# Radical Refactoring

*Let refactoring tools do the walking for you*

Eugene Belyaev, Maxim Shafirov, and Ann Oreshnikova

According to Martin Fowler, "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs." (*Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 2000). Clearly, refactoring is something all developers do, even when you hand-code and don't realize you're actually doing it. Despite its association with Extreme Programming (XP) and other radical coding methods, refactoring is a basic process.

When you edit code to make it more rational, portable, modular, or readable, much of that process almost always consists of applying refactorings—systematic changes that satisfy Fowler's definition. Refactoring can be as straightforward as renaming a method or variable, then updating all related references across the entire project so it doesn't break the build. Even in simple examples, few of us would refactor completely by hand—we're more likely to use automated search-and-replace tools, whether in GUI dialogs or via grep and sed.

*Eugene is president and CTO of JetBrains, Maxim is a senior developer for the IntelliJ IDEA Java IDE development project, and Ann is head of the Documentation and Support Department. They can be contacted at http://www.jetbrains.com/.*

Unfortunately, basic search-and-replace tools cannot ensure consistent modifications, since they lack the context-sensitive logic needed to determine the function of a particular bit of code. This means that you must verify all changes by hand. Even with simple refactorings, such tedium can discourage refactoring. Many developers have bemoaned the state of a large project as poorly structured, badly implemented, unscalable, and kludged—but still too complex to fix within the current budget or timeframe.

Refactoring becomes more powerful if you combine individual refactorings into a series. Together, they can help you accomplish complicated tasks that, at first glance, may seem impossible to automate and nearly as difficult to do by hand. In this article, we show how refactoring—treated as a process integral to development itself—is as frequently needed and crucial to projects as the initial coding itself.

### When Refactoring Is Needed

Refactoring can dramatically increase development efficiency during four main stages of software development:

1. Architectural/design decisions and design changes.
2. Reusing existing code.
3. Global changes that entail repetitive tasks, such as switching to newer APIs or eliminating deprecated APIs.
4. Everyday coding.

To illustrate, we use IntelliJ IDEA 3.0, a tool our company (Jetbrains Inc.) produces. Other tools may use different commands and results may vary. However, what's important is the sequence of refactorings.

### Refactoring Scenarios

Imagine a generic client-server application with its server portion originally designed on a single-threaded model. At some point, the number of client users grows rapidly and long delays occur before requests get handled. Switching to a multithreaded server model would solve the problem—but the original code made no provision for the required synchronized locks.

In Listing One, a Singleton server object *UserManagement* provides access to an *AccessRightsTable*, which allows read access to user rights. Listing One is the initial state of the *UserManagement* class, while Listing Two is the initial state of the *AccessRightsTable* class. The *Client* class, among others, has two methods that refer to *UserManagement*'s and *AccessRightsTable*'s methods; see Listing Three, the initial state of the *Client* class.

To switch to a multithreaded model, you need both synchronized (safe but slow) and nonsynchronized (fast but unsafe) versions of the class *AccessRightsTable.* At the same time, you need to ensure *copyTable*—returned by the *createImmutableCopy()* method—remains immutable.

You can accomplish this complex task through a series of refactorings and smart edits—just apply the following refactorings in order, which will change the application design without affecting the consistency of the existing code.

**Step 1.** Open the AccessRights Table.java file and choose Refactor|Extract Interface (Figure 1). Specify *AbstractAccessRightsTable* as the name for the new interface and select all methods from the list to be declared in it. The program creates a new interface with the methods declared, then

*(continued from page 26)*
automatically changes usages of *AccessRightsTable* to *AbstractAccessRightsTable* everywhere relevant in your project.

**Step 2.** Using Alt+Insert (or the New popup menu item) in Project view, create the new classes *MutableAccessRightsTable* and *ImmutableAccessRightsTable* to implement *AbstractAccessRightsTable*. In both classes, declare the *myDelegate* field of type *AbstractAccessRightsTable* and call the Code|Delegate Methods main menu item. Choose the *myDelegate* field and select all methods from the list to be delegated to this field.

**Step 3.** In *MutableAccessRightsTable*, go to each generated method body in sequence, highlight it, and press Ctrl+Alt+T to call the Surround With lookup list. Then select *synchronized{}* to automatically enclose the method body in this block (Figure 2).

