data volume, a file system solution may be sufficient. However, for high-performance, scalable, transactional and, robust applications, a database management system is required for storing business objects. A study has shown that, in spite of recent advancements in object-oriented databases, only a small percentage of the corporate data resides in these kinds of databases. The rest resides in the trusted relational databases. Corporations feel comfortable with the maturity of the RDBMS technology and the availability of numerous third-party tools for analyzing and managing the data in RDBMS. It is only natural that application and tools developers would like to represent the business objects in object-oriented languages (like C# and Java) and at the same time use relational databases for the persistence of those objects.

Since there is already a lot of data residing in relational databases, it is also very important that the existing data may be used in new object-oriented applications.

Using Relational Databases for Business Objects

Data is stored in rows of tables in a relational database. All rows of a table have a fixed number of typed columns. In other words, a table is a collection of the same kind of rows. There is uniformity to the way data is stored in relational databases. However, data in class objects may not be represented in a flat structured way. In general, an object may have complex structure such that an attribute of an object may reference an object (or a collection of objects) of another kind. Further, a referenced object may reference another set of objects. Now some of these referenced objects may really be integral to the containing object (for example, a purchase order may contain multiple line items). When a complex object needs to be made persistent in a relational database, different components of the object may need to be put in different tables because each table can store only one kind of row (object). It is also possible to independently store a component of an object but later on that component should be retrievable as part of the containing object. There is an inherent paradigm-mismatch between an object-oriented model and relational model. However, there is a genuine need to use relational databases to store business objects. How to bridge this gap?

First need is to easily define the mapping between the object model and the relational model. The second need is to store this mapping information in such a way that it can be used most naturally and conveniently. Thirdly, there is need for a product to understand this mapping and do appropriate translation between the object and relational data. That also includes the functionality of generating the relational schema definition with the given class definitions and mapping information. The product should also be able to work with existing schemas and be able to reverse-engineer object models and mapping information. And finally, there is need for defining an intuitive application-programming interface (API), which will make the task of a programmer easier by relieving him of the burden of generating low-level SQL statements to get and store the object data using the relational database. Software Tree has created NJDX product to meet these needs.

Why NJDX?

Simply put, NJDX bridges the gap between the worlds of objects and relations with a powerful and flexible solution. Adhering to some well thought-out **KISS** (Keep It Simple and Straightforward) principles, NJDX provides smooth integration with popular databases including Microsoft SQL Server, Oracle, IBM DB2, and Microsoft Access.

The KISS principles (please click here for more details) that have served as the guiding philosophy behind devlopment of NJDX include:

- Solve the most important problem in the simplest possible way
- Don't make the solution more complex than the original problem
- Be completely non-intrusive to the object model
- Make it easy to define, modify, comprehend, and share the mapping specification
- Avoid source code generation for data access
- No mind reading
- Expose small number of simple and consistent APIs
- Optimize data access logic automatically
- Keep the internal implementation simple, extensible, and efficient
- Offer intuitive tools to deal with object models, database schema, and mapping
- Provide a straightforward installer, lucid documentation, and readymade examples

Some of the salient features of NJDX include:

- Defining mapping between an object model and a relational model using declarative specification.
- An easy-to-use GUI tool (NJDXDemo) to simplify the process of mapping specification and verification.
- A small yet powerful set of APIs that can effectively be used by application programmers to meet their object persistence needs using an RDBMS.
- A non-intrusive programming model, which will not change your .NET code in any way (pre-processing or post-processing).
- The use of a highly optimized metadata-driven object-relational mapping engine that is lightweight, dynamic, and flexible.
- High-performance object caching
- Support for persistence for business objects defined in any CLR based language including Visual C# .NET, Visual Basic .NET, and Visual J# .NET.
- Seamless integration with Visual Studio .NET 2003 and Visual Studio 2005 to allow developers to easily perform tasks such as defining object-relational mapping, creating database schema, reverse-engineering C# classes from an existing database schema, and verifying OR-Mapping specification against live data

There are 3 simple steps to use NJDX:

1. Define business objects (classes in any CLR language)
2. Define object-relational mapping
3. Develop applications using intuitive and powerful NJDX API

The following chapters explain the Architecture, Concepts, Object-Relational Mapping grammar, Schema Generator, NJDXStudio, and APIs in more detail along with several examples and usage tips. The software comes with extensive documentation and tutorials.

