CHAPTER 6

Object-Oriented Database Management Systems: db4o

This chapter introduces the student to db4o, a very popular open-source object-oriented database management. db4o provides programming APIs in Java as well as several .Net languages, but this chapter focuses only on the Java API.

The `COMPANY` database of the Elmasri-Navathe text is used throughout this chapter. In Section 6.1, db4o installation as well as an overview of the API packages is introduced. Section 6.2 presents an elementary example of creating and querying a database. Section 6.3 presents database updates and deletes. The company database is defined in Section 6.4. Database querying is covered in Section 6.5. A complete Java application that creates and queries the company database is introduced in Section 6.6. Data about objects is read from text files and various persistent objects are created and queried. A Web application to access the company database is illustrated in Section 6.7.

6.1 db4o Installation and Getting Started

The db4o object database system can be obtained from www.db4o.com. The installation of the system is quite straightforward. All it requires is the placement of `db4o.jar` in the CLASSPATH of your favorite Java installation.

The main packages of the Java API are: `com.db4o` and `com.db4o.query`. The important classes/interfaces in the `com.db4o` package are:

1. `com.db4o.Db4o`: the factory class that provides methods to configure the database environment and to open a database file.
2. `com.db4o.ObjectContainer`: the database interface that provides methods to store, query and delete database objects as well as commit and rollback database transactions.

The `com.db4o.query` package contains the Predicate class that allows for "Native Queries" to be executed against the database. Native queries provide the ability to run one or more lines of code to execute against all instances of a class and return those that satisfy a predicate.

A typical sequence of statements to work with the database is shown below:

```
Configuration config = Db4o.configure();
ObjectContainer db = Db4o.openFile(config,"student.db4o");
…
…
db.close();
```

Initially, a configuration object is created. Various database configuration settings can be made. In this example, none of the settings are shown, however, in later examples some of the important

settings will be illustrated. The configuration object along with a database file name is provided to the `openFile()` method of the factory class. If the database file does not exist, a new database is created. Otherwise, the database present in the file is opened. A database transaction also begins at this point. The `ObjectContainer` object, `db`, is then used to perform database operations such as inserting new objects, updating or deleting existing objects, and querying the database. Transactions can also be committed or rolled back. At the end, the database is closed.

6.2 A Simple Example

In this section, a simple example is introduced in which a database of student objects is created and retrieved. Consider the following Student class:

```
public class Student {
  int sid;
  String lname;
  String fname;
  float gpa;

  public Student(int sid, String lname, String fname, float gpa) {
    this.sid = sid;
    this.lname = lname;
    this.fname = fname;
    this.gpa = gpa;
  }

  public int getSid() {
    return sid;
  }

  public void setSid(int sid) {
    this.sid = sid;
  }

  public String getLname() {
    return lname;
  }

  public void setLname(String lname) {
    this.lname = lname;
  }

  public String getFname() {
    return fname;
  }

  public void setFname(String fname) {
    this.fname = fname;
  }

  public float getGpa() {
```

```
    return gpa;
  }

  public void setGpa(float gpa) {
    this.gpa = gpa;
  }

  public String toString() {
    return sid+" "+fname+" "+lname;
  }
}
```

The class defines a student object with four simple attributes: `sid`, `lname`, `fname`, and `gpa`. There are the usual getter and setter methods along with a standard constructor and a `toString()` method. The following is a sample Java code that creates a few student objects and stores them in the database:

```
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.config.Configuration;

public class CreateStudent {
  public static void main(String[] args) {
    Configuration config = Db4o.configure();
    ObjectContainer db = Db4o.openFile(config,"student.db4o");
    createFewStudents(db);
    db.close();
  }

  public static void createFewStudents(ObjectContainer db) {
    //Create few student objects and store them in the database
    Student s1 = new Student(1000,"Smith","Josh", (float) 3.00);
    Student s2 = new Student(1001,"Harvey", "Derek", (float) 4.00);
    Student s3 = new Student(1002,"Lemon", "Don", (float) 3.50);
    db.store(s1);
    db.store(s2);
    db.store(s3);
  }
}
```

As can be seen in the above code, the program opens a database file and calls the `createFewStudents()` method which creates three student objects and stores each one of them into the database using the `store()` method. At the end, the main method closes the database.

The following Java code prints the contents of the database that was just created:

```
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
```

```
import com.db4o.config.Configuration;

public class PrintStudents {
  public static void main(String[] args) {
    Configuration config = Db4o.configure();
    ObjectContainer db = Db4o.openFile(config,"student.db4o");
    printStudents(db);
    db.close();
  }

  public static void printStudents(ObjectContainer db) {
    ObjectSet result = db.queryByExample(Student.class);
    System.out.println("Number of students: " + result.size()+"\n");
    while (result.hasNext()) {
      Student s = (Student) result.next();
      System.out.println(s);
    }
  }
}
```

The above program opens the database and calls `printStudents()` method to print all student objects in the database. The program illustrates one of the many ways to query the database: the query by example method. In this method, the `queryByExample()` method is called on the `ObjectContainer db`. By providing `Student.class` as input the method returns all student objects. db4o also provides Java 5 generics shortcut using the `queryByExample()` method; so the `printStudents()` code can be rewritten as follows:

```
List <Student> result1 = db.queryByExample(Student.class);
System.out.println("Number of students: " + result1.size()+"\n");
for (int i=0; i<result1.size(); i++) {
  Student s = result1.get(i);
  System.out.println(s);
}
```

The `queryByExample()` method can also be invoked with a template object and the method returns all objects that match the template object in the non-default values provided. For example, the following sequence of statements will query the database for all students whose last names is "`Smith`":

```
Student s = new Student(0,"Smith",null,(float) 0.0);
ObjectSet result = db.queryByExample(s);
```

Default values are `0` for `int`, `0.0` for `float`/`double`, and `null` for `String` and other objects. Default values act as wild cards in the template object and objects that match the non-default values in the template are returned by the method.

6.3 Database Updates and Deletes

Updating and deleting objects is quite straightforward in db4o. In either case, the first step is to retrieve the object to be updated or deleted. To update the object, an appropriate method defined in the class for the object is then invoked, which performs the necessary update in memory. Then, the `store()` method is invoked to make the update persistent on disk. Here is a code fragment that changes the GPA of student with `sid=1000`:

```
Student s = new Student(1000,null,null,(float) 0.0);
List <Student> result1 = db.queryByExample(s);
s = (Student) result1.get(0);
s.setGpa((float) 3.67);
db.store(s);
```

Here, the student object corresponding to `sid=1000` is retrieved in the first three lines of code. Then, the `setGpa()` method is called to update the GPA of the student. Finally, the `store()` method is called to make the change permanent.

