"Medical Diagnosis" Case Study
by Michael Clancy © 2007

Background

The problem of diagnosing diseases and recommending treatments is a difficult one because of the enormous number of combinations of symptoms and diseases possible. An experienced diagnostician will not only be able to associate possible diseases with a patient's symptoms, but will be able to remove diseases from consideration for which required symptoms aren't present. Code to be designed in this case study imitates this aspect of medical diagnosis, providing a sort of "physician's assistant".

Problem

Define a Java class named MedicalDataBase, together with appropriate JUnit test classes. A MedicalDataBase object will maintain three collections of information:

- symptom information that specifies, for each symptom recognized by the program, a set of possible diseases that the symptom might indicate;
- disease information that specifies, for each disease recognized by the program, a set of necessary symptoms that must accompany the disease;
- a set of symptoms displayed by a patient.

The MedicalDataBase constructor will read the first two collections of information from BufferedReader objects. (The format is explained below.) It will also initialize the patient's symptom set to empty. Two methods will be used to collect the patient's symptoms:

- void addSymptom (String symptom) throws UnrecognizedSymptomException
  adds the given symptom to those already provided for the patient. UnrecognizedSymptomException is thrown if the symptom does not appear in the symptom information.
- void reset ( )
  sets the patient's symptom set to empty, for example, to deal with a different patient.

Two more methods will provide information about a diagnosis.

- String [ ] diagnosis ( )
  returns a list of diseases consistent with the symptoms submitted thus far (since the most recent call to reset) for the patient. It finds the union of all possible diseases for the given symptoms, then from that union returns the diseases whose necessary symptoms have all been reported for the patient. The array returned by diagnosis should not contain any duplicate diseases.

- `String [ ] missingSymptoms (String disease)`
  is used to explore details of why a given disease was not diagnosed. It returns one of the following:
  a. `null` if the disease does not appear among the possible diseases indicated by the symptoms provided thus far;
  b. a zero-element array of symptoms if the disease is consistent with the symptoms provided thus far; or
  c. a nonempty array of symptoms missing from the set of required symptoms for the disease. This list should not contain any duplicate symptoms.

The `diagnosis` and `missingSymptoms` methods should respond quickly, even with a large data base.

The `MedicalDataBase` constructor is given two `BufferedReader` arguments, one for the symptom information, the other for the disease information. Each argument represents a sequence of lines (accessed by `readLine`); the formats of the two sequences are similar. Each line of the symptom information sequence contains the name of a symptom followed by one or more names of diseases that the symptom might indicate. (Each symptom name and disease name is a single word.) Each line of the disease information sequence contains the name of a disease followed by zero or more names of symptoms that must accompany that disease. The constructor may assume that all this information is correctly specified.

Examples

Consider the following data base.

symptom	*possible diseases*
fever	ague plague
chill	ague plague
shivering	ague
buboes	plague
shortness_of_breath	all-nighter caffeine_overdose exercise
sleepiness	all-nighter boring_lecturer

disease	*necessary symptoms*
plague	buboes
ague	fever chill
all-nighter	
caffeine_overdose	
exercise	shortness_of_breath
boring_lecturer	

The table below gives a sequence of calls to the various MedicalDataBase methods and the result of each call.

call	*result*
`addSymptom ("double_vision");`	throws UnrecognizedSymptomException since "double_vision" is not among the symptoms in the symptom information
`addSymptom ("fever");`	
`diagnosis ( )`	returns an empty array, since both the diseases suggested by "fever" require other symptoms as well
`missingSymptoms ("flu")`	returns null since "flu" (an unknown disease) isn't among the possible diseases for "fever"
`missingSymptoms ("all-nighter")`	returns null
`missingSymptoms ("plague")`	returns a one-element array containing "buboes"
`missingSymptoms ("ague")`	returns a one-element array containing "chill"
`addSymptom ("chill")`	

call	*result*
`diagnosis ( )`	returns a one-element array containing "ague"
`missingSymptoms ("all-nighter")`	returns null
`missingSymptoms ("plague")`	returns a one-element array containing "buboes"
`missingSymptoms ("ague")`	returns an empty array
`addSymptom ("shortness_of_breath")`	
`diagnosis ( )`	returns a four-element array containing "ague", "all-nighter", "caffeine_overdose", and "exercise"
`missingSymptoms ("all-nighter")`	returns an empty array
`missingSymptoms ("plague")`	returns a one-element array containing "buboes"
`missingSymptoms ("ague")`	returns an empty array
`missingSymptoms ("exercise")`	returns an empty array

Preparation

Code in this study uses ArrayLists and arrays, exceptions, input and output from files, the Scanner class, exceptions, multi-class programs, and JUnit.

Exercises

Analysis 1. Explain the difference between null and an empty array.

Analysis, Reflection 2. The problem statement says that "the diagnosis and missingSymptoms methods should respond quickly, even with a large data base." Supply more detail for that specification.