**Step 4.** In *ImmutableAccessRightsTable*, go to the methods that allow write access to the object and type *throw new UnsupportedOperationException("<Your exception text here>")* instead of the method body.

**Step 5.** Search for use of the *AccessRightsTable* constructor, using Alt+F7, and decide where its mutable or immutable version should be used. In this case, you need the mutable one in the *UserManagement* Singleton class's constructor, and the immutable wrapper in the return statement of the *createImmutableCopy()* method of the *Client* class.

Refactoring tools make this process easy—these steps took us only seven minutes. The refactorings produce Listings Four through Six and guarantee synchronized access to *AccessRightsTable* as well as the immutability of the copy created for read-only use. (By contrast, see the accompanying text box entitled "Doing It the Hard Way" for an equivalent solution done by hand.)

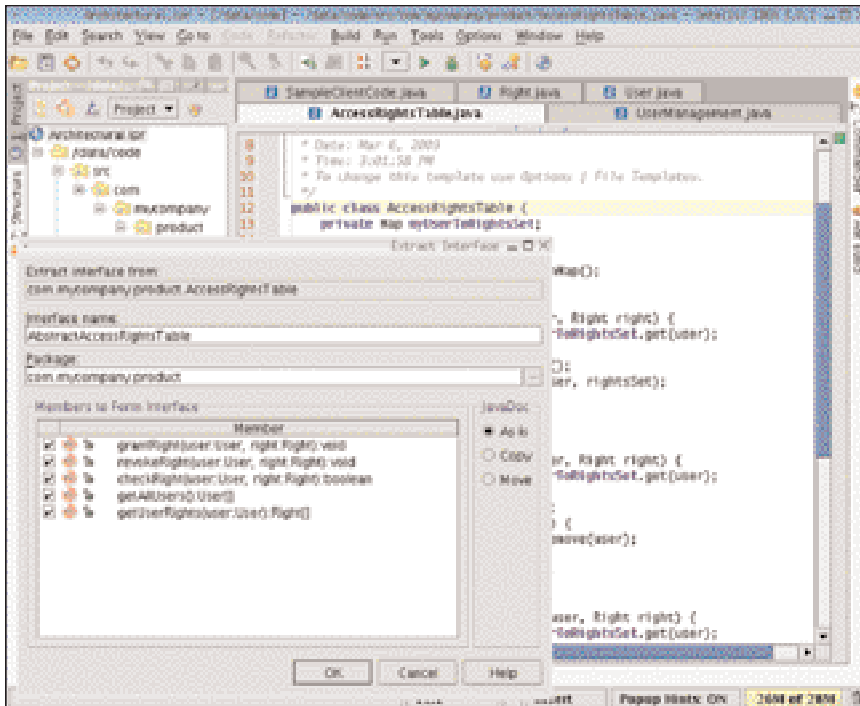

***Figure 1:*** *Use the Extract Interface dialog in IntelliJ IDEA to create an abstracted* AccessRightsTable *class.*

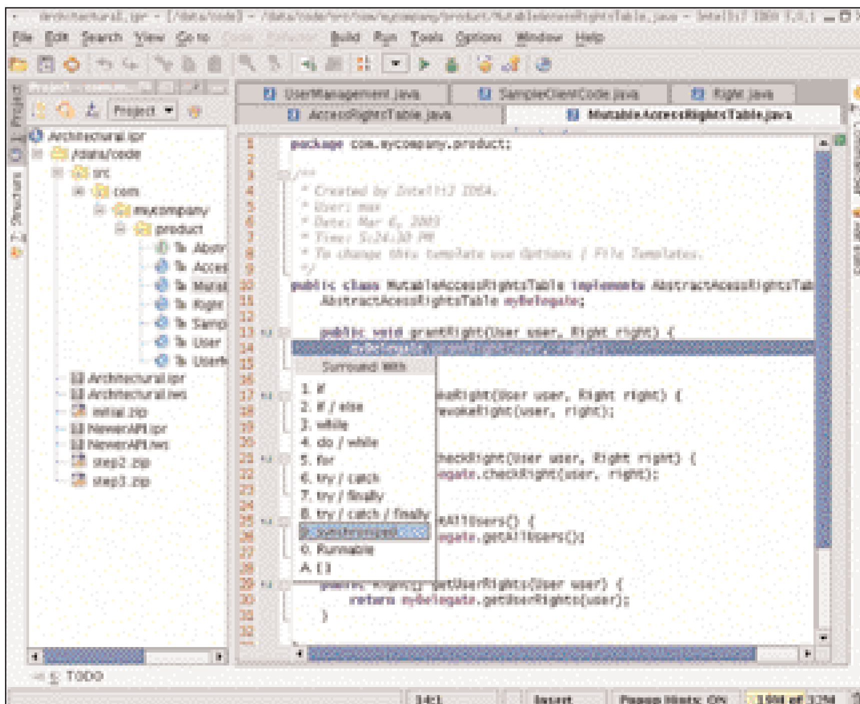

***Figure 2:*** *Use the Surround With lookup list to enclose the bodies of the generated methods with* synchronized{} *blocks.*

## Reuse of Existing Code

Most of us have needed to reuse code, whether our own or someone else's. We now present a real-world example that illustrates how refactoring makes code reuse much easier.

Imagine in your application a dialog window with two interacting components. It might be a tree-like list of options and a preview pane that is automatically modified at runtime to reflect the options selected or deselected. The dialog was written long ago by someone else. Now you need to create a similar dialog to work with alternative data. Your first approach might be to create a copy of the existing class containing the requisite dialog code, then simply make multiple corrections so that the new code reflects your new requirements. This may work, but the approach makes no provision for you (or your colleagues) to integrate another, similar dialog or multiple dialogs in the future.

Refactoring provides a better solution by generalizing the design for all such dialogs, while requiring only minimal corrections on your part to perform manually.

1. Open the existing class for the dialog.
2. Find the code block that creates the pane with the tree-like list of options and perform the Extract Method refactoring on it.
3. Do the same for the code block responsible for the preview pane.
4. Call the Extract Superclass refactoring, select all members to be moved to the superclass, and mark the two newly created methods to be declared abstract.

5. Create a new class for your new dialog, extending the newly created superclass.

