**README**

**FBN Flight Reservation System  
Version 1.0**

**Contents**

* [Introduction](http://www.reocities.com/SiliconValley/software/1664/scjd/readme.html#introduction)
* [Supported JDK version and platforms](http://www.reocities.com/SiliconValley/software/1664/scjd/readme.html#versionpaltform)
* [Installation](http://www.reocities.com/SiliconValley/software/1664/scjd/readme.html#install)
* [Running the server](http://www.reocities.com/SiliconValley/software/1664/scjd/readme.html#runserver)
* [Running the client](http://www.reocities.com/SiliconValley/software/1664/scjd/readme.html#runclient)
* [Location of db.db file](http://www.reocities.com/SiliconValley/software/1664/scjd/readme.html#locationofdb.db)
* [Location of the Design Choices document](http://www.reocities.com/SiliconValley/software/1664/scjd/readme.html#locationofdesign)
* [List of files](http://www.reocities.com/SiliconValley/software/1664/scjd/readme.html#contents)[submitted](http://www.reocities.com/SiliconValley/software/1664/scjd/readme.html#listoffiles)

**Introduction**

FBN Flight Reservation system (version 1.0) is designed to keep scheduled flight information for a number of airlines and routes, and to provide an easy way for searching for and booking flights that might meet customer's needs.

This reservation tool consists of a client system with graphical user interface (GUI) to search for and book flights, and a server system to handle multiple connected clients.

**Supported JDK version and platforms**

FBN Flight Reservation System is implemented in Java 2 SDK, Standard Edition, version 1.3.0. All the files were created on Windows 2000, and have been tested on Windows 2000, Windows NT 4.0 and Solaris 7. FBN system can be safely run on Java 2 (version 1.3.0) platform.

NOTE: To make the instructions more general, I will provide instructions for 1) WIN32 systems, including Windows NT 4.0 and Windows 2000; and 2) UNIX platforms, including Solaris, HP-UX and AIX. I assume that you are using csh on UNIX.

**Installation**

Before installing FBN Flight Reservation System, you need to have installed Java 2 (version 1.3.0) Runtime Environment properly on your computer.

When you see this readme document, you may have extracted the submitted jar file to an installation directory. In all the following examples, I suppose the installation directory is C:\scjd on WIN32 or /home/scott/scjd/ on UNIX. You can use any other directory, but make sure that no space is in the directory path name.

In your installation directory, you should find three sub-directories: client, server and starting.

In "client" and "server" directories, you can find all the required files for running the client and server system respectively. "starting" directory has all the files for API documentation, origin source code, modified and new source code, and compiled class files, etc.

NOTE: In the submission, for simplicity, I provide a java.policy file that gives global permission to anyone from anywhere. This policy file may be changed according to the requirements in a real product environment.

Similar to other client/server systems, FBN system requires that the server should be properly started before you can run the GUI client to search for or book flights. In the following two sections, you can find the detailed instructions for running the server and client.

**Running the server**

First, start RMI registry. The RMI registry is a naming repository that allows remote clients to get service from the server. Note that before starting the rmiregistry, you must make sure CLASSPATH environment variable is not set or is empty in your current shell or window.

To start the registry on the server, execute the rmiregistry command in your current directory. This command produces no output and is typically run in the background.

**Win32** (use javaw if start is not available):

set CLASSPATH=

start rmiregistry

**UNIX**:

unsetenv CLASSPATH

rmiregistry &

By default, the registry runs on port 1099. To start the registry on a different port, specify the port number (usually greater than 1024) on the command line. For example,

**Win32**:

start rmiregistry 1100

**UNIX**:

rmiregistry 1100 &

Second, start the server.

To run the server, you need to go to "server" directory, where you can find the server program -- FBNServer.jar and other supported files. Usually you run rmiregistry and the server on the same machine, which can be done by using the java.rmi.server.codebase property. The usage of the server program is:

java -Djava.rmi.server.codebase=URL\_of\_FBNServer\_jar\_File/FBNServer.jar  
       -Djava.security.policy= java.policy  
       -jar FBNServer.jar  
       Database\_FileName Host\_Name Port\_Number.

For example, if you run the server on a local machine, the command line is:

**Win32**:

java -Djava.rmi.server.codebase=file:/c:\scjd\server\FBNServer.jar

-Djava.security.policy=java.policy

-jar FBNServer.jar

db.db localhost 1100

**UNIX**:

java -Djava.rmi.server.codebase=file:/home/scott/scjd/server/FBNServer.jar

-Djava.security.policy=java.policy

-jarFBNServer.jar

db.db ite.gmu.edu 1100

If the code base specifies a URL, you need a web server installed and put the FBNServer in the corresponding directory. For example, in my PC, I place FBNServer.jar to C:\oracle\ora81\Apache\Apache\htdocs\scjd. In a UNIX machine, I put FBNServer.jar to a public\_html directory. The command line will become::

**Win32**:

java -Djava.rmi.server.codebase=http://localhost/scjd/server/FBNServer.jar

-Djava.security.policy=java.policy

-jar C:\oracle\ora81\Apache\Apache\htdocs\scjd\FBNServer.jar

db.db localhost 1100.

**UNIX**:

java -Djava.rmi.server.codebase=http://ite.gmu.edu/~scott/scjd/server/FBNServer.jar

-Djava.security.policy=java.policy

-jar /home/scott/public\_html/scjd/FBNServer.jar

db.db ite.gmu.edu 1100.