To delete an object, the `delete()` method is called on the object container database object with the object to be deleted as the argument. For example, if the student with `sid=1002` is deleted by executing the following code:

```
s = new Student(1002,null,null,(float) 0.0);
List <Student> result2 = db.queryByExample(s);
s = (Student) result2.get(0);
db.delete(s);
```

6.4 Company Database

In this section, the company database of the Elmasri-Navathe textbook is represented as an object-oriented database. The design includes classes for entity sets *Employee*, *Department*, *Project*, and *Dependent*. To represent the many-to-many relationship *worksOn*, a separate class is designed. The class definitions showing only the attributes are defined as follows:

```
public class Department {
  // attributes
  private int dnumber;
  private String dname;
  private Vector<String> locations;
  // relationships
  private Vector<Employee> employees;
  private Employee manager;
  private Vector<Project> projects;
  // one-to-many relationship (manager) attribute
  private String mgrStartDate;
}
```

```
public class Employee {
  // attributes
  private int ssn;
  private String fname;
  private char minit;
  private String lname;
  private String address;
  private String bdate;
  private float salary;
  private char sex;
  //relationships
  private Department worksFor;
  private Department manages;
  private Vector<WorksOn> worksOn;
  private Vector<Dependent> dependents;
  private Employee supervisor;
  private Vector<Employee> supervisees;
}

public class Project {
  // attributes
  private int pnumber;
  private String pname;
  private String location;
  // relationships
  Department controlledBy;
  Vector<WorksOn> worksOn;
}

public class Dependent {
  // attributes
  private String name;
  private char sex;
  private String bdate;
  private String relationship;
  // relationships
  private Employee dependentOf;
}

public class WorksOn {
  // attribute
  float hours;
  //owner attributes
  Employee employee;
  Project project;
}
```

The above design is a straightforward translation of the ER diagram for the Company database shown in page 225 of the 6th edition of the Elmasri-Navathe textbook. Simple attributes of entity sets are represented as instance variables of primitive types (e.g. `ssn` in `Employee`) and multiple-valued attributes as Java Vectors (e.g. `locations` in `Department`). Single-valued

relationships are represented as references to objects (e.g. `manager` in `Department`) and many-valued relationships are represented as Vectors of object references (`employees` in `Department`). The only many-to-many relationship, `worksOn`, is represented by a separate class with a simple attribute, hours, and two object references, one to `Employee` object and the other to `Project` object involved in the relationship. The usual constructors, getter and setter methods are also defined in the classes.

6.5 Database Querying

There are three main methods to query and retrieve objects in db4o: Query by Example, Native Queries, and SODA (Simple Object Database Access).

6.5.1 Query by Example

The query by example method has already been discussed. In this approach, an object template is provided as input to the `queryByExample()` method, which then retrieves all objects that match the non-default values of the template. This method works well in many cases, but has its limitations. For example, one cannot constrain objects on default values such as 0, null etc as these are treated as default values; one cannot perform advanced query operations such as "and", "or", and "not".

6.5.2 Native Queries

To avoid the limitations of the query by example method, db4o provides the native queries system. Native queries provide the ability to execute one or more lines of code on all instances of a class and select a subset of the instances that satisfy a criterion specified by the code. Here is a simple example in Java 1.5 to retrieve the department object for department with `dnumber = 5`.

```
List<Department> depts = db.query(new Predicate<Department>() {
  public boolean match(Department dept) {
    return (dept.getDnumber() == 5);
  }
});
Department d = depts.get(0);
```

The `query()` method takes a `Predicate` object as its argument. The `Predicate` object encapsulates a Boolean method called `match()` which will be applied to all instances of the class on which the predicate is defined. All instances that evaluate to `True` will be returned as the value of the `query()` method.

As a more complicated query, consider "*Find departments that have a location in Houston or have less than 4 employees or controls a project located in Phoenix*". The following Native query code prints such departments:

```
List<Department> depts = db.query(new Predicate<Department>() {
  public boolean match(Department dept) {
```

```
      int nEmps = dept.getEmployees().size();
      Vector<Project> prjs = dept.getProjects();
      boolean foundPhoenix = false;
      for (int i=0; i<prjs.size(); i++) {
        Project p = prjs.get(i);
        if (p.getLocation().equals("Phoenix")) {
          foundPhoenix = true;
          break;
        }
      }
      return dept.getLocations().contains("Houston") ||
             (nEmps < 4) || foundPhoenix;
    }
  });
  for (int i=0; i<depts.size(); i++)
    System.out.println("Department: "+depts.get(i));
```

The `match()` method checks for each of the three conditions (department has a location in `Houston`, department has less than 4 employees, and department controls a project located in `Phoenix`) and returns true if any one or more of the conditions is satisfied by the department. The example illustrates the power of Native queries where complex conditions can be coded in Java to be tested on instances of the class.

6.5.3 SODA (Simple Object Database Access) Queries

SODA API is db4o's low-level access to the nodes of the data graph of the underlying object-oriented database. It gives flexibility in expressing dynamic queries and therefore is an important too to use to build object-oriented database applications. For most routine situations, however, native queries are an excellent choice.

For the next set of examples, let us assume the following method to print the results of the query is available:

```
public static void printResult(ObjectSet result) {
    System.out.println(result.size());
    while(result.hasNext()) {
        System.out.println(result.next());
}
```

Query 1: Find employees with last name `King`.

```
Query query = db.query();
query.constrain(Employee.class);
query.descend("lname").constrain("King");
ObjectSet result=query.execute();
printResult(result);
```

The above code fragment first creates a `Query` object that represents a query graph with nodes denoting database objects that have constraints associated with them. The `Query` object is created by invoking the `query()` method on the `ObjectContainer db`. The `constrain()` method allows us to attach a constraint to the node in the query graph. A commonly used special argument to the method is a class, e.g. `Employee.class` in the above example, which constrains the node to objects of the class. The `descend()` method creates a child node in the query graph and associates the node with the specified field name parameter. In the example, a child node is created for the `lname` field, which is constrained by the string "`King`". Once the query graph is created with all the constraints, the query can be executed with the execute() method, which returns all objects for the root node of the query graph that satisfy the various constraints imposed. The above code fragment will print the employees with last name `King`.