Before getting into formal details of the product, the following examples are provided to give you a quick feel of the power and ease of using NJDX.

A Simple Example

Assume two C# business classes - SimpleDept and SimpleEmp. The deptId_ attribute of SimpleEmp identifies the related SimpleDept object through the attribute dept_.

```
public class SimpleDept {
   public int deptId_;
   public String deptName_;

   public SimpleDept () {
   }
   ...
}
```

```
public class SimpleEmp {
   public String empId_;
   public int deptId_;
   public String empName_;
   public String title_;
   public String ssn_;
   public float salary_;
   public System.DateTime hireDate_;
   public SimpleDept dept_;

   public SimpleEmp () {
   }
   ...
}
```

The object-relational mapping for the above object model can easily be described in the following declarative way:

```
CLASS SimpleDept TABLE Simple_Dept
   PRIMARY_KEY deptId_
;
CLASS SimpleEmp TABLE Simple_Employee
   PRIMARY_KEY empId_
   RELATIONSHIP dept_ REFERENCES SimpleDept WITH deptId_
;
```

Notice that the table names may be different from the class names. A column name can optionally be changed from the default of the corresponding attribute name to any other name. **Chapter 5: Object-Relational Mapping Specification gives** more details on the mapping specification.

Here is an example of invoking insert() API to store an instance (emp) of class SimpleEmp using NJDX subsystem identified by a handle, jdx1:

```
jdx1.insert(emp, 0, null);
```

Notice that the application programmer is shielded from low-level SQL statements. NJDX automatically generates the required SQL INSERT statement at runtime to persist the object in the appropriate database table.

Here is an example of retrieving all employees belonging to department 100:

```
int deptId = 100;
String predicate = "deptId_ =" + deptId;
ArrayList queryResults = jdx1.query("SimpleEmp", predicate,
                                    JDXS.ALL, 0, null);
```

Notice that the application programmer makes an object-oriented call to get the employees of department 100. NJDX generates the required SQL SELECT statements to fetch the appropriate rows, builds the business objects and returns them to the application.

Essentially, NJDX presents an object-oriented view of the relational data through an intuitive and powerful interface. The details of the interface are in the chapter **Application Programming Interface (API).**

NJDX can also generate relational schema (including table definitions) from an object model and generate C# classes from an existing relational schema. Here is an example of a command to create a database schema for the given object-relational mapping specification in a file abc.jdx

```
NJDXSchema -create abc.jdx
```

More details on NJDX schema generation and reverse-engineering tools can be found in the chapter **Schema Generator**.

