developerWorks

LDD Today

Lotus. software

Level: Advanced
Works with: Notes/Domino
Updated: 02-Sep-2003

by
Kai-Hendrik
Komp

If you are developing Domino applications with Domino Java classes, a recommended practice is to encapsulate technical Domino database implementation details in a set of common base classes and to build a hierarchy of business domain-driven Java classes. Borrowing some well-known ideas like the create and find methods from Enterprise JavaBeans (EJB) technology can simplify the code used to handle business objects. This article gives you a simple example of a small class hierarchy to illustrate this concept. This article is intended for experienced Java developers familiar with the Domino Java API.

Introduction

The documents in a Domino database represent different objects in a business process. The data in these documents may be about customers, employees, products, purchase orders, delivery reports, billing information, or other business objects that are part of a business logic. In a business workflow, the code can be differentiated between technical infrastructure code—such as accessing a Notes database, accessing a Notes document via a view, and accessing the different items in a Notes document—and the code implementing the business logic—such as accessing all orders delivered to a specific customer during the last ten days.

To keep the business logic code clear and easy to maintain, do not complicate it with too much technical code. The best way to do this is to encapsulate the Domino specific implementation details in a small set of common base classes. This object-oriented approach is the preferred way to go, particularly if you are not completely familiar with the Domino Java classes or if you work on projects in which Domino is one part of a Java-based infrastructure. Keeping in mind this design goal and borrowing the idea of create, find, and other methods from the EJB technology to build a hierarchy of business domain-driven Java classes simplifies the implementation of business logic code and maximizes code reuse and consistency.

A simple example of a business domain-driven class hierarchy

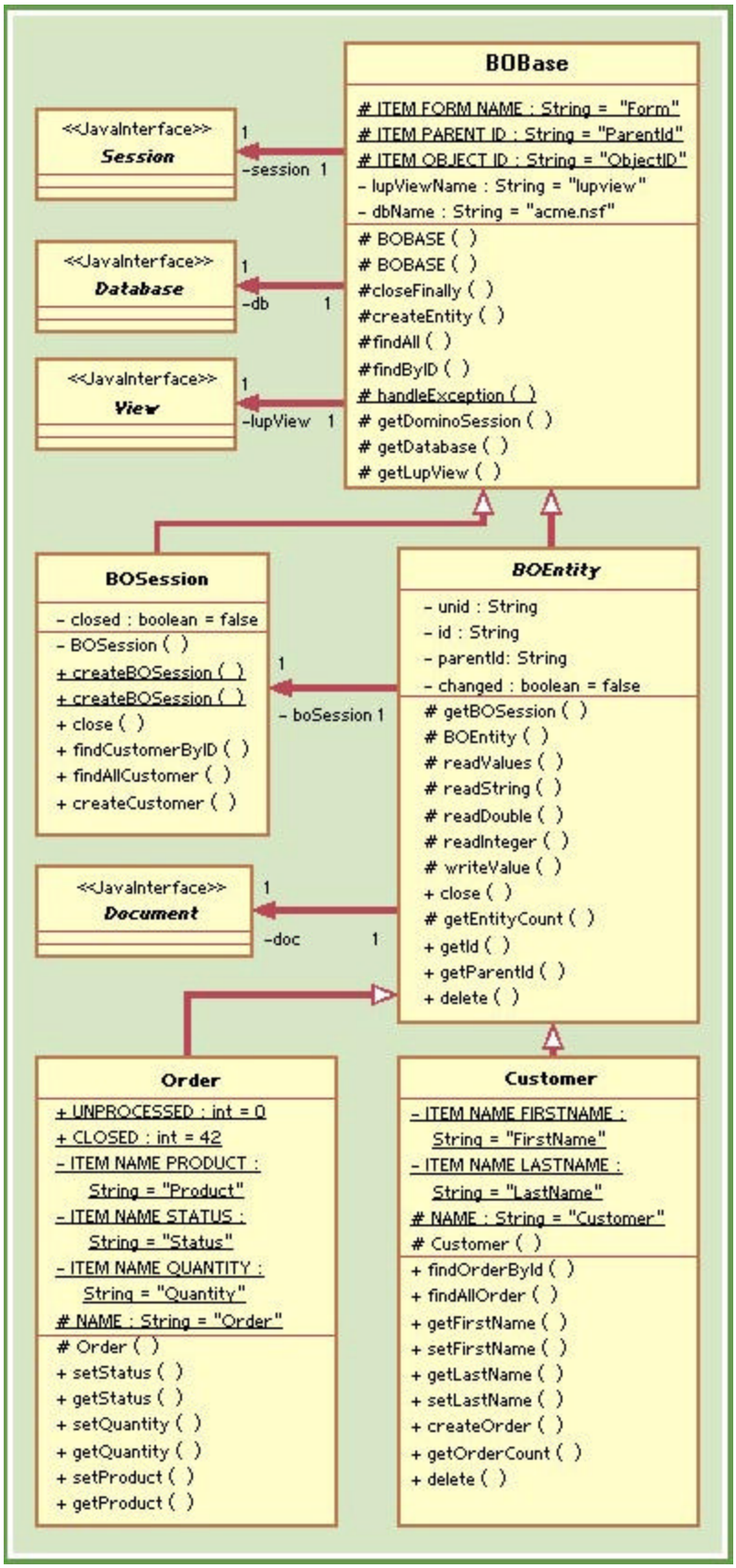
In the following sections of this article, we use a small and simple example to illustrate this concept. You can download sample files for this article from the Sandbox.