Query 2: Find employees with salary = 25000.

```
Query query = db.query();
query.constrain(Employee.class);
query.descend("salary").constrain(new Integer(25000));
ObjectSet result = query.execute();
printResult(result);
```

This example is similar to the previous one except that the constraint is on the salary field with an `Integer` object.

Query 3: Find projects not located in Houston.

```
Query query = db.query();
query.constrain(Project.class);
query.descend("location").constrain("Houston").not();
ObjectSet result = query.execute();
printResult(result);
```

This example illustrates the use of the `not()` modifier on a constraint. Any constraint that is associated with a query graph node is first evaluated and then negated.

Query 4: Find employees with last name King and salary = 44000.

```
Query query = db.query();
query.constrain(Employee.class);
Constraint constr = query.descend("lname").constrain("King");
query.descend("salary").constrain(new Integer(44000)).and(constr);
ObjectSet result = query.execute();
printResult(result);
```

This example illustrates the use of the `and()` modifier. A separate Constraint object for constraining the last name is created and is combined with the salary constraint using the `and()` modifier.

Query 5: Find projects pnumber > 90.

```
Query query = db.query();
query.constrain(Project.class);
query.descend("pnumber").constrain(new Integer(90)).greater();
ObjectSet result = query.execute();
printResult(result);
```

This example illustrates the use of `greater()` modifier to make comparisons other than "equality", the default comparison in the constrain() method. Other methods available are `smaller()`, `like()`, `startsWith()`, and `equal()`.

Query 6: Find projects with pnumber > 90 or with location in Houston.

```
Query query = db.query();
query.constrain(Project.class);
Constraint constr = query.descend("location").constrain("Houston");
query.descend("pnumber").constrain(new Integer(90))
                        .greater().or(constr);
ObjectSet result = query.execute();
printResult(result);
```

This example illustrates the use of the `or()` modifier.

Query 7: Find employees in sorted order.

```
Query query = db.query();
query.constrain(Project.class);
query.descend("pname").orderAscending();
ObjectSet result = query.execute();
printResult(result);
query.descend("pname").orderDescending();
result = query.execute();
printResult(result);
```

This example illustrates the `orderAscending()` and `orderDescending()` methods available to sort the results of a query.

Query 8: Find departments whose manager's last name is Wong.

```
Query query = db.query();
query.constrain(Department.class);
query.descend("manager").descend("lname").constrain("Wong");
ObjectSet result = query.execute();
Department d = (Department) result.next();
System.out.println("Wong managed department: "+d);
```

This example illustrates SODA querying which requires traversing an object reference. Here, the `manager` field of `Department` object is descended in the SODA query.

6.6 Company Database Application

In this section, a complete application that creates and queries the company database is introduced. It is assumed that the five classes: `Employee.java`, `Department.java`, `Project.java`, `Dependent.java`, and `WorksOn.java` are created as defined in Section 6.4. Included in their definitions are standard constructor, setter, getter, and toString methods. Once these class definitions are compiled the following application modules can be created.

6.6.1 CreateDatabase.java

The first module in the company application is the `CreateDatabase` program that reads data from text files and creates the object-oriented database. The main program along with the import statements is shown below:

```
import java.util.*;

import com.db4o.*;
import com.db4o.config.Configuration;
import com.db4o.query.*;

public class CreateDatabase {
  public static void main(String[] args) {
    String DB4OFILENAME = args[0];
    Configuration config = Db4o.configure();
    config.objectClass(Employee.class).cascadeOnUpdate(true);
    config.objectClass(Department.class).cascadeOnUpdate(true);
    config.objectClass(Project.class).cascadeOnUpdate(true);
    config.objectClass(Dependent.class).cascadeOnUpdate(true);
    config.objectClass(WorksOn.class).cascadeOnUpdate(true);
    config.updateDepth(1000);
    ObjectContainer db = Db4o.openFile(config,DB4OFILENAME);

    try {
      createEmployees(db);
      createDependents(db);
      createDepartments(db);
      createProjects(db);
      setManagers(db);
      setControls(db);
      setWorksfor(db);
      setSupervisors(db);
      createWorksOn(db);
      db.commit();
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
    }
    finally {
```

```
        db.close();
      }
      System.out.println("DONE");
    }
  }
```

The main program takes as command line argument the name of the database file, `company.db4o`. This file will contain the object-oriented database at the termination of the program. Initially, a `Configuration` object, `config`, is created and several properties are set. For each object class, the `cascadeOnUpdate` property is set to `true` using the following Java statement (shown for the `Employee class`):

```
config.objectClass(Employee.class).cascadeOnUpdate(true);
```

This property indicates to the db4o database engine that all updates should be made persistent including chained references. By default, in response to the `store()` method call, db4o makes only the top-level object persistent and any chained references to objects such as Vector of references etc. are not made persistent. By setting the property to `true`, db4o makes all chained references persistent as well. The other property setting is:

```
config.updateDepth(1000);
```

This sets the depth of the chained references to be a large number. After these settings, the database object is created. Once the database object is created, it can be used to create objects within each class by reading data from text files by calling various methods. Each of these methods is discussed next.

6.6.2 createEmployees

This method reads data about employees from a text file which contains the number of employees in the first line and details about each employee in subsequent lines. The individual fields describing the employee are separated by a colon. The first few lines of the text file, `employee.dat`, are shown below:

```
40
James:E:Borg:888665555:10-NOV-27:450 Stone, Houston, TX:M:55000
Franklin:T:Wong:333445555:08-DEC-45:638 Voss, Houston, TX:M:40000
Jennifer:S:Wallace:987654321:20-JUN-31:291 Berry, Bellaire, TX:F:43000
```

The method makes use of a text file reading class called `InputFile.java` which is provided along with the source code of the lab manual. The methods of this class are self explanatory. The code for the `createEmployees` method is shown below:

```
public static void createEmployees(ObjectContainer db)
                                    throws Exception {
  InputFile fin = new InputFile();
  if (fin.openFile("data/employee.dat")) {
```

```
      int nEmps = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nEmps; i++) {
        String line = fin.readLine();
        String[] fields = line.split(":");
        String fname = fields[0];
        char minit = fields[1].charAt(0);
        String lname = fields[2];
        int ssn = Integer.parseInt(fields[3]);
        String bdate = fields[4];
        String address = fields[5];
        char sex = fields[6].charAt(0);
        float salary = Float.parseFloat(fields[7]);
        Employee e = new Employee(
          ssn, fname, minit, lname, address, bdate, salary, sex);
        db.store(e);
      }
    }
  }
```

The method starts off by opening the text file and reading the first line into an integer variable. Then, for each employee, it reads the details into a string variable, uses the `split()` method to break up the individual fields, and finally creates the `Employee` object by calling its constructor and makes the object persistent by invoking the `store()` method. Note that at this point, none of the employee objects have their object references to other objects set. These will be done in subsequent method calls.