The preceding command defines several properties and arguments.

* The java.rmi.server.codebase property specifies the location, a code base URL, of classes originating from this server so that class information for objects sent to other virtual machines will include the this class's location thus a receiver can load it.
* The java.security.policy property is used to specify the policy file that contains the permissions you intend to grant specific code bases.
* FBNServer.jar is the server program including all the server classes.
* Database\_FileName is the database file name including path. In the above example, db.db is in the current directory. If you want to connect to a database in any other directory, you need to provide the path for it. For example, ../db.db is used to connect to db.db database in the upper directory. You may get a java.lang.OutOfMemoryError if the given file is in a wrong format.
* Host\_Name is your machine's Domain Name System (DNS) name. In a WIN32 system, you can use "localhost".
* Port\_Number is the port number on which the registry accepts calls from the clients.

The server's stub class is dynamically loaded into a client's virtual machine only when the class is not already available locally *and* the codebase has been set properly.

IMPORTANT: Make sure that no two running servers connect to the same database.

**Running the client**

Once the registry and the sever are running, you can start the client. The client connects to the database, either in local mode or in network-based mode. To run the client, you need to go to "client" directory, which is under your installation directory.

**In local mode**, you need only to give the database file name. The usage of the client program in local mode is:

java -jar FBNClient.jar Database\_FileName.

For example,

**Win32**:

set CLASSPATH=

java -jar FBNClient.jar db.db

**UNIX**:

unsetenv CLASSPATH

java -jar FBNClient.jar db.db

The preceding command defines several properties and arguments.

* FBNClient.jar is the client program including the data client and GUI client.
* Database\_FileName is the database file name including path. In the above example, db.db is in the current directory. If you want to connect to a database in any other directory, you need to provide the path for it. For example, ../db.db is used to connect to db.db database in the upper directory. You may get a java.lang.OutOfMemoryError if the given file is in a wrong format.

**In network-based mode**, you need to give the policy file name, the host name and port number of the server. The usage of the client program in network-based mode is:

java -Djava.rmi.server.codebase=URL\_of\_FBNServer\_jar\_File/FBNServer.jar  
       -Djava.security.policy= java.policy  
       -jar FBNClient.jar Host\_Name Port\_Number.

For example, if you run the client and the server on the same machine, the command line is:

**Win32**:

set CLASSPATH=

java -Djava.rmi.server.codebase=file:/c:\scjd\server\FBNServer.jar

-Djava.security.policy=java.policy

-jar FBNClient.jar

localhost 1100

**UNIX**:

unsetenv CLASSPATH

java -Djava.rmi.server.codebase=file:/home/scott/scjd/server/FBNServer.jar

-Djava.security.policy=java.policy

-jar FBNClient.jar

ite.gmu.edu 1100

If you are running the client and the server on different machines, the code base is usually a URL. Given the same example as that of the server, the command line will become:

**Win32**:

set CLASSPATH=

java -Djava.rmi.server.codebase=http://localhost/scjd/server/FBNServer.jar

-Djava.security.policy=java.policy

-jar FBNClient.jar

localhost 1100.

**UNIX**:

unsetenv CLASSPATH

java -Djava.rmi.server.codebase=http://ite.gmu.edu/~scott/scjd/server/FBNServer.jar

-Djava.security.policy=java.policy

-jar FBNClient.jar

ite.gmu.edu 1100.

The preceding command defines several properties and arguments.

* The java.rmi.server.codebase property specifies the location, a code base URL, of classes originating from this server so that class information for objects sent to other virtual machines will include the this class's location thus a receiver can load it.
* The java.security.policy property, used to specify the policy file that you intend to grant specific permissions for code from various sources.
* FBNClient.jar is the client program including the data client and GUI client.
* Host\_Name is the Domain Name System (DNS) name of your FBN server. In a WIN32 system, you can use "localhost" if you run the server and the client on the same machine.
* Port\_Number is the port number on which the registry accepts calls from the clients.

Note that the client's codebase property should be the same as the server's. If you run the client and the server on different machines, you may need an HTTP server installed on the server and the codebase specified should start with "http://".

After starting the client, you should be able to see a window with title "FBN Flight Reservation System" and an initial screen showing all the flights information. In this window, there are three menus: Connection, Action and Help.

Connection and its sub-menus are used to get connection Info or to exit; Action and its sub-menus can be used to search for flights and book seats; Help and its sub-menu provides information about this product.

**Searching for Flights**:

To search for the flights, you can select Action/Search menu, then a small window with title "Flight Search" will pop up. In this window, you will see two editable dropdown lists ( "Origin airport", "Destination airport") and two buttons ("Search", "Close").

You can select the origin and destination of flights with the two dropdown lists. A wildcard-like feature is provided with "\*" standing for any value of the origin, or the destination or both. The same effect can be achieved by leaving one of the fields or both empty. For example, if you choose "\*" from the "Origin airport" dropdown list and choose "SFO" as the destination, then after clicking "Search" button, you will find the info on all the flights arriving to "SFO" airport in the main screen. Note that the double quote (") should not be included when you make your selections.

The search window will automatically disappear if the search is successful, otherwise you get a message telling you no flight found.

**Booking Seats**:

To book the flights, you can choose Action/Book menu, then a small window with title "Book Seat" will pop up. This window contain an editable dropdown lists- "Flight number", an editable text field- "Number of seat(s)", and two buttons- "Book" and "Close".