Reflection 3. What aspects of the problem statement are easiest to understand, and what parts are most difficult? Briefly explain your answer.

Analysis 4. What does a call to diagnosis return immediately after the program segment below?

```
reset ( );
addSymptom ("buboes");
addSymptom ("shortness_of_breath");
```

Analysis 5. What should diagnosis return if no calls to addSymptom have been made?

Analysis 6. What aspects of the program are not tested by the calls in the example?

Analysis 7. Consider the sequence of calls

```
reset ( )
addSymptom (some symptom)
diagnosis ( )
```

Give an argument to addSymptom that, using the example data base above, results in the call to diagnosis returning a one-element array.

Analysis 8. What's wrong with the following data base?

symptom	*possible diseases*
fever	ague plague
chill	plague
shivering	ague
buboes	plague

disease	*necessary symptoms*
plague	buboes
ague	fever chill

Analysis 9. What does the answer to the previous question indicate about the relationship between the possible diseases for a given symptom and the necessary symptoms for each of those diseases?

Analysis 10. The problem statement says that the diagnosed diseases must all have been identified as possible diseases. Why not just have the diagnosis refer to *all* diseases?

Analysis 11. Can supplying more legal symptoms without any intervening calls to reset ever invalidate a diagnosis already made? For example, in the call sequence

```
diagnosis ( )
addSymptom (some symptom)
diagnosis ( )
```

must every disease in the array returned by the first call to diagnosis also be in the array returned by the second call? Briefly explain your answer.

Part 1: A first prototype

Class design

How might a programmer approach this problem?

For this problem, we take an object-oriented approach to design, that is, an approach organized around classes. Java supports object-oriented design quite well. Java also provides an extensive class library; it is likely that use of library classes will simplify our design considerably, at least in the initial stages.

A good way to start on an object-oriented design is to identify nouns in the problem specification. Each noun is a candidate for a class. We list a few below, each with an abstract representation.

noun	*representation*
symptom	a string
symptom list	an array (returned by `missingSymptoms`)
legal symptoms	a set of strings
disease	a string
diagnosis	an array of diseases (returned by `diagnosis`)
possible diseases	a set of diseases
necessary symptoms	a set of symptoms
data base	a set of patient symptoms, a set of legal symptoms, and two sets of *associations*. In the first set of associations, each association pairs a symptom with the set of diseases that exhibit the symptom. In the second set, each association pairs disease with the corresponding necessary symptoms.
symptom information	a `BufferedReader` (directly specified in the problem statement)
disease information	a `BufferedReader` (directly specified in the problem statement)

What kinds of data collections are involved in a solution?

Several of the quantities above are essentially *sets*, collections of items without duplicates. There is no implied sequence of elements of a set. Set operations include initialization, adding an element, removing an element, computing the union, difference, or intersection of two sets, and determining how many elements a set contains or whether a given item is one of the set elements.

We rewrite the steps of the various `MedicalDataBase` methods using set vocabulary, choosing variable names as appropriate.

Constructor

1. Initialize a set named legalSymptoms and an association set named symptomPossibleDiseases to empty.
2. For each line in the symptom information, let s be the first word on the line (the symptom). Each remaining word on the line names a possible disease for that symptom; let D be the set of those diseases. Add s to legalSymptoms, and add an association pairing s with D to symptomPossibleDiseases.
3. Initialize an association set named diseaseNecessarySymptoms to empty.
4. For each line in the disease information, let d be the first word on the line (the disease). Each remaining word on the line names a necessary symptom for that disease; let S be the set of those symptoms. Add an association pairing d with S to diseaseNecessarySymptoms.
5. Initialize a set named patientSymptoms to empty.

addSymptom

1. If the given symptom doesn't appear in legalSymptoms, throw an exception.
2. Add the given symptom to patientSymptoms.

reset

1. Set patientSymptoms to empty.

diagnosis

1. Initialize a set named possibleDiseases to empty.
2. Set possibleDiseases to the union, over all s in patientSymptoms, of the possible diseases for s.
3. Initialize a set named result to empty.
4. For each disease d in possibleDiseases, determine if all necessary symptoms for d are in patientSymptoms. (This is essentially an "is subset?" operation.) If so, add d to result.
5. Return result, converted to an array.

missingSymptoms

1. Initialize a set named possibleDiseases to empty.
2. Set possibleDiseases to the union, over all s in patientSymptoms, of the possible diseases for s.
3. If the given disease isn't in possibleDiseases, return null.
4. Let S be the necessary symptoms for the given disease. Determine difference (S, patientSymptoms), that is, all symptoms in S that aren't in patientSymptoms; then convert it to an array and return it.

What classes will represent the sets?

We assign class names to the various types of data as follows.

- Individual symptoms and diseases may each be represented in a Java String.
- The possibleDiseases set, the set of possible diseases associated with a given symptom, and the set of necessary symptoms for a given disease may each be represented by a class we'll call StringSet. (This class could also represent the results returned by diagnosis and missingSymptoms if these sets were not already specified to be arrays.)
- The symptomPossibleDiseases set and the diseaseNecessarySymptoms set may be represented by a class we'll call AssociationSet. Each element of this set will be an Association, a class with a String and a StringSet as instance variables.