6.6.3 createDependents

The data file (`dependent.dat`) from which the `createDependent` method reads information about dependent objects is shown below:

```
11
333445555,Alice,F,05-APR-1976,Daughter
333445555,Theodore,M,25-OCT-1973,Son
333445555,Joy,F,03-MAY-1948,Spouse
987654321,Abner,M,29-FEB-1932,Spouse
123456789,Michael,M,01-JAN-1978,Son
123456789,Alice,F,31-DEC-1978,Daughter
123456789,Elizabeth,F,05-MAY-1957,Spouse
444444400,Johnny,M,04-APR-1997,Son
444444400,Tommy,M,07-JUN-1999,Son
444444401,Chris,M,19-APR-1969,Spouse
444444402,Sam,M,14-FEB-1964,Spouse
```

It has a similar format as the `employee.dat` file. The first line contains the number of dependents in the file and the remaining lines contain the individual fields describing the dependent. The first field is the social security number of the employee owner of the dependent. The following is the code for creating the dependent objects:

```
public static void createDependents(ObjectContainer db)
          throws Exception {
  InputFile fin = new InputFile();
  if (fin.openFile("data/dependent.dat")) {
    int nDeps = Integer.parseInt(fin.readLine());
    for (int i = 0; i < nDeps; i++) {
      String line = fin.readLine();
      String[] fields = line.split(",");
      final int essn = Integer.parseInt(fields[0]);
      String name = fields[1];
      char sex = fields[2].charAt(0);
      String bdate = fields[3];
      String relationship = fields[4];
      List<Employee> emps = db.query(new Predicate<Employee>() {
        public boolean match(Employee emp) {
          return (emp.getSsn() == essn);
        }
      });
      Employee e = emps.get(0);
      Dependent d = new Dependent(name, sex, bdate, relationship);
      d.setDependentOf(e);
      db.store(d);
      e.addDependent(d);
      db.store(e);
    }
  }
}
```

The above method proceeds in a similar manner as `createEmployees`. One main difference is that a native query is executed to obtain a reference to the employee object corresponding to the social security member. The dependent object is created and the reference to the employee object is set by calling the `setDependentOf()` method. At the same time, a reference to the dependent object is set in the employee object using the method `addDependent()`. Both updates are made persistent by calling the `store()` method.

6.6.4 createDepartment

The data file (department.dat) consists of information about departments. The locations for the department listed at the end of the line, separated by commas. A sample file is shown below:

```
6
1:Headquarters:Houston
4:Administration:Stafford
5:Research:Bellaire,Sugarland,Houston
6:Software:Atlanta,Sacramento
7:Hardware:Milwaukee
8:Sales:Chicago,Dallas,Philadephia,Seattle,Miami
```

The code for the method is shown below:

```
  public static void createDepartments(ObjectContainer db)
            throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/department.dat")) {
      int nDepts = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nDepts; i++) {
        String line = fin.readLine();
        String[] fields = line.split(":");
        int dnumber = Integer.parseInt(fields[0]);
        String dname = fields[1];
        String[] ls = fields[2].split(",");
        Vector<String> locs = new Vector<String>();
        for (int j = 0; j < ls.length; j++)
          locs.add(ls[j]);
        Department d = new Department(dnumber, dname, locs);
        db.store(d);
      }
    }
  }
```

The code is similar to previous methods. One difference is that a vector of string objects is created to store the different locations for the department. Again, references to other objects are not stored in this method. This will be done in separate methods to follow.

6.6.5 createProjects

The data file for creating project objects is shown below:

```
11
1,Product X,Bellaire
2,Product Y,Sugarland
3,Product Z,Houston
10,Computerization,Stafford
20,Reorganization,Houston
30,New Benefits,Stafford
61,Operating Systems,Jacksonville
62,Database Systems,Birmingham
63,Middleware,Jackson
91,Inkjet Printers,Phoenix
92,Laser Printers,Las Vegas
```

The code to read this data and create project objects is shown below:

```
  public static void createProjects(ObjectContainer db)
            throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/project.dat")) {
      int nProjs = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nProjs; i++) {
```

```
        String line = fin.readLine();
        String[] fields = line.split(",");
        int pnumber = Integer.parseInt(fields[0]);
        String pname = fields[1];
        String loc = fields[2];
        Project p = new Project(pnumber, pname, loc);
        db.store(p);
      }
    }
  }
```

The code is self-explanatory as it is very similar to previous examples.

6.6.6 createWorksOn

The `worksOn` relationship is a many-to-many relationship between `Employee` and `Project`. The relationship also contains an attribute, `hours`. This has been modeled by a separate class, `WorksOn` which has only one ordinary attribute, `hours`. It also consists of two object reference, one points to the employee object and the other to the project object involved in the relationship. A portion of the data file is shown below:

```
48
123456789,1,32.5
123456789,2,7.5
666884444,3,40.0
453453453,1,20.0
```

There are three fields describing each worksOn relationship: social security number of employee, project number, and number of hours. The code to read the file and create the objects is shown below:

```
  private static void createWorksOn(ObjectContainer db)
            throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/worksOn.dat")) {
      int nWorksOn = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nWorksOn; i++) {
        String line = fin.readLine();
        String[] fields = line.split(",");
        final int essn = Integer.parseInt(fields[0]);
        final int pno = Integer.parseInt(fields[1]);
        float hours = Float.parseFloat(fields[2]);
        List<Employee> emps = db.query(new Predicate<Employee>() {
          public boolean match(Employee emp) {
            return (emp.getSsn() == essn);
          }
        });
        Employee e = emps.get(0);
        List<Project> prjs = db.query(new Predicate<Project>() {
```

```
          public boolean match(Project prj) {
            return (prj.getPnumber() == pno);
          }
        });
        Project p = prjs.get(0);
        WorksOn won = new WorksOn(hours);
        won.setEmployee(e);
        won.setProject(p);
        db.store(won);
        e.addWorksOn(won);
        p.addWorksOn(won);
        db.store(e);
        db.store(p);
      }
    }
  }
```

For each `worksOn` relationship entry, the method retrieves the `Employee` object for the given social security number and the `Project` object for the given project number using native query approach. Once these object references are obtained, a `worksOn` object is created with the given hours value and the two object references are set. Finally, all objects are made persistent using the `store()` method call.