You can select the flight number with the dropdown list. The "Number of seats" field can only accept a number equal to or greater than 1. For example, if you want to book one ticket for flight "BA002", you'll need to choose "BAO02" from the "Flight number" dropdown list and type 1 in the "Number of seat(s)" field, then click "Book" button and wait for processing.

If you have made a successful reservation, the book window will automatically disappear, meanwhile, a reservation summary window  will be displayed. Otherwise, you get a message about what is going wrong, such as, the flight number is unknown- "Unkown Flight" or there is no enough seats available- "No enough seats".

IMPORTANT: Make sure that no two running clients connect to the same database.

**Location of db.db file**

Two copies of db.db are provided. One is located in client sub-directory, the other is in server sub-directory.

**Location of Design Choices document**

[design.html](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html) -- the Design Choices document is located in the installation directory, which is the same directory as this document's.

**List of files submitted**

After unpacking the submitted jar file, you can find the documents, client/server programs, copy of original code, new source code, compiled classes, API document in your current directory.

This section contains a summary of the files and directories in the FBN system. The location and a brief description of each file are also provided.

Once again, to simplify the description, I assume the FBN system is installed at c:\scjd directory.

**README and design choices document:  
          --**in the root directory (c:\scjd)

README.txt -- a plain text file that describe the installation, execution, and file list.  
design.html     -- the design choices document in HTML format.  
readme.html   -- readme in HTML format.

**Client System:**

**--**in the client subdirectory (c:\scjd\client)

FBNClient.jar -- the executable jar file of the client system.  
db.db             -- the database file supplied by the assignment.  
java.policy      -- the policy file used when the client runs in network-based mode.

**Server System:   
          --**in the server subdirectory (c:\scjd\server)

FBNServer.jar -- the executable jar file of the server system.  
db.db               -- the database file supplied by the assignment.  
java.policy        -- the policy file for running the server.

**Compilation Directory:  
          --**in the starting\classes subdirectory (c:\scjd\starting\classes)

build.bat                     -- the batch file to compile and create the jar files (FBNClient.jar and FBNServer.jar).  
ManifestClientFile.txt  -- the manifest file to specify the Main-Class for FBNClient.jar.  
ManifestServerFile.txt -- the manifest file to specify the Main-Class for FBNServer.jar.

All the compiled class files under suncertify subdirectory are created by simply running build.bat on Windows 2000.

**API Documentation:  
          --**in the starting\docs subdirectory (c:\scjd\starting\docs)

api\index.html -- the starting page of the API document (that is, C:\scjd\starting\docs\api\index.html).

The API document is generated by running the following command in c:\scjd\starting directory. Note that you need the internet access to view JDK 1.3 API document.

          javadoc -d ./docs/api  
             -doctitle "FBN Flight Reservation System, V 1.0 <br> API Specification"  
             -sourcepath . -overview overview.html  
             -link <http://java.sun.com/products/jdk/1.3/docs/api>    
                          suncertify.db suncertify.dbclient suncertify.dbserver suncertify.guiclient suncertify.guiclient.command

**Original Copy of the Supplied Classes and Files:  
          --**in the starting\origincopy subdirectory (c:\scjd\starting\origincopy)

instructions.html       -- the instruction and requirement.

suncertify\db\ \*.java -- the source code for the four classes: Data, DatabaseException, DataInfo, and FieldInfo.  
suncertify\db\\*.class -- the compiled byte code for the four classes: Data, DatabaseException, DataInfo, and FieldInfo.  
suncertify\db\db.db  -- the database file supplied by the assignment.

**Source Code:**

**--**in the starting\suncertify subdirectory (c:\scjd\starting\suncertify)

overview.html -- the overview file for the API specification.

|  |  |
| --- | --- |
| *files in c:\scjd\starting\suncertify\db* *and their descriptions* | |
| Data.java DatabaseConstants.java DatabaseException.java DataInfo.java DataIntf.java FieldInfo.java Lock.java package.html | modified class to provide the basic database services. new interface to define the constants for the database. supplied class for the database access failure. supplied class to embody the record info. new interface to define the public interface of Data. supplied class to embody the field info. new class to lock database record. description of suncertify.db package. Related to database. |

|  |  |
| --- | --- |
| *files in c:\scjd\starting\suncertify\dbclient and their descriptions* | |
| ConnectionInfo.java DataClient.java NetworkData.java package.html | new class to embody the connection info. new class to handle the networking and the database access. new class to define a network-based version of Data class. description of suncertify.db package. Related to data client. |

|  |  |
| --- | --- |
| *files in c:\scjd\starting\suncertify\dbserver and their descriptions* | |
| DataServer.java DataServerConstants.java FBNServer.java RemoteDataServerIntf.java ReservationIntf.java package.html | new class to define the data server. new interface to define the constants of the data server. new class that is the main class of the server. new class to define the public interface of the data server. new interface to specify the public interface for booking. description of suncertify.dbserver package. Related to server. |

|  |  |
| --- | --- |
| *files in c:\scjd\starting\suncertify\guiclient and their descriptions* | |
| CommandAction.java DataTableCellRenderer.java DataTableModel.java FBNClient.java GUIClient.java MouseHandler.java SystemConstants.java package.html | new class to decouple the command's logic from action. new class to define a cell renderer. new class to show search result in JTable. new class that is the main class of the client. new class for the main frame of the GUI client. new class to show the description of menu items. new interface to define system constants. description of suncertify.guiclient package. Related to GUI. |

|  |  |
| --- | --- |
| *files in c:\scjd\starting\suncertify\guiclient\command and their descriptions* | |
| AboutCommand.java BookCommand.java CommandIntf.java ConnectInfoCommand.java ExitCommand.java SearchCommand.java package.html | new class to show product info. new class to book seats. new interface implemented by the command classes. new class to show the current connection info. new class to exit the client system. new class to search for flights. description of suncertify.guiclient.command package.   Related to book/search. |