It is common for a set to be represented by a class, and for its elements to be represented by a different class, as in the organization just described.

Figure 1 contains some framework code for MedicalDataBase. It will have two AssociationSet instance variables named symptomPossibleDiseases and diseaseNecessarySymptoms, plus a StringSet named patientSymptoms as just described. It will have (at least) five methods: a constructor, addSymptom and reset, and diagnosis and missingSymptoms.

Stop and help → *Supply as little as possible code for the "..." in Figure 1 so that the class compiles with no error messages.*

What methods will the set classes provide?

The methods for StringSet and AssociationSet may be inferred from the MedicalDataBase method descriptions. Consistent with convention, we use the phrase "this set" to mean the StringSet or AssociationSet referred to by this.

StringSet

- Initialize/reset this set to empty (constructor, reset).
- Given sets of diseases, return their union (diagnosis, missingSymptoms).
- Determine if a set is a subset of this set (diagnosis).
- Given two sets of symptoms, return their difference (missingSymptoms).
- Add an element to this set (addSymptom, diagnosis).
- Determine if a given string is an element of this set (missingSymptoms).
- Convert this set to an array (diagnosis, missingSymptoms).

```
import java.io.*;

public class MedicalDataBase {

    private AssociationSet symptomPossibleDiseases;
    private AssociationSet diseaseNecessarySymptoms;
    private StringSet patientSymptoms;

    public MedicalDataBase (BufferedReader symptomInfo,
                            BufferedReader diseaseInfo) {
        ...
    }

    // Add the given symptom to the patient symptom set.
    public void addSymptom (String symptom)
        throws UnrecognizedSymptomException {
        ...
    }

    // Set the patient symptom set to empty.
    public void reset ( ) {
        ...
    }

    // Return a diagnosis for the current set of patient symptoms.
    public String [ ] diagnosis ( ) {
        ...
    }

    // Return information about the appropriateness of the given disease
    // as a possible diagnosis for the current set of patient symptoms.
    public String [ ] missingSymptoms (String disease) {
        ...
    }
}
```

Figure 1
Framework for MedicalDataBase **class**

AssociationSet

- Initialize this set to empty.
- Add an association pairing a given String and StringSet to this set.
- Given a String, return the associated StringSet value.

Two class frameworks, shown in Figures 2 and 3, emerge from these operation lists. Some notes: The bracket notation for each collection class (e.g. <String>) shows that each element of the collection is a String (in StringSet) or an Association (in AssociationSet). Association is an example of a nested class, one defined inside the AssociationSet class; it's defined in this way because it's only used by AssociationSet, and furthermore is used only to pair the disease or symptom with the corresponding StringSet. These classes resemble classes in Java's Collection Framework, so we choose method names to match methods in this framework. In particular, AssociationSet is an example of a *map*, a type set up precisely to store key-

value associations. (A map is similar to tables used with assoc in Scheme. We'll revisit them later in this course.)

```
public class StringSet {
    private some_collection <String> strings;
    public StringSet ( ) {
        strings = new some_collection <String> ( );
    }
    // Add the given string to this set.
    public void add (String s) {
        ...
    }
    // Add all the strings in the set argument to this set.
    public StringSet addAll (StringSet set) {
        ...
    }
    // Return true if the argument is an element of this set; return false otherwise.
    public boolean contains (String set) {
        ...
    }
    // Return true if every string in the set argument is an element of this set.
    public boolean containsAll (StringSet set) {
        ...
    }
    // Return the set of all elements of this set that aren't in the argument set.
    public StringSet difference (StringSet set) {
        ...
    }
    // Make this set empty.
    public void reset ( ) {
        ...
    }
    // Return the result of converting strings to an array.
    public String [ ] toArray ( ) {
        ...
    }
}
```

Figure 2
Framework for StringSet class

Finally, we choose a type for the "some_collection" in the StringSet and AssociationSet frameworks. At this point in the design, a programmer should be worrying about correctness rather than efficiency, so we decide on a collection type that makes all the operations easy to implement. The java.util.ArrayList class should be appropriate; it already provides add, addAll, contains, and containsAll methods, so the rest shouldn't be difficult to code.

```
public class AssociationSet {

    private some_collection <Association> associations;

    public AssociationSet ( ) {
        associations = new some_collection <Association> ( );
    }

    // Add the association of s and set to this association set.
    public void put (String s, StringSet set) {
        ...
    }

    // Return the StringSet associated with the given String.
    public StringSet get (String s) {
        ...
    }

    private class Association {

        public String key;
        public StringSet values;

        public Association (String s, StringSet strings) {
            key = s;
            values = strings;
        }
    }
}
```

Figure 3
Framework for AssociationSet **class**

Application 12. Describe two other uses for an AssociationSet.