6.6.7 setManagers

This method sets the object references for the one-to-one relationship, `Manages`. The data file contains the department number, its manager's social security number and the start date for each department. The data file contents are shown below:

```
6
1,888665555,19-JUN-1971
4,987654321,01-JAN-1985
5,333445555,22-MAY-1978
6,111111100,15-MAY-1999
7,444444400,15-MAY-1998
8,555555500,01-JAN-1997
```

The following method reads the data present in the file and sets the appropriate object references in the already created employee and department objects.

```
  public static void setManagers(ObjectContainer db)
            throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/manager.dat")) {
      int nMgrs = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nMgrs; i++) {
        String line = fin.readLine();
        String[] fields = line.split(",");
        final int dno = Integer.parseInt(fields[0]);
```

```
        final int essn = Integer.parseInt(fields[1]);
        String startDate = fields[2];
        List<Department> depts = db.query(new Predicate<Department>() {
          public boolean match(Department dept) {
            return (dept.getDnumber() == dno);
          }
        });
        Department d = depts.get(0);
        List<Employee> emps = db.query(new Predicate<Employee>() {
          public boolean match(Employee emp) {
            return (emp.getSsn() == essn);
          }
        });
        Employee e = emps.get(0);
        d.setMgrStartDate(startDate);
        e.setManages(d);
        d.setManager(e);
        db.store(d);
        db.store(e);
      }
    }
  }
```

The method finds the `Employee` object given the social security number and the `Department` object given the department number using native query approach. Then, it sets the appropriate object references in the two objects to point to the other. It also sets the start date field in the `Department` object. Finally, the method makes the changes persistent by calling `store()`.

6.6.8 setControls

This method sets the object references for the one-to-many relationship, controls, between Department and Project. The data file consists of the department number followed by a list of project numbers of projects controlled by the department separated by commas. The data file is shown below:

```
5
1:20
4:10,30
5:1,2,3
6:61,62,63
7:91,92
```

The following method read the data file and sets the object references for the relationship.

```
  public static void setControls(ObjectContainer db)
              throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/controls.dat")) {
      int nControls = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nControls; i++) {
```

```
        String line = fin.readLine();
        String[] fields = line.split(":");
        final int dno = Integer.parseInt(fields[0]);
        String[] projects = fields[1].split(",");
        List<Department> depts = db.query(new Predicate<Department>() {
          public boolean match(Department dept) {
            return (dept.getDnumber() == dno);
          }
        });
        Department d = depts.get(0);
        for (int j = 0; j < projects.length; j++) {
          final int pno = Integer.parseInt(projects[j]);
          List<Project> prjs = db.query(new Predicate<Project>() {
            public boolean match(Project prj) {
              return (prj.getPnumber() == pno);
            }
          });
          Project p = prjs.get(0);
          p.setControlledBy(d);
          db.store(p);
          d.addProject(p); // add p to the "projects" vector of d
        }
        db.store(d);
      }
    }
  }
```

The above code retrieves the Department object for the given department number. Then, for each project number it retrieves the `Project` object and then sets the appropriate object references in both these objects. Finally, the updates are made persistent using the `store()` method.

6.6.9 setWorksFor

This method sets the object references for the one-to-many relationship, `worksFor`, between `Employee` and `Department`. It reads the data from a text file whose first few lines are shown below:

```
6
1:888665555
4:987654321,987987987,999887777
5:123456789,333445555,666884444,453453453
```

The department number is followed by a list of the social security numbers of employees working for the department. The following method reads this data and sets the object references for the relationship.

```
  private static void setWorksFor(ObjectContainer db) throws Exception {
    InputFile fin = new InputFile();
    if (fin.openFile("data/worksFor.dat")) {
```

```
      int nWorksFor = Integer.parseInt(fin.readLine());
      for (int i = 0; i < nWorksFor; i++) {
        String line = fin.readLine();
        String[] fields = line.split(":");
        final int dno = Integer.parseInt(fields[0]);
        String[] emps = fields[1].split(",");
        List<Department> depts = db.query(new Predicate<Department>() {
          public boolean match(Department dept) {
            return (dept.getDnumber() == dno);
          }
        });
        Department d = depts.get(0);
        for (int j = 0; j < emps.length; j++) {
          final int ssn = Integer.parseInt(emps[j]);
          List<Employee> es = db.query(new Predicate<Employee>() {
            public boolean match(Employee emp) {
              return (emp.getSsn() == ssn);
            }
          });
          Employee e = es.get(0);
          e.setWorksFor(d);
          db.store(e);
          d.addEmployee(e); // add e to "employees" vector of d
        }
        db.store(d);
      }
    }
  }
```

This method is identical to the `setControls()` method discussed before.

6.6.10 setSupervisors

This method sets the object references for the one-to-many relationship, `supervisor`, between `Employee` and `Employee`. It reads the data from a text file whose first few lines are shown below:

```
19
888665555:333445555,987654321
333445555:123456789
987654321:999887777,987987987
333445555:666884444,453453453
111111100:111111101,111111102,111111103
222222200:222222201,222222202,222222203
222222201:222222204,222222205
```

The supervisor number is followed by a list of the social security numbers of employees working under the supervisor. The following method reads this data and sets the object references for the relationship.

```
private static void setSupervisors(ObjectContainer db)
                throws Exception {
  InputFile fin = new InputFile();
  if (fin.openFile("data/sups.dat")) {
    int nSups = Integer.parseInt(fin.readLine());
    for (int i = 0; i < nSups; i++) {
      String line = fin.readLine();
      String[] fields = line.split(":");
      final int superssn = Integer.parseInt(fields[0]);
      String[] subs = fields[1].split(",");
      List<Employee> emps = db.query(new Predicate<Employee>() {
        public boolean match(Employee emp) {
          return (emp.getSsn() == superssn);
        }
      });
      Employee s = emps.get(0);
      for (int j = 0; j < subs.length; j++) {
        final int essn = Integer.parseInt(subs[j]);
        List<Employee> subworkers=db.query(new Predicate<Employee>() {
          public boolean match(Employee emp) {
            return (emp.getSsn() == essn);
          }
        });
        Employee e = subworkers.get(0);
        e.setSupervisor(s);
        db.store(e);
        s.addSupervisee(e); // add e to "supervisees" vector of s
      }
      db.store(s);
    }
  }
}
```

This method is identical to the `setControls()` method discussed before.