Let's suppose that a company called ACME has an order tracking system based on a Domino server infrastructure. This Domino application is integrated in the new ACME customer portal based on WebSphere Portal. To access the Domino database in an efficient way, you need to create a domain-driven class hierarchy.

A database named acme.nsf contains documents with customer data. Every customer can have one or more orders associated with him, stored in separate documents in the same database. The customer document contains a unique ID, and the order contains the unique customer ID as well as an order ID unique to the

customer. Therefore, in this example, the customer ID as well as the order ID is required to find a specific order. The business objects—customer and order—share some common functionality, like creation, deletion, retrieval, updating, and accessing the attributes of appropriate instances of these business object entities.

The following class diagram shows the classes BOBase, BOEntity, BOSession, Customer, and Order. The first three classes constitute the base classes of the class hierarchy. The latter two classes represent business domain-driven example classes. Additionally, the associations with the Domino Java API elements Session, Database, View, and Document are included in the class diagram and are tagged with an appropriate <<JavaInterface>> stereotype.



The base class for all business object entities stored in a Domino database is the class BOEntity. Because the business objects are structured in a containment hierarchy, the BOEntity class contains a reference to the instance and the UNID of the associated Notes document, the business domain-driven ID, and, if applicable, the associated parent object ID. A containment hierarchy allows the access of all child elements from a parent very easily, for instance, access to all orders associated with a specific customer.

The BOSession class offers static factory methods to create an instance of this class which encapsulates a Notes Session instance used to access the Notes database. This Notes session can be instantiated with either user name and password or an LTPA token. Alternatively, an anonymous Notes session can be instantiated. Every BOEntity instance contains an association to its belonging BOSession.

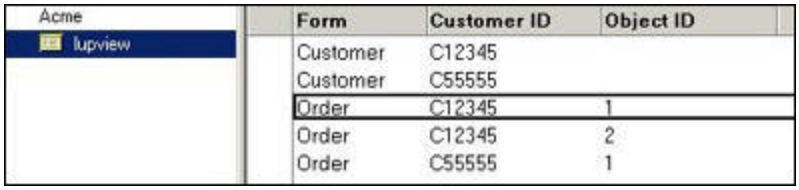
The customer class contains a method to create a new order and to find a specific order or to find all existing orders associated with a customer. A session class contains equivalent methods to create a new customer and to find one or all customers in the database. All derived entity classes and the session class inherit the common code of the create and find methods from the BOBase class. Later in this article, we describe these classes in more detail.