Analysis 13. Explain why strings can be accessed in the Association constructor, even though Association is a private class.

Analysis 14. Can AssociationSet methods access the AssociationSet instance variables key and values? Briefly explain.

Analysis 15. Explore the Java Collection classes to find other methods that they provide that might be useful for this problem.

Coding and testing for correctness

How should implementation of these classes proceed?

At this point, we may proceed in several directions. One would be to set up "dummy" StringSet and AssociationSet classes and use those to test the control flow of the MedicalDataBase methods. This is a *top-down* approach that starts with the top-level class and moves down to the details. Another would be to implement the program *bottom up*, coding and testing the StringSet and AssociationSet classes before worrying about the code that uses them. We choose the latter approach.

Moreover, we will apply the technique of *test-driven development*. This technique suggests that test code be written *before* program code, and that as little program code as possible be provided to satisfy the tests.

We start with StringSet. Its methods are the following:

- a constructor
- add and addAll
- contains and containsAll
- difference
- reset
- toArray

Our test suite will exercise all these methods.

Stop and predict →

Describe the test suite for the StringSet *class.*

A good way to start the test suite design is to construct small StringSets, say, with zero, one, and two strings, then to verify their construction. We will also need to make sure that duplicates are handled correctly—since we're working with *sets*, addition of a string to a set that already contains it should have no effect.

Stop and help →

Review the methods provided by the JUnit *testing facility. In particular, determine which of the methods will be most useful.*

How is the StringSet **constructor tested?**

The first test method is below. It calls the constructor to produce an empty set. Of the StringSet methods, only contains and containsAll provide access to an already-built set; we remedy this deficiency by adding a size method, and use it to verify that the empty set contains no elements. (Sometimes the public interface of a class shouldn't be changed; for instance, the problem specification forbids adding any public methods to the MedicalDataBase class. The StringSet class, however, has no such restrictions—we're designing it on our own. Moreover, the additional method doesn't provide any way to *change* the set, and thus it shouldn't contribute to any hard-to-find bugs.)

```
public void testEmpty ( ) {
    StringSet s = new StringSet ( );
    assertTrue (s.size ( ) == 0);
}
```

Stop and help →

Why is it not necessary to include a test that s != null *?*

The code for the constructor and size methods is easy:

```
private ArrayList<String> strings;

public StringSet ( ) {
    strings = new ArrayList<String> ( );
}

public int size ( ) {
    return strings.size ( );
}
```

The test passes. So far, so good.

How is the add method tested?

Moving on to a one-element set, we observe that the contains method can be used to help determine *what's in* the set. Here's one way of doing this.

```
public void test1element ( ) {
    StringSet s = new StringSet ( );
    s.add ("x");
    assertTrue (s.size ( ) == 1);
    assertTrue (s.contains ("x"));
    assertFalse (s.contains ("y"));
    // "y" was chosen arbitrarily.
}
```

This test requires code for the add and contains methods.

```
public void add (String s) {
    if (!strings.contains (s)) {
        // avoid duplicate elements!
        strings.add (s);
    }
}

public boolean contains (String s) {
    return strings.contains (s);
}
```

Again the tests pass.

The add method should be tested more, so we do that now. In addition, we'll test the reset method. Here are some tests.

```
public void testAddAndReset ( ) {
    StringSet set = new StringSet ( );
    set.add ("x");
    set.add ("y");
    assertTrue (set.size ( ) == 2);
    assertTrue (set.contains ("x"));
    assertTrue (set.contains ("y"));
    set.reset ( );
    assertTrue (set.size ( ) == 0); // any leftovers?
    assertFalse (set.contains ("x"));
    assertFalse (set.contains ("y"));
    set.add("x");
    assertTrue (set.size ( ) == 1);
    assertTrue (set.contains ("x"));
    assertFalse (set.contains ("y"));
    set.add("x");           // set should be unchanged
    assertTrue (set.size ( ) == 1);
    assertTrue (set.contains ("x"));
    assertFalse (set.contains ("y"));
    set.reset ( );
    assertTrue (set.size ( ) == 0);
    assertFalse (set.contains ("x"));
    assertFalse (set.contains ("y"));
}
```

The only code to be added is the reset method. Tests reveal no errors.

```
public void reset ( ) {
    strings = new ArrayList<String> ( );
}
```

How can testing of addAll, containsAll, and difference be simplified?

The methods addAll, containsAll, and difference are next to be tested. All of these involve sequential processing of the elements of a set, which suggests that enumeration methods will be useful.

The pattern for an enumeration—the process of successively returning the elements of the set—consists of three methods: one to initialize the enumeration, another to indicate whether or not the enumeration is complete, and a third to return the next element. We'll name these methods initEnumeration, hasMoreElements, and nextElement.*

Another requirement is a variable that keeps track of the progress of the iteration. By convention, it will indicate the next value to be returned. The nextElement method must then save the indicated value, advance the "keeping track" variable, then return the saved value.