6.6.11 Complex Retrieval Example

In this example, a complex retrieval is illustrated. Consider the request: *retrieve departments that have a location in Houston or have less than 4 employees or controls a project located in Phoenix*. The code to solve this problem is shown below in its entirety.

```
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.config.Configuration;
import com.db4o.query.Predicate;

import java.io.*;
import java.util.List;
import java.util.Vector;
```

```
public class DisplayDept2 {
  public static void main(String[] args) {
    String DB4OFILENAME = args[0];
    Configuration config = Db4o.configure();
    ObjectContainer db = Db4o.openFile(config,DB4OFILENAME);
    try {
      List<Department> depts = db.query(new Predicate<Department>() {
        public boolean match(Department dept) {
          int nEmps = dept.getEmployees().size();
          Vector<Project> prjs = dept.getProjects();
          boolean foundPhoenix = false;
          for (int i=0; i<prjs.size(); i++) {
            Project p = prjs.get(i);
            if (p.getLocation().equals("Phoenix")) {
              foundPhoenix = true;
              break;
            }
          }
          return dept.getLocations().contains("Houston") ||
                       (nEmps < 4) ||
                       foundPhoenix;
        }
      });
      for (int i=0; i<depts.size(); i++)
        System.out.println("Department: "+depts.get(i));
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
      }
    finally {
      db.close();
    }
  }
```

The main part of this code is the native query predicate specification. The `match` method checks for the number of employees by examining the size of the employees vector in the department object (`dept.getEmployees().size()`). It also check for the "Houston" location of the department by the expression: `dept.getLocations().contains("Houston")`. To check for the location of the projects, a loop is set up for all projects controlled by the department and each project's location is checked. Rest of the code is self-explanatory.

6.7 Web Application

This section introduces methodology to access db4o databases from the Web. The company browser example of Chapter 4 is duplicated in Java with the company database that is already created in this chapter.

6.7.1 Client-Server Configuration

For a Web application to access the db4o database, a client-server configuration of the database is more appropriate. The following class, `Db4oServletContextListener`, is associated with a Servlet Context:

```
import java.io.File;
import javax.servlet.*;
import com.db4o.*;

public class Db4oServletContextListener
                implements ServletContextListener {

  public static final String KEY_DB4O_FILE_NAME = "db4oFileName";
  public static final String KEY_DB4O_SERVER = "db4oServer";
  private ObjectServer server=null;

  public void contextInitialized(ServletContextEvent event) {
    close();
    ServletContext context=event.getServletContext();
    String filePath =
      context.getRealPath("WEB-INF/db/"+
                context.getInitParameter(KEY_DB4O_FILE_NAME));
    server = Db4o.openServer(filePath,0);
    context.setAttribute(KEY_DB4O_SERVER,server);
    context.log("db4o startup on "+filePath);
  }

  public void contextDestroyed(ServletContextEvent event) {
    ServletContext context = event.getServletContext();
    context.removeAttribute(KEY_DB4O_SERVER);
    close();
    context.log("db4o shutdown");
  }

  private void close() {
    if(server!=null
      server.close();
    server=null;
  }
}
```

The `contextInitialized()` method is invoked when the Context is activated; The method creates a db4o server object by associating it with a db4o database available in the `WEB-INF/db` directory of the Web application; the name of the file is available in the `web.xml` file of the Web application as a Context parameter, `db4oFileName`. The server reference is saved in the Context parameter `db4oServer` for other servlets to look up and use. The `contextDestroyed()` method is invoked when the Context is shut down. This method destroys the db4o server object.

The `web.xml` file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
  <web-app>
    <context-param>
      <param-name>db4oFileName</param-name>
      <param-value>company.db4o</param-value>
    </context-param>
    <listener>
      <listener-class>Db4oServletContextListener</listener-class>
    </listener>
    <servlet>
      <servlet-name>DisplayDepartment</servlet-name>
      <servlet-class>DisplayDepartment</servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>DisplayDepartment</servlet-name>
      <url-pattern>/DisplayDepartment</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

The company database browser application consists of the following servlets:

(1) `AllDepartments.java`: This servlet displays all departments in the database in a HTML table format with two columns, one for department number and he other for department name. The department number values are hyper-linked to the `DisplayDepartment` servlet; the department number is sent as a `GET` argument under the name `dno`.
(2) `AllEmployees.java`: This servlet produces a listing of all employees and is similar to `AllDepartments`.
(3) `AllProjects.java`: This servlet poduces a listing of all projects and is similar to `AllDepartments`.
(4) `DisplayDepartment.java`: This servlet accepts the `dno` parameter and produces a detailed listing of all the details of the given department. It lists the name of the department, it's manager name (hyper-linked to employee detail) and start date, a listing of all department locations, a tabular listing of all employees who work for the department with the employee `ssn` hyper-linked to the servlet that produces the employee details, and a listing of all projects controlled by the department with the project number hyper-linked to the servlet that produces the project details.
(5) `DisplayEmployee.java`: This servlet is similar to `DisplayDepartment` and produces a listing of the given employee's details.
(6) `DisplayProject.java`: This servlet is similar to `DisplayDepartment` and produces a listing of the given project details.