Implementation details

To keep the sample code simple, all documents are stored in the database acme.nsf and are accessed via one view called lupview. The view in this example contains three ascending sorted columns:

- Column one contains the form name, representing the business object type, such as Customer or Order.
- Column two contains the unique customer ID.
- Column three contains the per customer unique object ID.

This lookup view is used by the find methods—findCustomerById(), findAllCustomer(), findOrderById() and findAllOrder()—in this sample. The following screen shows the lupview.



The following code sample uses the business object classes. The first thing to do when you use these classes is to create a new session instance with the static method createBOSession(). The session is used to find an existing customer, to create a new order associated with this customer, and to set some attributes with the appropriate set methods. Although this example class accesses a Notes database, a view, and multiple documents, it does not need to import the Domino Java class packages directly because the Domino Java API classes are encapsulated in the business object classes.

```
import java.util.*;
import acme.*;

// Example1 - Find a customer and create a new
// order associated with this customer

public class Example1 {
    public static void main(String[] args) {
        BOSession boSession = BOSession.createBOSession();
        try {
            Customer customer = boSession.findCustomerById("C12345");

            Order newOrder = customer.createOrder("7");
            newOrder.setStatus(Order.UNPROCESSED);
            newOrder.setProduct("Chair");
```



```
            newOrder.setQuantity(4);
            newOrder.close();

            // further processing ...

            customer.close();
        } catch (InvalidStateException e) {
            e.printStackTrace();
        } catch (InvalidIdException e) {
            e.printStackTrace();
        } catch (CreateException e) {
            e.printStackTrace();
        } finally {
            boSession.close();
        }
    }
}
```

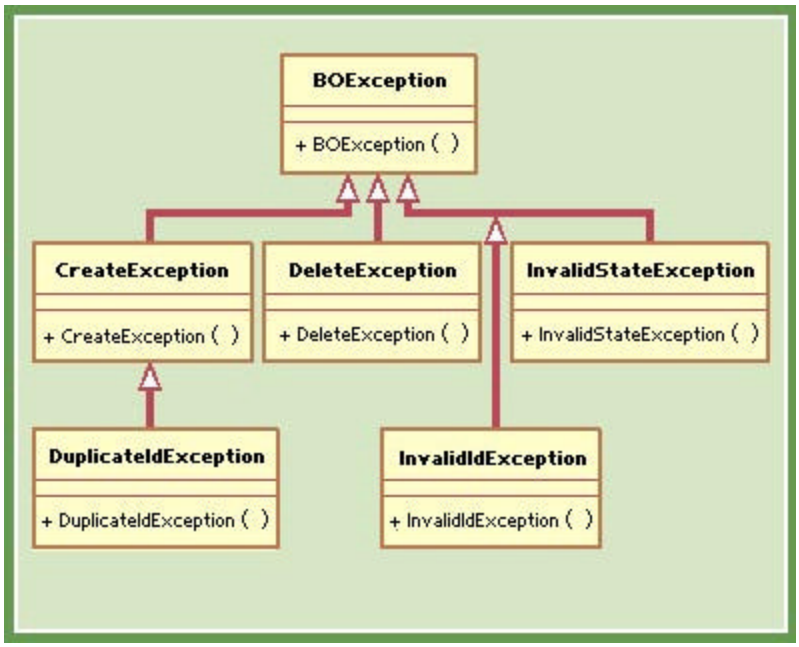
If the customer ID is invalid, an appropriate InvalidIdException exception is thrown in the following stacktrace:

```
acme.InvalidIdException: Business Object with ID [Customer C12349] not found!
    at acme.BOBase.findByID(BOBase.java:116)
    at acme.BOSession.findCustomerById(BOSession.java:71)
    at Example1.main(Example1.java:13)
```

If a new order is created with an already existing order ID for this customer, an appropriate DuplicateIdException is thrown in the following stacktrace. The class DuplicateIdException extends the base class CreateException. The mechanism to generate customer and order IDs and to guarantee their uniqueness is described later in this article.

```
acme.DuplicateIdException: Order with ID 7 already exists.
    at acme.Customer.createOrder(Customer.java:84)
    at Example1.main(Example1.java:15)
```

Exception classes
The next class diagram shows the different exceptions classes used in this example. They represent different fault situations that can occur in the processing of business logic and represent a different level of abstraction than the technical-driven NotesException class of the Domino Java API.