Finally, calling hasMoreElements shouldn't change the state of the enumeration. Though the enumeration methods hasMoreElements and nextElement will typically be called in alternation (if more elements remain, retrieve and process the next one), two consecutive calls to hasMoreElements without an intervening call to nextElement should always return the same value. (Similarly, two calls to nextElement *without an intervening call to* hasMoreElements should return the next two elements in the set.)

How are the enumeration methods tested?

To test the enumeration methods, one may either make them private and test them indirectly through addAll, containsAll, and difference, or make them public and test them directly in a JUnit method. We choose the latter, both because they can't be used to change the StringSet—see the discussion about the size method above—and also because it is conventional in Java for a collection class to provide the enumeration capability. (The enumeration we've designed is somewhat simplistic. We will encounter how to implement a more general version soon in this course.)

As in earlier test methods, we start by enumerating an empty set, then a set with one element, then a set with two:

```
public void testEnumeration ( ) {
    // 0-element set
    StringSet set = new StringSet ( );
    set.initEnumeration ( );
    assertFalse (set.hasMoreElements ( ));
    assertFalse (set.hasMoreElements ( ));
    // 1-element set
    set.add ("x");
    set.initEnumeration ( );
    assertTrue (set.hasMoreElements ( ));
```

* Names are consistent with the methods of java.util.Enumeration, which we'll study soon.

```
        assertTrue (set.hasMoreElements ( ));
        String s;
        s = set.nextElement ( );
        assertTrue (s.equals ("x"));
        assertFalse (set.hasMoreElements ( ));
        assertFalse (set.hasMoreElements ( ));
        set.add ("x");              // shouldn't change set
        set.initEnumeration ( );
        assertTrue (set.hasMoreElements ( ));
        assertTrue (set.hasMoreElements ( ));
        s = set.nextElement ( );
        assertTrue (s.equals ("x"));
        assertFalse (set.hasMoreElements ( ));
        assertFalse (set.hasMoreElements ( ));
        // 2-element set
        set.add ("y");
        set.initEnumeration ( );
        assertTrue (set.hasMoreElements());
            ...
```

Stop and help→ *Why are the calls to* hasMoreElements *duplicated?*

Our initial impulse for the two-element set is to assume that the elements will be enumerated in the order they were added, that is, first "x", then "y". But sets are *unordered*; we shouldn't depend on the enumeration returning elements in a particular sequence. Here is code that provides for "x" and "y" to be enumerated in *either* order:

```
            ...
        String s1 = set.nextElement();
        assertTrue (set.hasMoreElements());
        assertTrue (set.hasMoreElements());
        String s2 = set.nextElement();
        assertFalse (set.hasMoreElements());
        assertFalse (set.hasMoreElements());
        assertTrue ((s1.equals("x")&& s2.equals("y"))
            || ((s1.equals("y") && s2.equals("x"))));
        set.initEnumeration ( );
        s1 = set.nextElement ( );
        s2 = set.nextElement ( );
        assertTrue ((s1.equals("x")&& s2.equals("y"))
            || ((s1.equals("y") && s2.equals("x"))));
    }
```

How are the enumeration methods coded?

We move on to the code for the enumeration methods. As noted above, this code will require a variable to keep track of the enumeration; we'll name this nextIndex. initEnumeration will set it to 0; hasMoreElements will check if its value has reached the number of elements in the set; nextElement will save the indexed value, update nextIndex, and return the saved value as described previously. Here is the code.

```
private int nextIndex;
// index in strings of the next string to be enumerated

public void initEnumeration ( ) {
    nextIndex = 0;
}
```

```
public boolean hasMoreElements ( ) {
    return nextIndex < strings.size();
}

// Return the next element in the enumeration.
// Precondition: hasMoreElements ( )
public String nextElement ( ) {
    String s = strings.get (nextIndex);
    nextIndex++;
    return s;
}
```

Tests pass. We're on a roll!

Stop and consider →

Should nextElement *call* hasMoreElements *to verify the precondition? If a call to* hasMoreElements *were to be included in* nextElement*, what should happen if* hasMoreElements *returns* false*?*

Stop and help →

Trace through the steps taken by initEnumeration *and* hasMoreElements *when* strings *is empty.*

How are addAll**,** containsAll**, and** difference **tested?**

Thorough testing of addAll, containsAll, and difference should start with small test arguments: empty sets, and sets with one element. Tests involving sets with common elements, equal sets, and unequal sets will follow. Here, for example, is test code for containsAll.

```
public void testContainsAll ( ) {
    // equal sets, 1 element each
    StringSet s1 = new StringSet ( );
    s1.add ("x");
    StringSet s2 = new StringSet ( );
    s2.add ("x");
    assertTrue (s1.containsAll (s2));
    assertTrue (s2.containsAll (s1));
    // 1-element vs. 2-element set
    s2.add ("z");
    assertFalse (s1.containsAll (s2));
    assertTrue (s2.containsAll (s1));
    // empty set vs. 1-element set
    StringSet s3 = new StringSet ( );
    assertFalse (s3.containsAll (s1));
    assertTrue (s1.containsAll (s3));
    // 1-element vs. 1-element set (not the same element)
    s3.add ("z");
    assertFalse (s3.containsAll (s1));
    assertFalse (s1.containsAll (s3));
}
```