NJDXStudio

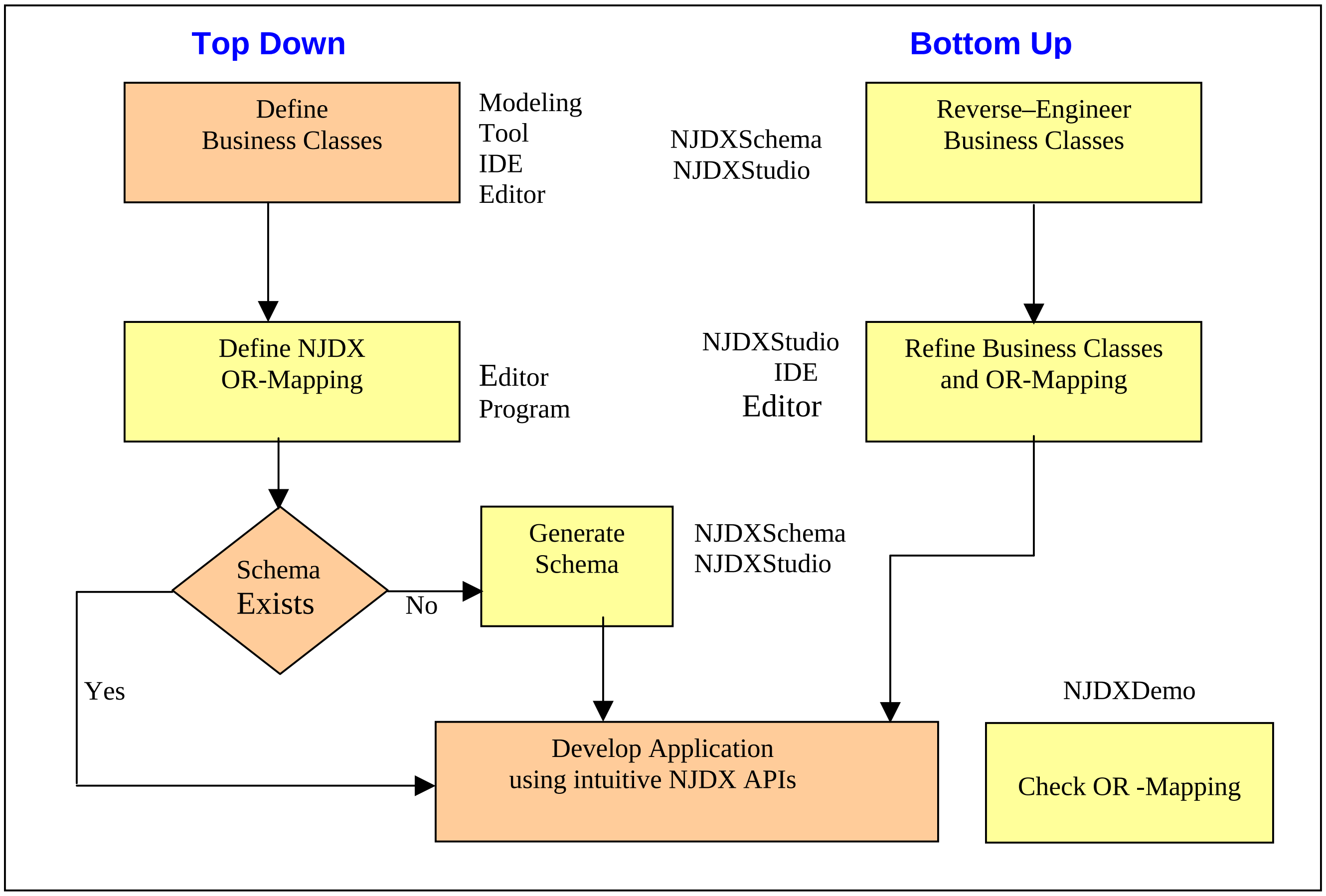
To further simplify the life of an application programmer, NJDX add-in(NJDXStudio) is provided which tightly integrates with Visual Studio .NET 2003 and Visual Studio 2005. NJDXStudio allow developers to achieve the following:

- Reverse-engineer C# classes from an existing database schema
- Create database schema from an existing object model
- Define object-relational mapping
- Verify OR-Mapping against a live database

NJDXStudio provides an intiutive menu and a toolbar to launch various OR-Mapping related activities, from within the Visual Studio. Extensive help in the form of tooltips and quick help files are available for each activity. For more information related to NJDXStudio, see the chapter titled “NJDXStudio: A Visual Studio .NET Add-In.”

The following flow diagram summarizes different steps needed to develop a .NET application using NJDX. The flow diagram also enlists typical tools, which may be used at different stages of the application development.

Development Steps with NJDX



The TopDown steps are taken when the application development starts with a fresh object model. The database schema may or may not exist in this case.

The BottomUp steps are taken when a new application needs to be developed using an existing schema and the developer wants to get a jump-start by creating an object-model based on the existing schema.

The NJDXDemo utility program provides a GUI to quickly check the OR-Mapping specification against a live database. NJDXDemo can also be invoked from within Visual Studio. NJDXDemo is explained in more detail later in the manual.

The following pages provide a summary of the important highlights of the NJDX product.

NJDX™ OR-Mapper Highlights

Object-relational mapping is one of the most complex issues to address in modern application architecture. NJDX provides a simple, practical and robust solution to this important problem. NJDX solves the tough problem of persistence for .NET applications by eliminating endless lines of ADO.NET/OleDB/SQL code.

You don't need to get bogged down with overly complex methodologies and over-arching frameworks. NJDX gives you the control you need to be an effective developer, helping you create more flexible and higher quality applications faster.