The code for the `AllDepartments` servlet is given below:

```
  import com.db4o.*;
  import com.db4o.query.*;
```

```
import java.io.*;
import java.util.List;
import javax.servlet.*;
import javax.servlet.http.*;

public class AllDepartments extends HttpServlet {

  public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException {
  doPost(request,response);
  }

  public void doPost (HttpServletRequest request,
                      HttpServletResponse response)
      throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    ServletContext context = getServletContext();
    ObjectServer server =
        (ObjectServer)context.getAttribute("db4oServer");
    ObjectContainer db = server.openClient();

    out.println("<html>");
    out.println("<head>");
    out.println("<title>All Departments</title>");
    out.println("</head>");
    out.println("<body>");

    out.println("<h2>Departments of Company</h2>");
    out.println("<table border=2>");
    out.println("<tr>");
    out.println("<th>Department Number</th>");
    out.println("<th>Department Name</th>");
    out.println("</tr>");

    try {
      Query query = db.query();
      query.constrain(Department.class);
      query.descend("dnumber").orderAscending();
      ObjectSet results = query.execute();
      while (results.hasNext()) {
        Department d = (Department) results.next();
        out.println("<tr>");
        out.println("<td><a href=\"DisplayDepartment?dno="+
                    d.getDnumber()+"\">"+
                    d.getDnumber()+"</a></td>");
        out.println("<td>"+d.getDname()+"</td>");
        out.println("</tr>");
      }
```

```
      } catch (Exception e) {
          out.println("Exception: " + e.getMessage());
        }

      out.println("</body>");
      out.println("</html>");
      out.close();

      db.close();
    }
  }
```

To connect to the db4o server object, the servlet retrieves the servlet Context using the following statement:

```
ServletContext context = getServletContext();
```

Then, it retrieves the db4o server object that was created at the start of the Context using the statement:

```
ObjectServer server =
        (ObjectServer)context.getAttribute("db4oServer");
```

Finally, the `ObjectContainer` object is obtained using the `openClient()` method as follows:

```
ObjectContainer db = server.openClient();
```

The `ObjectContainer` object is used to perform various database transactions and at the end it is closed. Notice that a SODA query is used to retrieve all the `Department` objects in a sorted manner. Rest of the servlet code is self-explanatory.

The `DisplayDepartment` servlet code is shown next.

```
import com.db4o.*;
import com.db4o.query.Predicate;
import java.io.*;
import java.util.List;
import javax.servlet.*;
import javax.servlet.http.*;

public class DisplayDepartment extends HttpServlet {
  public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doPost(request,response);
  }

  public void doPost (HttpServletRequest request,
```

```
                HttpServletResponse response)
  throws ServletException, IOException {
final int dno = Integer.parseInt(request.getParameter("dno"));

response.setContentType("text/html");
PrintWriter out = response.getWriter();

ServletContext context = getServletContext();
ObjectServer server =
    (ObjectServer)context.getAttribute("db4oServer");
ObjectContainer db = server.openClient();

out.println("<html>");
out.println("<head>");
out.println("<title>Department View</title>");
out.println("</head>");
out.println("<body>");
try {
  List<Department> depts = db.query(new Predicate<Department>() {
    public boolean match(Department dept) {
      return (dept.getDnumber() == dno);
    }
  });

  Department d = depts.get(0);
  out.println("<B>Department: </B>" + d.getDname());
  out.println("<P>Manager: <a href=\"DisplayEmployee?ssn="+
              d.getManager().getSsn()+"\">"+
              d.getManager().getLname()+", "+
              d.getManager().getFname()+"</a></br>");
  out.println("Manager Start Date: "+d.getMgrStartDate());

  out.println("<h4>Department Locations:</h4>");
  for (int i=0; i<d.getLocations().size(); i++)
    out.println(d.getLocations().get(i)+"<BR>");

  out.println("<h4>Employees:</h4>");
  out.println("<table border=2>");
  out.println("<tr>");
  out.println("<th>Employee SSN</th>");
  out.println("<th>First Name</th>");
  out.println("<th>Last Name</th>");
  out.println("</tr>");

  for (int i=0; i<d.getEmployees().size(); i++) {
    Employee e = d.getEmployees().get(i);
    out.println("<tr>");
    out.println("<td><a href=\"DisplayEmployee?ssn="+
                e.getSsn()+"\">"+e.getSsn()+"</a></td>");
    out.println("<td>"+e.getFname()+"</td>");
    out.println("<td>"+e.getLname()+"</td>");
    out.println("</tr>");
```

```
        }
        out.println("</table>");

        out.println("<h4>Projects:</h4>");
        out.println("<table border=2 cellspacing=2 cellpadding=2>");
        out.println("<tr>");
        out.println("<th>Project Number</th>");
        out.println("<th>Project Name</th>");
        out.println("</tr>");

        for (int i=0; i<d.getProjects().size(); i++) {
          Project p = d.getProjects().get(i);
          out.println("<tr>");
          out.println("<td><a href=\"DisplayProject?pno="+
                      p.getPnumber()+"\">"+p.getPnumber()+"</a></td>");
          out.println("<td>"+p.getPname()+"</td>");
          out.println("</tr>");
        }
        out.println("</table>");
      } catch (Exception e) {
          out.println("Exception: " + e.getMessage());
        }

      out.println("</body>");
      out.println("</html>");
      out.close();

      db.close();
    }
}
```

The above code connects to the db4o server and performs a native query to find the department object corresponding to the `dno` value that is sent as a parameter. The details of the department object are then printed. First the department name, manager name and start date are printed, then a list of all employees working for the department are printed in tabular format, and finally a list of all projects controlled by the department is printed in tabular format. All employee `ssn` values as well as project numbers are hyper-linked to the detail pages.

Exercises

NOTE: All references to Exercises and Laboratory Exercises in the following problems refer to the numbering in the Elmasri/Navathe text.

1. Consider the following class definitions (only instance variables are shown) for a Portfolio Management database:

```
public class Security {
  // Attribute Data members
  private String symbol;
```

```
  private String companyName;
  private float currentPrice;
  private float askPrice;
  private float bidPrice;
  // Relationship Data Members
  // Set of transactions for this security
  private Vector<Transaction> transactions;
}

public class Member {
  // Attribute Data members
  private String mid;
  private String password;
  private String fname;
  private String lname;
  private String address;
  private String email;
  private double cashBalance;
  // Relationship Data Members
  // Set of transactions for this member
  private Vector<Transaction> transactions;
}

public class Transaction {
  // Attribute Data members
  private Date transDate;
  private String transType; // Buy or Sell
  private float quantity;
  private float pricePerShare;
  private float commission;
  // Relationship Data Members
  private Member member; // member for this transaction
  private Security security; // security for this transaction
}
```