How is containsAll **coded?**

We use the enumeration methods to write containsAll as follows, then test.

```
public boolean containsAll (StringSet set) {
    initEnumeration ( );
    while (hasMoreElements ( )) {
        if (!strings.contains (nextElement ( ))) {
            return false;
        }
    }
```

```
        return true;
    }
```

Stop and predict → *What's wrong with this code?*

Oops. A test failed, namely

```
// 1-element vs. 2-element set
s2.add ("z");
assertFalse (s1.containsAll (s2));
```

The contents of s1 is "x"; s2 contains "x" and "z"; s1 obviously doesn't contain the "z" in s2. What should be happening is that every element in the set argument (s2) is examined to make sure it is also in this set (s1).

Examination of containsAll reveals that's not what's happening at all. Instead, it looks at every element of *this* set rather than the argument set. (A clue to the problem: the set argument isn't used anywhere in containsAll.) The code below fixes the problem.

```
public boolean containsAll (StringSet set) {
    set.initEnumeration();
    while (set.hasMoreElements()) {
        if (!strings.contains(set.nextElement())) {
            return false;
        }
    }
    return true;
}
```

Stop and predict → *Describe a thorough set of tests for the* addAll *method.*

How is addAll **tested?**

Appearing below are tests for addAll. Some statements are missing, namely those indicated by "verify ..." comments.

```
public void testAddAll ( ) {
    // add to an empty set
    StringSet s1 = new StringSet ( );
    StringSet s2 = new StringSet ( );
    s2.add ("x");
    s2.add ("y");
    s1.addAll (s2);
    // verify s1 = {"x", "y"}

    // add an empty set
    StringSet empty = new StringSet ( );
    s1.addAll (empty);
    // verify s1 = {"x", "y"}

    // add with duplicates
    StringSet s3 = new StringSet ( );
    s3.add ("y");
    s3.add ("z");
    s1.addAll (s3);
    // verify s1 = {"x", "y", "z"}

    // add with no duplicates
    StringSet s4 = new StringSet ( );
    s4.add ("a");
```

```
        s4.add ("b");
        s1.addAll (s4);
        // verify s1 = {"a", "b", "x", "y", "z"}
    }
```

The comments suggest that a method to verify that a StringSet exactly contains a given set of strings would be a useful submethod to write. Furthermore, the notation of values within curly brackets (braces) that is used to represent a set matches the notation used to initialize an array. Can arrays can be used to simplify testing? Yes. To check whether all the strings in the array are in a StringSet named set, one merely loops through the array, calling set.contains on each element. (The approach is similar to that of containsAll.)

Here is code that combines these two observations.

```
public void testSetEquals (StringSet set, String [ ] arg) {
    assertTrue (set.size ( ) == arg.length);
    for (int k=0; k<arg.length; k++) {
        assertTrue (set.contains(arg[k]));
    }
}
```

A sample call sequence (the last verify in the testAddAll code above) would be the following.

```
String [ ] abxyz = {"a", "b", "x", "y", "z"};
testSetEquals (s1, abxyz);
```

We replace the "verify" comments by similar code segments, then move on to the code for addAll. It is similar to containsAll:

```
public void addAll (StringSet set) {
    set.initEnumeration ( );
    while (set.hasMoreElements ( )) {
        add (set.nextElement ( ));
    }
}
```

How is difference tested?

The tests succeed. We move on to the difference method. As with addAll, we test empty sets, equal sets, and overlapping sets:

```
public void testDifference ( ) {
    StringSet s1 = new StringSet ( );
    StringSet s2 = new StringSet ( );
    s2.add ("b");
    s2.add ("c");
    StringSet diff;
    String [ ] empty = {};
    String [ ] bc = { "b", "c" };
    testSetEquals (s1.difference (s2), empty);
    testSetEquals (s2.difference (s1), bc);
    testSetEquals (s1.difference (s1), empty);
    testSetEquals (s2.difference (s2), empty);
    String [ ] a = { "a" };
    String [ ] d = { "d" };
```

```
        s2.add ("d"); // now contains {"b", "c", "d"}
        StringSet s3 = new StringSet ( );
        s3.add ("a");
        s3.add ("b");
        s3.add ("c");
        testSetEquals (s2.difference (s3), d);
        testSetEquals (s3.difference (s2), a);
    }
```

Stop and help → *Supply comments in* testDifference *that describe the role of each test.*

How is difference **coded?**

The code for the difference method is somewhat similar to that of containsAll.

```
// Return those elements of this set that aren't
// contained in the set argument.
public StringSet difference (StringSet set) {
    StringSet rtn = new StringSet ( );
    initEnumeration ( );
    while (hasMoreElements ( )) {
        String s = nextElement ( );
        if (!set.contains (s)) {
            rtn.add (s);
        }
    }
    return rtn;
}
```

Stop and help → *Describe the similarities and differences between* containsAll *and* difference.