An example of the DeleteException is shown later in this article and the meaning of the InvalidStateException is described in a subsequent section.

Changing the status of an order
The second example shows how to find a customer and all associated orders for this customer, iterating through them in order to change the status attribute. This operation is completed by closing the changed orders. The close method writes any changes back to the Notes database, if any attribute has changed.

```
// Example2 - Find a customer, find all orders of this customer,
// iterate all orders and close them

...

    BOSession boSession = BOSession.createBOSession();
        try {
            Customer customer = boSession.findCustomerById("C12345");

            Vector vOrders = customer.findAllOrder();
            Enumeration enumOrders = vOrders.elements();
            while (enumOrders.hasMoreElements()) {
                Order order = (Order) enumOrders.nextElement();
                System.out.println("Order Id:"+order.getId()+" Customer Id:"+order.getParentId()+" Product:"+
                order.getProduct());

                // further processing ...

                order.setStatus(Order.CLOSED);
                order.close();
                }
                customer.close();
            } catch (InvalidIdException e) {
                e.printStackTrace();
            } finally {
                boSession.close();
            }
```


Deleting a customer
The third example shows how to find a customer name and how to delete it. In this simple example, the delete method of the customer class contains appropriate code to check if one or more orders for this customer still exist. If this is true, a DeleteException is thrown. An alternative implementation may try to delete the customer object and all dependent order objects in the database respectively or any other implementation that is consistent with the appropriate business logic.

```
// Example3 - Find a customer and try to delete it.
// If an order for this customer exists, an exception
// will be thrown.

...

BOSession boSession = BOSession.createBOSession();
try {
    Customer customer = boSession.findCustomerById("C12345");

    // further processing ...

    customer.delete();
    customer.close();
} catch (InvalidStateException e) {
    e.printStackTrace();
} catch (InvalidIdException e) {
    e.printStackTrace();
} catch(DeleteException e) {
    e.printStackTrace();
} finally {
    boSession.close();
}
```

Here is the DeleteException:

```
acme.DeleteException: Can't delete customer [C12345], because associated orders exist.
at acme.Customer.delete(Customer.java:100)
at Example3.main(Example3.java:18)
```

The BOBase and BOEntity classes
The following two code samples show the important parts of the base classes BOBase and BOEntity. The first code snippet shows the base class BOBase. It contains the common code for the findById and findAll methods accessing the lookup view and a basic create method, called createEntity, that creates a document in the Notes database and sets the required items: formname, object ID, and parent ID. Additionally, the base class contains a method to return the actual database instance. In this sample, only one database is used, but in a multiple database solution, the derived business object classes can override this method, returning their appropriate home Notes database instance. (In this example, a private String attribute can also be declared in derived business object classes, called dbName, with the name of the appropriate database.)