Now you have an identical dialog and only need to define the two methods that accept your data. Better yet, in future uses, the superclass becomes a base for creating additional dialogs of the same type.

We often use such refactorings to produce GUI modules. This keeps the UI consistent and guards against possible bugs that can creep into the code when using a copy/paste approach.

### Boring and Repetitive Tasks

A few years ago, like many other developers, we frequently utilized the class *java.util.Vector* in our Java applications. In large applications, such as our IntelliJ IDEA Java IDE, this class could be used in thousands of places. The main problem was the synchronized behavior of this class, but there was no way to avoid using it until the nonsynchronized *ArrayList* class appeared in the new Java SDK API, as a substitute for *java.util.Vector*. This new class improved application performance, so we decided to switch to this newer API. Doing so manually would take hundreds of man hours, but there was no obvious way to automate the task. After analyzing several approaches, however, we found we could use our own tool's refactoring support to make sequenced, automated changes and save an incalculable amount of time and money.

Here is how we used refactoring to switch to a newer API (or eliminate a deprecated API). Listing Seven uses *java.util.Vector* and its methods. To switch to the *ArrayList*, we only needed to:

1. Create a new class *Vector* in one of the project's packages and make it extend *ArrayList*.
2. Define a method lacking in *ArrayList* but available in *java.util.Vector*; see Listing Eight. (For similar tasks, you need to redefine all differing methods.)
3. Migrate the project globally from *java.util.Vector* to your newly created *Vector*, calling the migration utility with the Tools|Migrate… command (Figure 3).
4. In the new *Vector* class, call the Refactor Inline… command for each method (in the example, *elementAt()*). Listing Nine shows the resulting changes in the *SampleClass*.
5. Finally, migrate the project globally from *Vector* to *java.util.ArrayList*.

Bingo! We switched from *java.util.Vector* to *java.util.ArrayList* with no remaining inconsistency. We could then easily delete your custom *Vector* class as unnecessary (Listing Ten).

### Refactoring as a Third Hand

The previous scenario illustrated how refactoring serves as an extra mechanism that lets you alter and improve application design. If you have access to advanced refactoring tools, you quickly find that individual refactorings are applicable to a wider spectrum of common development procedures. No one on our team could imagine their everyday coding tasks without the use of sophisticated refactoring tools. In the next scenario, we show you the potential of some particular refactoring procedures in everyday production to help develop streamlined and efficient code.