NJDX highlights

Simple, Non-intrusive, and Flexible Design

Smart and Elegant Mapping Specification

Support for Complex Object Modeling including Class Hierarchies

Small Set of Simple and Flexible APIs

Lightweight and Optimized Mapping Engine

Object Caching

High-performance and Scalable Implementation

Powerful and Intuitive GUI Tools (NJDXStudio and NJDXDemo)

Nifty Components and Facilities to Simplify Development

Works with Most Popular Databases, Existing Schema, and Application Servers

Easy-to-learn and Easy-to-use

Robust and Market-Proven Technology

Quick ROI

- **Simple, Non-intrusive, and Flexible Design**
  - Simple concepts; easy development steps; quick learning curve
  - No need to inherit from a base class or implement any special interfaces
  - No static generation and maintenance of large amounts of messy code; dynamic mapping engine
  - Easy evolution of object and relational models
  - Flexible usage in any tier of an application – be it standalone, or ASP.NET based

- **Smart and Elegant Mapping Specification**
  - Declarative mapping specifications based on simple grammar
  - Human-readable and easily comprehensible; no need to struggle with complex XML files
  - Compact; most default mapping is automatically deduced; avoids verbosity
  - Normalized and modular specification; no repetition of the same information
  - Most mapping primitives are orthogonal to each other, avoiding unnecessary tight coupling and enabling easy evolution
  - Allows cross-referencing of classes and collections no matter in which order their mappings have been defined
  - Intuitive and flexible ways of mapping complex object structures
  - Mappings for collections and relationships are defined at the object level, not at the relational level; makes it easier to understand and modify such mappings
  - A mapping specification can optionally be stored in and accessed from the database

- **Support for Complex Object Modeling including Class Hierarchies**
  - Associative and aggregated relationships
  - Persistence-by-reachability
  - 1-to-1, 1-to-many, and many-to-many relationships
  - Multiple options to store class-hierarchy objects
  - Polymorphic queries

- **Small Set of Simple and Flexible APIs**
  - Flexible query options – deep, shallow, and anything in-between. Supports loading of partial objects, sophisticated query predicates (including path-expressions), named queries, and powerful object-streaming functionality.
  - Aggregated operations
  - Flexible APIs for Stored Procedures and
  - Dynamic data routing

- **Lightweight and Optimized Mapping Engine**
  - Connection pooling
  - Prepared statements
  - Optimized SQL statements
  - Minimal database trips
  - Caching of metadata

- **Object Caching**
  - Caching options at individual class level to improve performance
  - Regular and LRU caches

- **High-performance and Scalable Implementation**
  - Short code paths
  - Optimistic concurrency control

- **Powerful and Intuitive GUI Tool (NJDXDemo and NJDXStudio)**
    - Excellent packaging of all the needed functionality to simplify mapping configuration (defining and verifying OR-Mapping, creating and reverse-engineering database schema)
    - Extensive online help available at every step of the way
    - Prototype queries without writing any code
    - Verify OR-Mapping against a live database

- **Nifty Components and Facilities to Simplify Development**
    - Utility components for pooling NJDX handles
    - Persistently unique sequence generators
    - Object-viewing facilities
    - Interactive development of queries using live data
    - Support for instance callback methods

- **Works with Most Popular Databases and Existing Schema**
    - Supports Oracle, SQL Server, DB2, Access, and any OleDB data sources
    - Easily reverse-engineer an object model from any existing relational schema
    - Mapping-in-the-middle for existing object models and schema including stored procedures
    - Supports Trusted Connections
    - APIs for SQL bypass
    - Works with ASP.NET and standalone programs

- **Easy-to-learn and Easy-to-use**
    - Clean design – not an over-engineered framework with complex semantics
    - Small Set of Simple and Flexible APIs
    - No learning of a new query language required
    - Powerful and intuitive GUI tool
    - Extensive documentation (Comprehensive user manual, API docs, tutorials, and online help)
    - Many working examples with easy-to-configure NANT scripts
    - Meaningful error and debug messages to diagnose any problem
    - Optional logging of all SQL statements

- **Robust and Market-Proven Mapping Engine**
    - Underlying mapping engine has been progressively enhanced based on feedback from thousands of users over last 8 years
    - Proven enterprise-class technology being used in real-world applications
    - Great customer testimonials

- **Quick ROI**
    - More modular and better performing applications
    - Increased programmer productivity
    - Reduced risks and faster time-to-market