The method getLupView() returns the lookup view instance; this method is used by all find methods in the BOBase class. This method can also be overridden in a multiple database solution to return a valid view instance of the appropriate home Notes database or equivalent to the database name. A private String attribute can be declared with the appropriate view name.

```
package acme;

import java.util.Vector;
import lotus.domino.*;
```



```
public class BOBase {

 ...

protected Document createEntity(String id, String parentId, String boName) {
    Document doc = null;
    try {
        doc = getDatabase().createDocument();
        doc.replaceItemValue(ITEM_FORM_NAME,boName);
        doc.replaceItemValue(ITEM_OBJECT_ID,id);
        if(parentId != null) doc.replaceItemValue(ITEM_PARENT_ID,parentId);
        doc.save();
    }

    catch(NotesException ne) {
        handleException(ne);
    }
    return doc;
}

protected DocumentCollection findAll(Key key) {
    DocumentCollection retval = null;
    try {
        View lup = getLupView();
        if (lup != null) {
            retval = lup.getAllDocumentsByKey(key.getEntries(), true);
        }
    } catch (NotesException ne) {
        handleException(ne);
    }
    return retval;
}

protected Document findByID(Key key) throws InvalidIdException {
    Document retval = null;
    try {
        View lup = getLupView();
        if (lup != null) {
            Document doc = lup.getDocumentByKey(key.getEntries(),true);
            if (doc == null) {
                InvalidIdException idex =
                    new InvalidIdException("Business Object with ID "+key+" not found!");
                throw idex;
            }
        retval = doc;
        }
    } catch (NotesException ne) {
        handleException(ne);
    }
    return retval;
}

 ...

protected synchronized View getLupView() {
    try {
```

```
        if(lupView == null) {
        lupView = getDatabase().getView(lupViewName);
    }
        else {
        lupView.refresh();
    }
    } catch(NotesException ne) {
        handleException(ne);
    }
    return lupView;
    }
}
```

The following code sample shows a part of the base class BOEntity with the common code for the methods to read and write document items, to save the Notes document instance if it is changed, to recycle, and to delete the document. This class is the parent class for all business object classes which encapsulate a Notes document. The protected method checkState() verifies the private Notes document instance reference. If this reference is equal to null, an appropriate InvalidStateException is thrown. This methods allows a smart guard condition at the beginning of every business logic method that relies on the validity of the underlying Notes document. The method createOrder() in the customer class (see the code sample that follows this one) uses this guard condition method.

```
package acme;

import java.util.*;
import lotus.domino.*;

public abstract class BOEntity extends BOBase {

 ...

protected Vector readValues(String itemName) {
    Vector retval = null;
    try {
        retval = (Vector) doc.getItemValue(itemName);
    } catch (NotesException ne) {
        handleException(ne);
    }
    return retval;
}

 ...

protected void writeValue(String itemName, Object value) {
    try {
        if (value == null) {
            doc.removeItem(itemName);
        }
        doc.replaceItemValue(itemName, value);
    } catch (NotesException ne) {
        handleException(ne);
    }
    changed = true;
}

public void close() {
    if (doc == null)
```

```
            return;
        try {
            if (changed == true) {
                doc.save();
            }
            doc.recycle();
            doc = null;
        } catch (NotesException ne) {
            handleException(ne);
        }
    }

    ...

    protected boolean delete() throws DeleteException, InvalidStateException {
        this.checkState();

        boolean ret = false;
        try {
            synchronized (this) {
            ret = doc.remove(true);
            doc = null;
            }

        } catch(NotesException ne) {
            handleException(ne);
        }
        return ret;
    }

    protected void checkState() throws InvalidStateException {
        if(doc == null) throw
            new InvalidStateException("Business Object with id ["+this.getId()+"] is in an invalid state");
        }
    }
```

The customer class

The next code sample shows a part of the customer class with the methods to find one or all associated orders, to create a new order, to access the attributes of the customer via set and get methods, and to delete the customer document itself. The customer class extends the BOEntity class, like every business object class representing information stored in a Notes document. All constructors of derived business object entity classes are protected; therefore, every code outside the business object package must use the appropriate create methods of the parent classes to instantiate the business object entity classes.