The *Sample* class (the fragment between delimiters; Listing Eleven) includes a block of code that selects users with read-access rights. Say you want to create a new method that selects users with particular rights. In IntelliJ IDEA, you select the fragment between delimiters and press Ctrl+Alt+M, or call the Refactor|Extract Method command. You then specify a new method name (*getUsersWithRight*, for example), press OK, and it becomes Listing Twelve.

For a truly generic new method, avoid using the constant variable (READ_ACCESS), and instead pass the value as a parameter. To do so, select the field (within the *if* condition; Listing Twelve) and press Ctrl+Alt+P, or call the Refactor|Introduce Parameter command. We specify the new parameter name as *pRight* and the field is now passed as a parameter (the fragment between delimiters; Listing Thirteen).

## Doing It the Hard Way

While the refactoring sequence in the first scenario produces an extremely efficient redesign of existing code, the operations themselves are highly repetitive. Unless you have an unending supply of unpaid, volunteer programmers, you will want to use automation tools, such as a refactoring-capable IDE. In contrast, let's look at the painful steps needed to perform the same necessary operations manually:

**Stage 1.** Manually create an interface to be implemented by the *AccessRightsTable*, as well as by the manually created *MutableAccessRightsTable* and *ImmutableAccessRightsTable*. Manually create all methods in the two new classes and make them use the delegate field in the method body. Manually wrap methods in *MutableAccessRightsTable* with the *synchronized{}* block. After having spent, say, half an hour or more at that, you would take a coffee break (another 10–15 minutes) and come back with a clear mind to tackle the second stage—involving work so boring, you do not even want to think about it.

**Stage 2**. Find all usage of the *AccessRightsTable* class in your project (field types, variable types, parameter types, method return types). For each usage, decide whether to manually switch to the *AbstractAccessRightsTable* interface instead of the class. Find all usages of the *AccessRightsTable* constructor in your project. Analyze the contextual role of the variable or field holding an instance of *AccessRightsTable* and try to foresee possible future usages of each particular instance, to either leave it as-is or wrap it up with the *MutableAccessRightsTable* or *ImmutableAccessRightsTable* constructor calls.

Depending on the size of your project, these steps might take you anywhere from one to dozens of man hours to complete. By contrast, the right automation tool gets it done in seven minutes.

—*E.B., M.S., and A.O.*

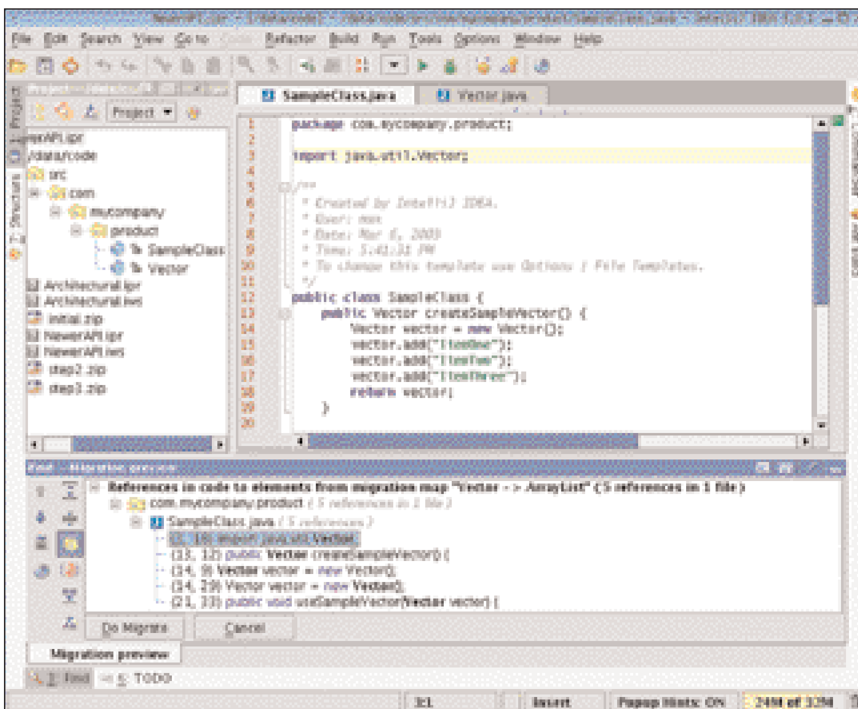

***Figure 3:*** *Migrating from* java.util.Vector *to* com.mycompany.product.Vector.

The two variables of different scopes having the same name (*rightfulUsers*) reduce the code's readability and, in larger projects, we would probably rename one of these variables (Refactor|Rename…). In this scenario, you do better to just inline the variable used in *arrangeUsersByRight()* (by calling the Refactor|Inline… command), so it becomes:

```
myUsersWithRights =
        getUsersWithRight(READ_ACCESS);
```