The tests all succeed.

How is toArray **tested and coded?**

The last operation to implement is toArray. We've already written the testSetEquals method to test whether an array and a StringSet contain the same elements, so we just reuse it for a JUnit test:

```
public void testToArray ( ) {
    StringSet s = new StringSet ( );
    testSetEquals (s, s.toArray ( ));
    s.add ("a");
    testSetEquals (s, s.toArray ( ));
    s.add ("b");
    testSetEquals (s, s.toArray ( ));
}
```

The code itself uses a standard array coding pattern:

```
    public String [ ] toArray ( ) {
        String [ ] rtn = new String [strings.size ( )];
        for (int k=0; k<strings.size ( ); k++) {
            rtn[k] = strings.get (k);
        }
        return rtn;
    }
```

Tests succeed. We're done with StringSet. Code for StringSet.java and StringSetTest.java appears in Appendix A. That

appendix also contains code for AssociationSet.java and AssociationSetTest.java, derived in pretty much the same way as the StringSet implementation and tests.

How should MedicalDataBase methods be tested?

We move to the MedicalDataBase class. The problem specification provides a relatively large set of examples of intended behavior of these methods. Before spending time inventing tests from scratch, it makes sense to review the examples to see what aspects of the program they already exercise. Some observations:

- The reset method isn't mentioned in the examples.
- The disease information contains diseases with no required symptoms, diseases with one required symptom, and a disease with more than one required symptom. This seems like a good variety.
- The symptom information contains symptoms with 1, 2, and 3 possible diseases. (There must be at least one possible disease for each symptom.) Again, this seems like a good variety.
- The situation of a patient with no symptoms isn't mentioned.
- There is an example of an unknown symptom argument to addSymptom, as well as several examples of legal symptom arguments.
- All three possibilities for missingSymptom return values are covered.
- Several cases for diagnosis return values are covered (empty, one disease, four diseases).

We conclude that we need add only a call to reset, followed by calls to diagnosis and missingSymptoms, to the calls already listed in the problem specification.

Code for the MedicalDataBase methods emerged directly from the pseudocode we designed at the start of this narrative. The only tricky part was deciding what to return as a diagnosis if there are no symptoms. (Tests revealed that our initial choice for this was incorrect.)

The code for MedicalDataBase.java and MedicalDataBaseTest.java appears in Appendix B.

Application

16. Implement the equals method for the StringSet class. Conforming to Java conventions, the header for the equals method is

    ```
    public boolean equals (Object obj)
    ```

    Given a StringSet object s as argument, equals returns true if s and this set contain exactly the same elements.

Analysis 17. To what extent would the availability of the StringSet equals method simplify testing?

Analysis 18. Comment on the advisability of adding a public method sort to the StringSet class to simplify testing. The sort method would rearrange StringSet elements to be in a predictable order.

Analysis 19. Consider a representation for string sets that internally allowed sets to contain duplicate elements. (This representation would make the add method more efficient because its call to contains is no longer necessary.) What other StringSet methods would have to be changed to accommodate the new representation?

Modification 20. Modify the StringSet class to implement the representation for sets described in exercise 19.

Analysis 21. Suppose the call to contains in the StringSet add method were removed. What would be the first test in StringSetTest to fail as a result?

Debugging 22. Consider the following implementation of the enumeration methods:

```
private int enumIndex;
// index of the most recently returned string

public void initEnumeration ( ) {
    enumIndex = -1;
}

public boolean hasMoreElements ( ) {
    return enumIndex < strings.size ( );
}

public String nextElement ( ) {
    enumIndex++;
    return strings.get (enumIndex);
}
```

The code contains a bug. Find and fix it.

Analysis 23. MedicalDataBase tests didn't check if the diagnosis contains duplicate values. Why not?

Modification 24. The ArrayList iterator method returns an Iterator object, which provides methods hasNext and next. Rewrite the enumeration methods to use ArrayList.iterator.

Application 25. Rewrite the enumeration operations to be part of a nested class named ElementEnumeration.

Reflection 26. What's the naming convention for the string arrays used in testAddAll and testDifference? What names would you have chosen for these arguments?

Performance testing

How should `diagnosis` and `missingSymptoms` be tested on a large data base?

The problem specification requires that "the `diagnosis` and `missingSymptoms` methods should respond quickly, even with a large data base." Verifying that our solution satisfies this requirement involves testing it on a large data base, preferably a data base that produces the longest running time for a query. Obviously we don't want to type the data ourselves; instead, we'll write a program to generate the input.