An important category of methods in every business object class that can contain dependent business objects is the set of different find methods to retrieve these dependent objects. In the customer class, two different find methods are offered. The first method, called findOrderById(), builds a key with the searched object type, the customer ID of this instance, and the passed order ID. This key instance is passed to the generic findById() method inherited from the BOBase class (see the previous code sample). The returned Notes document of this method is wrapped in an instance of the order class. The second find method in the customer class, findAllOrder(), uses a key with the searched object type and the customer ID of this instance. This key instance is passed to the generic findAll() method inherited from the BOBase class. Every Notes document of the returned Notes document collection is wrapped in its own instance of the order class. The findAllOrder() method returns all order instances in a vector instance.

```
package acme;
```



```
import lotus.domino.*;
import java.util.Vector;

public class Customer extends BOEntity {

 ...

public Order findOrderById(String orderId) throws InvalidIdException {
    Key key = new Key();
    key.appendEntry(Order.NAME);
    key.appendEntry(this.getId());
    key.appendEntry(orderId);

    Document doc = findByID(key);
    return new Order(orderId, this.getId(), doc, getBOSession());
}

public Vector findAllOrder() {
    Key key = new Key();
    key.appendEntry(Order.NAME);
    key.appendEntry(this.getId());

    DocumentCollection documents = findAll(key);
    Vector vecOrders = new Vector();
    try {
        if(documents != null && documents.getCount() > 0) {
            Document doc = documents.getFirstDocument();
            while (doc != null) {
                Order order = new Order(doc.getItemValueString(Order.ITEM_OBJECT_ID),
                    this.getId(), doc, getBOSession());
                vecOrders.addElement(order);
                doc = documents.getNextDocument();
            }
        }
    } catch (NotesException ne) {
        handleException(ne);
    }
    return vecOrders;
}

...

public String getLastName() {
    return readString(ITEM_NAME_LASTNAME);
}
public void setLastName(String lastName) {
    writeValue(ITEM_NAME_LASTNAME,lastName);
}

public Order createOrder(String newOrderId) throws CreateException, InvalidStateException {
this.checkState();

    synchronized (this.getClass()) {
    try {
        Order order = this.findOrderById(newOrderId);
        if(order != null) {
            order.close();
```

```
                throw new DuplicateIdException("Order with Id "+newOrderId+" already exists.");
            }
        } catch (InvalidIdException e) {
            // ignore exception if an order with the new id doesn't exist
        }
        Document doc = createEntity(newOrderId, getParentId(),Order.NAME);
        return new Order(newOrderId, getParentId(), doc, getBOSession());
        }
    }

    public boolean delete() throws DeleteException, InvalidStateException {
        long n = getOrderCount();
        if(n != 0) {
            // optionally check the status of the order
            throw new DeleteException("Can't delete customer ["+this.getId()+"], because associated orders exist.");
    }
    // ... or alternative implementation: delete all dependent objects
    return super.delete();
}
```

The createOrder() method

The method createOrder() in the previous code sample uses the findOrderById() method of the customer class to determine if an order with the passed ID for this customer already exists in the Notes database. If an order already exists, an appropriate DuplicateIdException is thrown, and the creation of the order is canceled. If no order with the passed ID exists, the thrown InvalidIdException is ignored, and a new document to house the order information is created for this customer with the createEntity() method. The returned new document contains the parent (customer) ID and its own order ID. Furthermore, the form item specifies the document of the type order. This new document is passed to the constructor of the order class. This business object encapsulates the Notes document in a private instance variable inherited from the BOEntity class, so any direct access to this document and its items is prevented. Only set and get methods of the order class allow the access of the information of this business object.

The method createOrder() encloses the code described previously with a synchronized block. This synchronized code block acquires the lock associated with the customer class object before it executes. This code guarantees that the sequence of code is executed by only one instance of the customer class in one thread in a Java Virtual Machine at the same time, regardless of how many threads try to execute the code of the method createOrder() of their different customer instances.

The delete method in the previous code sample demonstrates how the classes can help to assure the consistency of the business objects. In this example business case, it is not allowed to delete a customer as long as associated orders exist.