### Conclusion

Automated refactoring tools are incredibly powerful and deserve a place in every developer's arsenal. For complex code modification on large source trees, automated refactoring is especially useful. However, by staying aware of both the process of refactoring and the tools that make that process easier, refactoring's usefulness can extend to all stages of project development. Used properly, refactoring makes even the most routine development processes much more efficient.

**DDJ**

### Listing One

```
public class UserManagement {
   private static UserManagement ourSingleInstance;
   private AccessRightsTable myRightsTable;
   private UserManagement() {
      myRightsTable = new AccessRightsTable();
   }
   public static UserManagement getInstance() {
      if (ourSingleInstance == null) {
         ourSingleInstance = new UserManagement();
      }
      return ourSingleInstance;
   }
   public AccessRightsTable getRightsTable() {
      return myRightsTable;
   }
}
```

### Listing Two

```
public class AccessRightsTable {
    private Map myUserToRightsSet;
    public AccessRigtsTable() {
        myUserToRightsSet = newHashMap();
    }
    public void grantRight(User user, Right right){
        Set rightsSet = (Set) myUserToRightsSet.get(user);
        if(rightsSet == null) {
            rightsSet = new HashSet ();
            myUserToRightsSet.put(user, rightsSet);
        }
        rightsSet.add(right);
    }
    public void revokeRight(User user, Right right) {...}
    public Boolean checkRight(User user, Right right) {...}
    public User[] getAllUsers() {...}
    public Right[] getUserRights(User user) {...}
}
```

### Listing Three

```
public class Client {
   /** This method revokes CREATE_NEW_ACCOUNT right from all users
    * currently registered in rights table.
    */
   public static void revokeCreateNewAccount() {
      AccessRightsTable rightsTable =
```

```
                           UserManagement.getInstance().getRightsTable();
      User[] allUsers = rightsTable.getAllUsers();
      for (int i = 0; i < allUsers.length; i++) {
         User user = allUsers[i];
         rightsTable.revokeRight(user, Right.CREATE_NEW_ACCOUNT);
      }
   }
   /** This method creates new AccessRightsTable copy. This table is not
    * intended to be modified. @return new copy table. NOT TO BE MODIFIED!
    */
   public static AccessRightsTable createImmutableCopy() {
      AccessRightsTable rightsTable =
                            UserManagement.getInstance().getRightsTable();
      AccessRightsTable copyTable = new AccessRightsTable();
      User[] allUsers = rightsTable.getAllUsers();
      for (int i = 0; i < allUsers.length; i++) {
         User user = allUsers[i];
         Right[] rights = rightsTable.getUserRights(user);
         for (int j = 0; j < rights.length; j++) {
            Right right = rights[j];
            copyTable.grantRight(user, right);
         }
   }
      return copyTable;
      }
}
```

## Listing Four

```
public class UserManagement {
   private static UserManagement ourSingleInstance;
   private AbstractAccessRightsTable myRightsTable;
   private UserManagement() {
      myRightsTable = new MutableAccessRightsTable(new AccessRightsTable());
   }
   public static UserManagement getInstance() {
      if(ourSingleInstance == null) {
         ourSingleInstance = new UserManagement();
      }
      return ourSingleInstance;
   }
   public AbstractAccessRightsTable getRightsTable() {
      return myRightsTable;
   }
}
```

## Listing Five

```
public class MutableAccessRightsTable implements AbstractAccessRightsTable {
   private AbstractAccessRightsTable myDelegate;
   public MutableAccessRightsTable(AbstractAccessRightsTable pDelegate) {
      myDelegate = pDelegate;
   }
   public void grantRight(User user, Right right) {
      synchronized (this) {
         myDelegate.grantRight(user, right);
      }
   }
   public void revokeRight(User user, Right right) {
      synchronized (this) {
          myDelegate.revokeRight(user, right);
      }
   }
}
```