**Directory tree structure:**

c:\scjd

The root directory of the FBN system installation. Contains README and design document.

c:\scjd\client

The working directory of the client system. Contains an jar file, a database file and a policy file.

c:\scjd\server

The working directory of the server system. Contains an jar file, a database file and a policy file.

c:\scjd\starting

The root directory of the FBN development environment. Contains overview.html and four sub-directories: classes, docs, orgincopy and suncertify.

c:\scjd\starting\classes

The compilation directory. Contains a batch file, two manifest files for creating the jar files, and a subdirectory which contains all the compiled classes.

c:\scjd\starting\docs

The directory of HTML formatted API document. C:\scjd\starting\docs\api\index.html is the start page.

c:\scjd\starting\origincopy

The directory to store the supplied instruction, classes and db.db.

c:\scjd\starting\suncertify

The source code directory of the whole system. Contains five java packages in the FBN system: suncertify.db,  suncertify.dbclient, suncertify.dbserver, suncertify.guiclient, suncertify.guiclient.command. For further information, see the design document -- design.html.

The Java 2 SDK, Standard Edition, is a product of Sun MicrosystemsTM, Inc. Windows and Windows NT are trademarks of Microsoft Corporation.

Copyright © 1997-2001, Flight By Night Services.

**Design Choices Document**

**FBN Flight Reservation System  
Version 1.0**

**Contents**

* [Introduction](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#introduction)
* [Architecture](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#architecture)
* [Network approaches](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#network)
* [Data class modification](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#data)
* [Record locking](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#recordlock)
* [The data server](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#server)
* [The data client](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#client)
* [The GUI client](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#gui)
* [Error handling](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#error)
* [Code standards](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#code)
* [Design patterns](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#pattern)
* [Deliverables](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#deliverables)
* [Summary](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#summary)

**Introduction**

This document describes my major design choices and provides the reasons for those choices. All the design choices are based on the specifications and requirements of the developer's assignment. To design and implement the system, I keep several import rules in mind:

(1) Make it easy to use - The system should be easy to start and use.  
(2) Keep it simple         - The design should be simple, clear and easy to implement.  
(3) Keep it right            - The design should satisfy the requirements.  
  
The FBN flight reservation system is designed to keep scheduled flight information, and to provide a convenient way for searching for and booking  flights. The main development platform is Java 2 SDK, Standard Edition, version 1.3.0 on Windows 2000.

**Architecture**

The application is a traditional client-server system. From a user's point of view, it includes a client with graphical user interface, and a server handling multiple connected clients. From a developer's point of view, it has three key parts:

(1) The data server    - the server-side database with network server functionality  
(2) The data client     - the client-side database client part that handles the networking for the GUI  
(3) The GUI client     - the client-side graphical user interface

The data server exposes the same public methods as Data class and implements the flight booking with record locking. The data client is the layer between the data server and GUI, and makes the networking connection, database access including record locking, and business logic transparent to GUI client. The GUI client deals with the user interface only.

The system can work in a local or a network-based mode.

In the local mode, the data client accesses a local database, and bypasses the networking entirely.  
In the network-based mode, the data client accesses a remote database by using RMI.

The user is able to choose the connection mode in the command line. To keep the system simple, the program is designed to stay in one mode once it has started up.

To implement the system, five java packages are created to organize the classes:

(1) suncertify.db                             - package related to the database  
(2) suncertify.dbclient                     - package related to data client  
(3) suncertify.dbserver                    - package related to data server  
(4) suncertify.guiclient                     - package related to GUI client  
(5) suncertify.guiclient.command     - package related to GUI client event handling.

**Network approaches**

***RMI is chosen over socket as the approach to network connection.***

Although the socket approach gives the programmer more control over the communication, RMI is a better choice for this particular application for the following reasons:

First of all, RMI is a pure Java solution, so using it can let developers enjoy all the benefits from Java and avoid dealing with any other network protocols or outside systems. Besides, it allows the Object Oriented paradigm to extend over the network, which can lead to a cleaner designs

Secondly, RMI based system is easier to implement, debug, and requires less code, since RMI takes care of the network communication, object serialization, exception handling specific to network error, security management, etc.  All of this can let the developer easily concentrate on implementing the business requirement, and make the implementation easier to be understood and maintained by other developers.

Finally, RMI is a good frame work for a distributed architecture. It simplifies the deployment and maintenance of distributed systems by supporting dynamic-loading and object serialization over network.

**Data class modification**

This section covers the design decisions about the Data class enhancement. I decide to do the following: modify Data class, correct the deprecated methods, and implement criteriaFind(String), lock (int) and unlock (int) methods. Since the implementation of lock/unlock involves creating a new class - Lock, its design will be covered in section [Record locking](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#recordlock).

To make it consistent, the API document/comment of suncertify.db package has been revised slightly. For example, in  find() method, "the key field is the record number field" is changed into "the key field is the first field".

ASSUMPTION: the database file (db.db) can be accessed by only one client or one server. A database can not be concurrently accessed by more than one application.