(a) Write a Java program that reads data from a text file (security.dat) and creates security objects in a db4o database. The first line of the text file contains a number denoting the number of securities that are described. Every subsequent five lines describe one security (symbol, company name, current price, ask price, and bid price). A sample data file follows:

```
3
ORCL
Oracle Corporation
15.75
15.25
15.45
SUNW
Sun Microsystems
14.75
14.25
```

```
14.45
MSFT
Microsoft
55.75
55.25
55.45
```

(b) Write a Java program that implements the following menu of functions on the database:

```
(1) Member Log In
(2) New Member Sign In
(3) Quit
```

A new member may use option (2) to create a new account. This option prompts the new member for the member id, a password, first name, last name, address, email, and initial cash balance. A new member object should be created if no existing member has the same member id, otherwise an error message should be printed. An existing member can use option (1) to login to their account. The member is prompted for the member id and password. Invalid member id or password should result in an error message. Upon successful login, the following menu of options should be presented:

```
(1)  View Portfolio
(2)  Print Monthly Report
(3)  Update Account Data
(4)  Price Quote
(5)  Buy
(6)  Sell
(7)  Quit
```

View Portfolio option should produce a well-formatted report of all current holdings and their values, with a total value displayed at the end. Print Monthly Report should prompt the user for the month and year and a monthly report of all transactions in that month should be displayed. Update Account should prompt user for new values of password, address, and email. Price Quote should prompt the user for a security symbol and then display the current, ask, and bid prices. Buy and Sell should prompt the user for stock symbol and number of shares and perform the action if possible. Otherwise, an error message should be generated.

2. Convert the application described in the previous problem into a Web application with appropriate user interfaces.

3. Consider the GRADEBOOK database described in Exercise 6.36 (Page 193) of the ElMasri/Navathe textbook. In addition to the tables described there, consider the following two additional tables:

```
component(Term,Sec_no,Compname,Maxpoints,Weight)
score(Sid,Term,Sec_no,Compname,Points)
```

The component table records all the grading components for a course offering such as exams, quizzes, home work assignments etc. with each component given a name, a maximum points and a weight (between 0 and 100). The sum of weights of all components for a course offering should normally be 100. The score table records the points awarded to a student for a particular course component for a course offering in which he or she is enrolled.

(a) Design a db4o database schema for the GRADEBOOK database.
(b) Write a Java program that implements the following menu of options for a user (some user interaction is shown – for other menu options you may introduce appropriate user program-user interactions).

```
GRADEBOOK PROGRAM

(1) Add Catalog
(2) Add Course
(3) Add Students
(4) Select Course
(q) Quit

Type in your option: 1
Course Number: 6710
Course Title: Database Systems
Added Catalog Entry

GRADEBOOK PROGRAM

(1) Add Catalog
(2) Add Course
(3) Add Students
(4) Select Course
(q) Quit

Type in your option: 2
Term: sp02
Section Number:
5556
Course Number: 6710
A Cutoff: 90
B Cutoff: 80
C Cutoff: 70
D Cutoff:
60
Course was added! Success!

GRADEBOOK PROGRAM

(1) Add Catalog
(2) Add Course
```

```
(3) Add Students
(4) Select Course
(q) Quit

Type in your option: 3
ID (0 to stop): 1111
Last Name: Jones
First Name: Tony
Middle Initial: A
ID (0 to stop): 2222
Last Name: Smith
First Name: Larry
Middle Initial: B
ID (0 to stop): 0

GRADEBOOK PROGRAM

(1) Add Catalog
(2) Add Course
(3) Add Students
(4) Select Course
(q) Quit

Type in your option: 4
Term: sp02
5556 6710 Database Systems

Select a course line number: 5556

SELECT COURSE SUB-MENU

(1) Add Students to Course
(2) Add Course Components
(3) Add Student Scores
(4) Modify Student Score
(5) Drop Student from Course
(6) Print Course Report
(q) Quit

Type in your option: 1
Student Id (0 to stop): 1111
Student Id (0 to stop): 2222
Student Id (0 to stop): 0

SELECT COURSE SUB-MENU

(1) Add Students to Course
(2) Add Course Components
(3) Add Student Scores
(4) Modify Student Score
(5) Drop Student from Course
(6) Print Course Report
```

```
(q) Quit

Type in your option: q

GRADEBOOK PROGRAM

(1) Add Catalog
(2) Add Course
(3) Add Students
(4) Select Course
(q) Quit

Type in your option: q
```

(c) Write a Java program that reads a text file with the following format

```
sp02
5556
1111 Jones Tony A
2222 Smith Larry B
0000
```

The first line in the text file contains the term, the second line contains the Sec_no and subsequent lines contain the student id, last name, first name, and middle initial per line for each student enrolled in the course. The program should enroll the students in the course. If the student object already exists, only the enrollment should take place, however, if the student object does not exist, it must be created and then the enrollment should take place. You may assume that the course object already exists.

4. Convert the application described in the previous problem into a Web application with appropriate user interfaces.

5. Consider the following class definitions for a geographical database of states and cities of the United States:

```
public class State {
  // Attribute Data members
  private String stateCode;
  private String stateName;
  private String nickname;
  private int population;
  // Relationship Data Members
  // Capital city
  private City capitalCity;
  // Set of cities
  private Vector<City> cities;
}

public class City {
  // Attribute Data members
```

```
  private String cityCode;
  private String cityName;
  // Relationship Data Members
  // State city lelongs to
  private State state;
}
```

(a) Write a Java program to read data from a text file containing data about states and cities and populate the db4o database. The format of the text file is:

```
50
GA:Georgia:Peach State:6478216
Atlanta(ATL),Columbus(CLB),Savannah(SVN),Macon(MCN)
IL:Illinois:Prairie State:12128370
Springfield(SPI),Bloomington(BMI),Chicago(ORD),Peoria(PIA)
…
…
```

(b) Write a program in Java which implements the following menu-based system:

```
MAIN MENU

(1)    Given the first letter of a state, print all states along
   with their capital city names.
(2)    Print the top 5 populated states in descending order of
   population.
(3)    Given a state name, list all information about that state,
   including capital city, population, state nickname, major
   cities, etc. The report should be formatted nicely.
(4)    Print an alphabetical count of number of states for each
   letter of the English alphabet. The list should be nicely
   formatted; 4 letter counts to a   line. The count is the
   number of states whose names begin with the letter.
(5)    Quit
```

6. Convert the program of part (b) of the previous problem into a Web application with appropriate user interfaces.