## Listing Six

```
public class Client {
   public static void revokeCreateNewAccount() {...}
   public static AbstractAccessRightsTable createImmutableCopy() {
      AbstractAccessRightsTable rightsTable =
                              UserManagement.getInstance().getRightsTable();
      AbstractAccessRightsTable copyTable = new AccessRightsTable();
      User[] allUsers = rightsTable.getAllUsers();
      for (int i = 0; i < allUsers.length; i++) {
         User user = allUsers[i];
         Right[] rights = rightsTable.getUserRights(user);
         for (int j = 0; j < rights.length; j++) {
            Right right = rights[j];
            copyTable.grantRight(user, right);
         }
     }
     return new ImmutableAccessRightsTable(copyTable);
  }
}
```

## Listing Seven

```
public class SampleClass {
   public Vector createSampleVector() {
      Vector vector = new Vector();
      vector.add("ItemOne");
      vector.add("ItemTwo");
      vector.add("ItemThree");
      return vector;
   }
   public void useSampleVector(Vector vector) {
      for (int i = 0; i < vector.size(); i++) {
         String s = (String) vector.elementAt(i);
         System.out.println("s = " + s);
      }
   }
}
```

## Listing Eight

```
public class Vector extends ArrayList {
   public Object elementAt (int index) {
      return get(index);
   }
}
```

## Listing Nine

```
public class SampleClass {
   public Vector createSampleVector() {
      Vector vector = new Vector();
      vector.add("ItemOne");
      vector.add("ItemTwo");
      vector.add("ItemThree");
      return vector;
   }
   public void useSampleVector(Vector vector) {
      for (int i = 0; i < vector.size(); i++) {
         String s = (String) vector.get(i);
         System.out.println("s = " + s);
      }
   }
}
```

## Listing Ten

```
public class SampleClass {
   public ArrayList createSampleVector() {
      ArrayList vector = new ArrayList();
      vector.add("ItemOne");
      vector.add("ItemTwo");
      vector.add("ItemThree");
      return vector;
   }
   public void useSampleVector(ArrayList vector) {
      for (int i = 0; i < vector.size(); i++) {
         String s = (String) vector.get(i);
         System.out.println("s = " + s);
      }
   }
}
```

## Listing Eleven

```
public class Sample {
   static final int READ_ACCESS = 1;
   static final int WRITE = 2;
   ArrayList myUsers;
   ArrayList myUsersWithRights;
   public void arrangeUsersByRight() {
      ... <some code here> ...
      ArrayList rightfulUsers = new ArrayList();
      for (int i = 0; i < myUsers.size(); i++) {
         User user = (User) myUsers.get(i);
         if ((user.myAccessRight & READ_ACCESS) != 0) {
            rightfulUsers.add(user);
         }
    }
     myUsersWithRights = rightfulUsers;
     ... <some code here> ...
   }
}
```

## Listing Twelve

```
public void arrangeUsersByRight() {
      ... <some code here> ...
      ArrayList rightfulUsers = getUsersWithRight();
      myUsersWithRights = rightfulUsers;
      ... <some code here> ...
}
private ArrayList getUsersWithRight() {
      ArrayList rightfulUsers = new ArrayList();
      for (int i=0; i < myUsers.size(); i++) {
            User user = (User) myUsers.get(i);
            if ((user.myAccessRight & READ_ACCESS) != 0) {
                  rightfulUsers.add(user);
            }
      }
      return rightfulUsers;
}
```

## Listing Thirteen

```
public void arrangeUsersByRight() {
      ...<some code here>...
      ArrayList rightfulUsers = getUsersWithRight(READ_ACCESS);
      myUsersWithRights = rightfulUsers;
      ...<some code here>...
}
private ArrayList getUsersWithRight(int pRight) {
      ArrayList rightfulUsers = new ArrayList();
      for (int i = 0; i < myUsers.size(); i++) {
            User user = (User) myUsers.get(i);
            if ((user.myAccessRight & pRight) !=0) {
                  rightfulUsers.add(user);
            }
      }
      return rightfulUsers;
}
```

**DDJ**