***Data class is enhanced by modification.***

One reason to choose modification instead of subclassing is that in the specification, Data class is not used in any released product, and the changing would not have impact on other systems.

Besides, because the existing Data class is an incomplete implementation, subclassing it would lead to maintain two classes instead of one.

One more consideration is that object composition (encapsulation) is usually preferred to inheritance.

***Implementing*DataIntf *interface.***

Data class is retrofitted to implement DataIntf which defines all the public methods of Data class. This will significantly simplify the data client implementation and have no impact on Data class. More detailed explanation can be found in the [data client](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#client)design.

***Correction of the deprecated methods.***

Two methods in Data class are deprecated in Java 2 platform.

(1) The String constructor used in readRecord() is deprecated.

The related statement:           "rv[i]=new String(buffer,0,offset,description[i].getLength());"  
The String constructor used:   public String(byte[] ascii, int hibyte, int offset, int count);

*According to Java 2 API document, this constructor does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a character-encoding name or that use the platform's default encoding.*So I choose the String constructor that uses the platform's default encoding. The corrected constructor and statement become:

The correct String constructor:  public String(byte[] bytes, int offset, int length);  
The corrected statement:          "rv[i] = new String(buffer, offset, description[i].getLength());"

(2) The String method - getBytes() used in writeRecord(String[]) is deprecated.

The related statement:            "newData[i].getBytes(0, toCopy, buffer, offset);"  
The String method used:          public void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin);

*According to Java 2 API document, this method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the getBytes(String enc) method, which takes a character-encoding name, or the getBytes()method, which uses the platform's default encoding.*So I choose the String's getBytes() method using the platform's default encoding. The corrected method and statement become:

The correct getBytes method:  public public byte[] getBytes();  
The corrected statement:        "System.arraycopy(newData[i].getBytes(), 0, buffer, offset, toCopy);"

***Implementation of flight search -*public DataInfo[] criteriaFind(String criteria)*.***

This method searches the database for entries matching the criteria supplied. Criteria take the form of a comma separated list of <field name>=<value to match> specifications. For example, the following argument string would select all records describing flights by the SpeedyAir carrier that departs from San Francisco.

"Carrier='SpeedyAir',Origin airport='SFO'"

According to the requirement, DataInfo[] criteriaFind(String criteria) is designed to behave like this:

    If criteria is null or an empty string (""), this method returns all the records.  -- "no criteria"  
    Otherwise, if criteria is a string without "=" in it, this method returns null.         -- "invalid search criteria"  
    Otherwise, if there is an invalid field-name in criteria, this method returns null. -- "invalid field-name"  
    Otherwise, if there is no matched record, this method returns null.                        -- "no records found"  
    Otherwise, this method returns array of DataInfo objects                                     -- "records found"

The matching is case-sensitive and only exact matches are supported.

Note that the white spaces at the front and the end of the criteria string, the white spaces around the single quotes('), and the white spaces around the equal character("=") are trimmed and ignored. The the field names and values read from the database are also trimmed to allow the user to define the criteria easily. For example, "Carrier='SpeedyAir'" won't match any record unless the white spaces in the field name read from the database are removed, or the white spaces are appended to "Carrier" in the criteria. I chose to trim since this is an easier solution.

criteriaFind() is synchronized to keep Data class thread-safe. The synchronization does come at a cost of performance and increases the lock contention. Using synchronized statement blocks in criteriaFind() could reduce the lock contention among the threads and thus improve the responding times. Considering the rule to keep it simple, I just synchronized the method.

The algorithm complexity of criteriaFind() is O(cn), where c is the total number of fields; n is the number of records in the database.

To avoid NullPointerException being thrown, the null criteria is assigned the meaning of "no criteria" -- asking for all the records.

To implement the method criteriaFind(), I use a Hashtable to store (field-index, value-to-match) pairs constructed from the criteria string. For each record read from database, the method performs the field value matching for every field/value pair in the Hashtable. If a record matches all the field/value pairs in the Hashtable, the record is added into a Vector object. At the end of the method, if no record found, the method returns null, otherwise, it calls toArray(Object[]) in Vector to create and return an array of DataInfo objects.

To construct the above Hashtable, I add a private method -- Hashtable parseCriteriaHashtable(String criteria) in Data class.  This method parses the criteria string and returns a Hashtable object whose key/value pair is the field index and the corresponding field value parsed from the criteria. The key is an Integer, and the value is a String. I use Hashtable since it is synchronized and thus can keep Data class thread-safe. StringTokenizer is heavily used to simplify the parser implementation.

**Record locking**

Record locking can be done by calling lock(int)and unlock(int)of Data class. Lock/unlock are implemented to allow concurrent use of the database when multiple clients are booking flights. Note that lock/unlock are not synchronized.

Record locking makes the flight booking safe. If the sequence: *lock->read->modify->unlock*, is always followed whenever the clients attempt to modify the database , the *read* just after the *lock* is always correct for the lock owner since no one can*modify* the database without a *lock*.

lock/unlock is usually needed only in the network mode. In the local mode, since only one client is running against a database, the record locking gets nothing but cost. Besides, considering the GUI and the business logic are usually separated, the record locking which is used to implement the business logic, such as flight booking, should be completely hidden from GUI.

***Lock and unlock***

I implement the lock/unlock by creating a helper class -- Lock. This can avoid the synchronized statement and separate the detailed locking implement from Data class.