With the checkState() method call as a guard condition in the delete method of BOBase, it is impossible to retrieve a customer instance via the appropriate findCustomerById() method of the BOSession class, to delete this customer, and to create a new order for this customer in the same sequence of code without enforcing an exception to be thrown, as demonstrated in the following code sample.

```
Customer customer = boSession.createCustomer("C9999");

// further processing ...
customer.delete();
// further processing ...

// this customer instance isn't valid anymore,
// therefore the createOrder() will throw a InvalidStateException
customer.createOrder("22");
customer.close();
```


This sequence of code causes an InvalidStateException.

```
acme.InvalidStateException: Business Object with id [C9999] is in an invalid state
    at acme.BOEntity.checkState(BOEntity.java:165)
    at acme.Customer.createOrder(Customer.java:79)
    at Example6.main(Example6.java:26)
```

Things you need to know

It should be mentioned that the samples shown in this article have some simplifications. The following list summarizes some of these topics and outlines possible enhancements of the presented example:

- *Only one Notes database is considered to be the source for associated business objects.*
  Alternatively, every derived business object entity class can hold information about its source Notes database by overriding the BOBase's getDatabase() method and the appropriate associated views returned by an overridden getLupView() method.

- *Only one Notes lookup view is used by the find methods.*
  In a more complex system, more specialized lookup views and corresponding find methods may be needed. For instance, you can use the findCustomerByName method to search for a customer by the last name to implement general-purpose uses case scenarios.

- *In the example, all information is retrieved by accessing the appropriate documents.*
  Special read-only business objects can control the access and encapsulate the information delivered by a Notes View Entry Collection, for example, to return a huge collection of customer summary information from an overview list—instead of using the findAllCustomer method—return a collection of encapsulated customer documents.

- *Instead of the vector class used in the example, you can use the Java 2 collection API interfaces and classes in a Notes 6 or Domino 6 infrastructure context.*
  The Java 1.1 class hierarchy contains only rudimentary support for container and abstract data type classes; Java 2 introduces the Java Collection API to fix this shortcoming in the Java 1.1 APIs.

- *The customer class can be an abstract base class for a family of different customer types, such as private or business customer.*
  A findAllCustomer method can return a vector of instances of different derived customer types, and specialized find methods can return the appropriate subset.

- *In the example, no programmatic associations between the business objects exist.*
  You can store a collection of orders in an instance variable for the appropriate customer.

- *In this example, the BOSession class is used in a stand-alone program context with an anonymous Notes session.*
  However, it can be instantiated with an LTPA token in a servlet or EJB context.

- *The demonstrated approach to prevent duplicate IDs with the synchronized code in the create methods works well if all create operations are executed in the same Java Virtual Machine (JVM).*
  If programs are executed in different JVMs (for instance, using the BO classes in Web-triggered Domino Java Agents or in two different WAS instances), the described mechanism for the creation of unique IDs doesn't work! A solution may be to replace the synchronized code in the create methods with an appropriate call via HTTP of a servlet in a one "centralized" servlet engine. This centralized servlet engine should offer high availability with appropriate failover mechanisms. However, that topic is outside of the introductory scope of this article.

- *The statement to guarantee the consistency of the business objects can only be fulfilled if no other code than the business object code accesses the appropriate Notes documents directly.*

- *In the example, it is necessary to get a customer instance before accessing the associated orders because*

of the dependency between these two business objects.
This containment hierarchy doesn't exist in every business domain context. All business objects are accessible via their own unique key instead of a hierarchical key.

Summary

The presented concept of using business domain-driven class hierarchies can simplify Domino application development because of the higher amount of reused code. The encapsulation of Domino specific implementation details in a small set of common base classes reduces the need for appropriate Domino Java API skills and supports the creation of error free and easily maintainable code.

ABOUT THE AUTHOR
Kai-Hendrik Komp is a Systems Architect with IBM Software Services for Lotus (ISSL) in Munich, Germany and has been working on various development projects over the last few years. Kai holds a diploma in Physics and different Lotus Principal CLP and Sun Java certifications. He has been programming computers since 1985 at the beginning transputer-based parallel computing systems with C; began using Java in 1997; and joined IBM in the same year.

ACKNOWLEDGEMENT
Thanks to Ragnar Schierholz, a diploma candidate whom Kai-Hendrik advised in the last year and who is now working as a Ph.D. candidate at the University of St. Gallen, for the helpful discussion of parts of the presented concept.