To support the whole database locking, lock takes an argument (-1) to lock all the records one by one.

Lock method blocks until the requested lock can be applied, without timeout, as required in the specification. The details can be found in the [Lock class](http://www.reocities.com/SiliconValley/software/1664/scjd/design.html#lock class) design.

The signature of lock/unlock is kept unchanged. The specification requires that unlock() should ignore the unlock request from a non-lock-owner. This requirement seems to ask for a clientID in the lock/unlock signatures. Actually, my first implementation changed the signatures of lock/unlock by adding clientID. In the final version, I implement all the business logic (such as booking seats) on the data server in the network mode, so lock/unlock are kept on the server. By  this way, I can save the signatures of lock/unlock, and make the booking executed effectively on the server, thereby improve the system performance. Even if a user of this Data class really wants to use lock/unlock method in the data client,  it can still handle the concurrent booking by following the call sequence: *lock -> read -> modify -> unlock*.

In lock(int) and unlock(int), Thread.currentThread() is used as a locker ID to guarantee that unlocking a lock in another thread should be ignored, although this can cause a little cost.

lock() throws an IOException if the caller requests the lock with an invalid record number. Any non-positive number other than -1 is an invalid record number for the record locking.

unlock(int) simply removes the lock for the record. It is usually called in a *finally* block to ensure the record is unlocked even if an exception is thrown.

**Lock** ***class***

This class works as a lock handler, and is used to lock records in a database.

Lock class has a Hashtable to hold the locks. The (key, value) pair of the Hashtable is (record number, the record's lock-owner). The key is an Integer, and the value is an Object.

All the methods in Lock class are synchronized to keep the class thread-safe.

lock() calls wait() to block the caller when the request record is already locked. unlock() calls notifyAll() to wake up all the threads that are waiting on this Lock object's monitor. The awakened threads will compete in the usual manner. Note that only one thread can get the monitor. A thread can get a record lock only if it gets the monitor and its requested record is not locked by anyone else.

Considering that the server usually could handle the requests fast enough, no queue or timeout is implemented to handle multiple client requests. This can keep the implementation simple. Note that a queue makes the server behave more friendly when the number of clients is huge.

To make the recording safe, a second locking request for the same record without unlock is ignored if the request is from the same lock holder.

unlockAll() is added to unlock the whole database in case of unexpected exception.

To avoid the deadlock, it is recommended that each newly requested record number from a lock holder should be always larger than any locked record numbers held by this lock holder.

Note that the record locking implementation still needs the class user to follow the call sequence: *lock -> read -> modify -> unlock*.

**The data server**

The data server accepts multiple concurrent network connections and allows them to interrogate and manipulate the database. The data server is designed to be thread-safe.

To achieve the requirements, I define an interface RemoteDataServerIntf and implement it in DataServer. The main class -- FBNServer accepts the user command line and starts the data server.  DataServerConstants interface defines the constants, such as the data server name to register in RMI registry.

I also choose to implement all the business logics on the server because this can reduce the network traffic and make the system more robust. Besides, lock/unlock is used to implement the flight booking on the server. Since the server handle the record locking, it can always release the lock even if the client crashes for some reason. Finally, I don't need to change the signatures of lock/unlock to identify which client is locking the record. Just as in the real life, the data server handles the data integrity.

ASSUMPTION: no two servers concurrently connect to the same database.

**RemoteDataServerIntf *interface.***

RemoteDataServerIntfinterface extends Remote, DataIntf and ReservationIntf interfaces. Remote interface is required to implement the remote data server, while DataIntf defines the same public methods as the Data class, and ReservationIntf defines the constants and the method for flight booking.

**DataServer *class.***

DataServer is a UnicastRemoteObject and implements RemoteDataServerIntf. It holds a DataIntf object, which can make the server connect to a local or remote database. In the assignment, the data server can only connect to a local database. I intentionally make it work this way since there is no requirement in the specification to support the remote database on the server.

* DataServer is an adapter for a DataIntf object, and exposes the same public methods as the Data class.
* DataServer is also an adapter for a Reservation object, and exposes bookSeats() method of Reservation class.

DataServer is not implemented as a singleton, because I think the data server should not be restricted to only one database connection. The data server should be able to support multiple database connections.

Note that DataServer, DataInfo, FieldInfo, and String are all serializable, which is required for the RMI framework.

**Reservation *class and the flight booking.***

Flight booking is implemented in Reservation class. The booking method is "void bookSeats(String flightNum, int numSeat)", which is defined in ReservationIntf interface.

The idea to add Reservation class comes from the thought of keeping the business logic, such as flight booking, on the sever. Since the client can work either in a local or a network mode, I also need to implement a local version (without lock/unlock) of booking method if I implement the flight booking in DataServer class. To reduce the redundant code and keep Data class as lean as possible, the booking method is implemented in an individual class -- Reservation.

Two constructors are provided for Reservation class, depending on whether the record locking is needed or not.

The main procedure in bookSeats(String flightNum, int numSeat) can be described as following:

    If flightNum is null or can't be found in the database, it throws "Unknown flight" exception.  
  
    In network-based mode:  
        lock the record  
        read the record again (the record may be modified just before the locking)  
  
    If there are enough seats available, it modifies the record in database, otherwise, it throws "No enough seats" exception.  
  
    In network-based mode:  
        unlock the record in a finally block if the record is locked in this method.

**The Main class -- FBNServer*.***

FBNServer processes the command line arguments and starts the data server.

This class creates only one DataServer object and rebinds the data server name to that object. Any existing binding for the name is replaced.

**Logging*.***

To make the data server more user friendly, I let the data server print brief transaction information to the screen, such as searching criteria, booking flight number and number of seats. The user can redirect the output to a specific file.

**Limitation*.***

There are some important features in a real products, but they are not required in the specification. To keep it simple, I did not implement them. Examples of these features are: server administration tools/utilities, transaction rollback, reservation cancellation,  non-directly flight reservation and so on.

**The data client**

The data client is the bridge between the data server and the GUI client. It hides all the business logic and network handling for the GUI. The GUI only knows the data client, and does not even need to know where the database or the data server is.

The data client can work in either local or network-based mode. If a database filename is provided, it works in the local mode; if a host name and a port number are provided, it works in the network-based mode. Different constructors in DataClient class are used to support the two modes.

NetworkData is the class implementing the same public methods as Data class.

NOTE: DataIntf is implemented by both NetworkData and Data class.

**DataClient *class.***

DataClient holds a DataIntf object, a ReservationIntf object, a TreeSet of airport codes, a TreeSet of flight numbers, and a ConnectionInfo object. These objects are all created in the constructor.

In the local mode, DataClient accepts a database file name, it creates a Data object, and a Reservation object without record locking. Then it constructs the airport codes and flight numbers from the database. At the last step of the constructor, it constructs a ConnectionInfo object, which embodies the database connection information, such as database file name, host name, port number and connection mode.

In the network-based mode, DataClient accepts a host name and a port number. It creates a RMISecurityManager first if necessary, gets a ReservationIntf object by looking up the registry, and constructs a NetworkData object. Then it follows the same steps as in the local mode to construct the airport codes, flight numbers, and a ConnectionInfo object. Note that the returned reference from the registry lookup is of data type ReservationIntf since DataServer implements ReservationIntf interface.

TreeSet is used to store the airport codes and flight numbers since it is a sorted Set. So the user can get a sorted list in the GUI.

Since the airport codes and flight numbers are read from the database, the GUI does not need to hard code them. This makes the system more flexible. There may be some performance issues when the database becomes very large, but it is still a good choice for a small application like this. In a real product, the airport codes and flight numbers should be stored in some specific database tables, and can be retrieved in the same way.

**The GUI client**

The GUI client is designed to be easy to use. The expectation of future enhancements is also considered in the design.

All the components in the system are from Swing components in Java 2.

**Overall picture*.***

The system has one main window to show the main data output, such as the search result. To allow the user to run the system, a MenuBar is provided with three menus: Connection, Action and Help.

* Connection and its sub-menus are used to get the connection information or exit the system.
* Action and its sub-menus are used to search for flights and book seats
* Help and its sub-menu are used to get product information.

Menu mnemonics is implemented to support keyboard shortcuts.

Since the flight search and booking are the main functionalities, I create two small windows for them.

In the flight search window, two editable dropdown lists (JComboBox) allow the user to choose/enter the origin and destination airports; two buttons allow the user to start search or close the window. I use JComboBox because it allows the user to select from a list instead of typing in the airport codes, flight numbers. Besides, the user can always enter a new airport code if necessary since the JComboBox is editable.

In the flight booking window, there are one editable dropdown list for inputting the flight number, a text field for inputting the number of seats, and two buttons for starting the flight booking and closing the window.

The GUI client can work in a local mode or a remote mode, depending on what arguments are given in the command line. Once the GUI is started, the user can not change the connection mode, but can show the connection info.

**Layout and JTable*.***

The main window and the two windows for flight search and booking are JFrame's; the About  window and the reservation summary window are JDialog's; the connection info window is a JOPtionPane wrapping a JTable.

BorderLayout, BoxLayout, and GridLayout are used in the system. BorderLayout is the layout managers for the windows that keep the data output in the "CENTER", show the status in the "SOUTH", and display the user input, such as the search criteria in the "NORTH". BoxLayout and GridLayout are only used to layout the components, such as buttons and labels, in an inner container. This strategy can keep the user interface consistent even if the application needs functionality enhancements or extensions.

The search result is shown in the main windows by using JTable. In the initial screen,  the GUI client shows all the flights so that the user knows what is in the database. I choose to show all the flights since the database is small.

I extend AbstractTableModel to implement the data model for showing search result in the JTable. The reason I choose AbstractTableModel instead of DefaultTableModel is because the search result is the main data output, thus the data model for it should be flexible. In the implementation class, all the field values are trimmed to show the results nicely.

DefaultTableCellRenderer is extended to implement a cell renderer to align the columns. I re-aligned the columns because the number/date/time columns are better right aligned. So the key column (column 1) is center aligned, and all others are right aligned.

Besides, the connection info and reservation summary are also displayed in JTable. Since the data shown in these two windows are very simple. I use DefaultTableModel to make the JTable uneditable.

Additionally, JOptionPane is used whenever possible to show simple messages, such as "no records found" or "unknown flight number".

**Swing and multi-threading*.***

In the GUI, I create worker threads for the flight searching and booking. So the screen will not be blocked when the client is waiting for the response from the server. To avoid confusion, I disallow the user to send a second searching request before the client gets the response from the server. This feature is implemented by disabling the search button in the search window. A similar strategy is used in the flight booking window.

Besides, SwingUtilities.invokeLater(Runnable) is used to guarantee that only the event thread should change the screen appearance.

**Searching for flights*.***

To search for the flights, the user needs to select from Action/Search menu.

A small window with title "Flight Search" is used for setting up searching criteria. In the search window,  there are two editable dropdown lists ( "Origin airport", "Destination airport") and two buttons ("Search", "Close").

The user can select the origin and destination of flights with the two dropdown lists. A wildcard-like feature is provided with "\*" standing for any value of the origin, or the destination or both. The same effect can be achieved by leaving one of the fields or both empty. The wildcard-like feature is implemented in the GUI client only, since a simple process can filter the search criteria string before calling criteriaFind(String criteria) in Data class. For example, "Origin airport='\*',Destination airport='ATL'" can be filtered to "Destination airport='ATL'" since the two criteria are essentially the same.

Note that the double quote (") should not be included when user makes their selections. The search window will automatically disappear if the search is successful, otherwise user will get a message showing no flight found.

**Booking seats*.***

To book the flights, the user needs to select from Action/Book menu.

A small window with title "Book Seat" is the place to choose the flight number and number of seats to book. In the booking window, there is an editable dropdown lists - "Flight number", an editable text field - "Number of seat(s)", and two buttons- "Book" and "Close".

The user can select the flight number with the dropdown list. The "Number of seats" field can only accept an integer with value 1 or larger. An inner class - NumSeatVerifier (extends InputVerifier) is used to verify the "Number of seats" input.

If the user has made a successful reservation, a reservation summary window will be displayed. Otherwise, the user will get a message about what is going wrong, such as, "Unkown Flight" or "No enough seats".

**Error handling**

In most cases, the exceptions are thrown all the way up to the client by using throws clauses. This makes the code clean and the exception handling easy to manage.

In some situations, if a method can handle or do something for an exception, I let the method catch the exception and react.

In the data client and data server, messages are printed on the screen. In the GUI client, the message boxes are provided whenever necessary.

**Code standards**

Java Code Conventions from Sun are followed to implement the system.

Besides, the constants are classified into class/object level constants and package level constants. The class/object level constants are declared as *private final* variables in classes; and the package level constants are declared as public constants in interfaces.

The naming conventions used is very similar to the conventions given in Java Code Conventions.

* The interfaces are always ended with "Intf" or "Constants". An interface whose name ends with "Intf" defines or inherits at least one abstract method; an interface whose name ends with "Constants" defines constants only.

**Design patterns**

The standard Java package facilities are used wherever possible.

Adapter, Factory and Command patterns are used.

One of the classes that apply Adapter pattern is NetworkData class. NetworkData is an adapter class, which wraps a reference to the remote object and implements the same public methods as the Data class. I make Data and Network implement the same interface -- DataIntf, so I can use DataIntf as a common interface to support the local and remote database connections.

Factory pattern is applied in DataClient class whose two constructors allow it to support local mode and network-based mode.

Command pattern is applied in the GUI implementation. The whole package -- suncertify.guiclient.common is an implementation of command pattern, which separates the control logic or action from the user interface. Therefore, the system can be easily configured to use different UI elements without having to rewrite the control or callback logic. Note that Action interface in the Swing also provides a mechanism to detach the action from the UI. By applying the command pattern, it is possible to decouple the command and the action and therefore, the control logic can be further separated from the UI.

Most importantly, Java 2 platform itself contains lots of built-in patterns. A typical example is the Observer pattern used in JTable, whereby the data are represented by the model and the view by the visual component. To keep the system clean and simple, I always follow the patterns used in Java 2 whenever possible.

**Deliverables**

I create two jar files to reduce the number of the deliverables for the user. FBNServer.jar is for the server, and FBNClient.jar is for the client.

**FBNServer.jar**

FBNServer.jar is created to make the server starting procedure easier. I specify the Main-Class as suncertify.dbserver.FBNServer in it, so the user does not need to provide the class name to run it.

jar -cvmf ManifestServerFile.txt FBNServer.jar suncertify/db suncertify/dbserver

is the command line to create the jar file.

**FBNClient.jar**

FBNClient.jar is created to make the client starting procedure easily. Again, I specify the Main-Class as suncertify.guiclient.FBNClient in it, so the user can run it without providing the class name.

jar -cvmf ManifestClientFile.txt FBNClient.jar  
     suncertify/db  
     suncertify/dbserver/RemoteDataServerIntf.class  
     suncertify/dbserver/ReservationIntf.class suncertify/dbserver/Reservation.class  
     suncertify/dbclient suncertify/guiclient

is the command line to create the jar file.

The stub class is not included in the client jar file since it can be dynamically downloaded from the server.

**java.policy**

The following is the content of the policy file used in the system.

grant {  
    permission java.security.AllPermission;  
};

It grants all the permissions for the reason of simplicity. In a real product, this file may be changed according to the requirements.

**Summary**

To summarize, the major design choices for the FBN flight reservation systems are:

* Use RMI over serialized object as the approach to the network connection
* Modify Data class and let it implement DataIntf
* criteriaFind(String) is implemented with a parsing method, which parses the criteria by using StringTokenizer
* Record locking is implemented with a helper class - Lock; the whole database locking is also implemented
* Booking is implemented in the server
* Data client hides the networking, business logic and lock/unlock from the GUI
* AbstractTableModel is extended for the search result; DefaultTableModel is used for the booking summary
* JComboBox's are used for the airport code and the flight number selection
* Multi-threading is used in the GUI for searching and booking