We observe that the actual names of diseases and symptoms aren't relevant to the program's performance; for example, it doesn't matter to the program whether a symptom is named `"buboes"` or `"dfahefxz"`. To simplify the generation of data files, we'll name the symptoms `"s0"`, `"s1"`, `"s2"`, and so on, and the diseases `"d0"`, `"d1"`, `"d2"`, etc.

We also decide that all symptoms should have the same number of possible diseases, and all diseases should have the same number of necessary symptoms. The uniformity of this scheme should be easier to implement in the data-generating program. Also, we suspect that a short disease or symptom list will never involve more processing than a long one, so extending all short lists to be the same length shouldn't result in a data base on which performance is accidentally good.

Finally, we decide to generate symptom information as follows:

```
s0  d0  d1  d2  ...
s1  d1  d2  d3  ...
s2  d2  d3  d4  ...
   .
   .
   .
```

We suspect that this uniformity will not affect the running time, and it should also be easy to implement in the data-generating program.

What should the disease information look like?

The symptom information format doesn't at first glance suggest any obvious relation between diseases and candidates for necessary symptoms. For better insight, we examine an example, using the symptom information we generated.

```
s0 d0 d1 d2
s1 d1 d2 d3
s2 d2 d3 d4
s3 d3 d4 d0
s4 d4 d0 d1
s5 d0 d1 d2
s6 d1 d2 d3
s7 d2 d3 d4
s8 d3 d4 d0
s9 d4 d0 d1
```

Each disease is suggested by six symptoms, as shown in the table below.

disease	*symptoms that suggest it*
d0	s0 s3 s4 s5 s8 s9
d1	s0 s1 s4 s5 s6 s9
d2	s0 s1 s2 s5 s6 s7
d3	s1 s2 s3 s6 s7 s8
d4	s2 s3 s4 s7 s8 s9

The necessary symptoms for each d will be chosen from the corresponding list above. (See exercise 9.) How should that be done? There's still no obvious formula that, given a disease, lets us choose from the symptoms that suggest that disease.

However, a solution appears: instead of regenerating the possibilities for each disease's necessary symptoms, we will keep track of them in an array. Each time a disease d[j] is printed in the line for symptom s[k], we will register the pair (j,k) in the array. Then we go through the array, generating a line for each disease that contains a subset of the associated symptoms.

How should the data generating program be organized to be as flexible as possible?

The data can now be organized according to five quantities:

- the number of recognized symptoms (we'll name this symptomCount);
- the number of recognized diseases (diseaseCount);
- the number of possible diseases per symptom (diseasesPerSymptom);
- the number of necessary symptoms per disease (symptomsPerDisease); and
- the number of patient symptoms (patientSymptomCount).

Organizing the program around these five quantities will allow for easy changes to produce a variety of test data.

Stop and predict → *Describe what one would do to "organize the program around these five quantities".*

Stop and consider → *What values for these quantities would be most representative of a "real-world" medical data base?*

Appendix C contains the program to generate sample data files. The five quantities around which the program is organized appear as static int constants. The first part, which produces the symptom information, produces possible diseases in the uniform format described above and tallies each symptom-disease pair in the array named table. The table array is two-dimensional; it's indexed by a symptom index and a disease

index. The second part produces disease information using table as just described. The third part merely produces a list of the first patientSymptomCount symptoms. All three parts use standard Java idioms for file processing and processing all pairs of integers j and k in a given range.

How are the data files used?

To make use of the generated data files, we add a main method to MedicalDataBase:

```
public static void main (String [ ] args) throws Exception {
    // arg 0 is name of symptom info file
    // arg 1 is name of disease info file
    // arg 2 is name of patient symptom file
    BufferedReader symptomInfo = new BufferedReader (new FileReader (args[0]));
    BufferedReader diseaseInfo = new BufferedReader (new FileReader (args[1]));
    MedicalDataBase db = new MedicalDataBase (symptomInfo, diseaseInfo);
    BufferedReader patientSymptoms = new BufferedReader (new FileReader (args[2]));
    String symptom = patientSymptoms.readLine ( );
    while (symptom != null && symptom.length ( ) > 0) {
        db.addSymptom (symptom);
        symptom = patientSymptoms.readLine ( );
    }
    String [ ] diag = db.diagnosis ( );
    for (int k=0; k<diag.size(); k++) {
        System.out.println (diag[k]);
    }
}
```

It is supplied names of the symptom information, disease information, and patient symptom files as command-line argument. The method first constructs BufferedReaders for the first two and sends them to the MedicalDataBase constructor. Then it opens the patient symptoms file. Assuming that file contains one symptom per line, it repeatedly reads a line and sends it to addSymptom. Finally, it produces a diagnosis.

What are the results of testing?

We begin performance testing* with the following values:

```
symptomCount = 10000
diseaseCount = 5000
diseasesPerSymptom = 200
symptomsPerDisease = 100
patientSymptomCount = 100
```

The program crashes with an "out of memory" error after a bit more than a minute. (It didn't even print a message saying where in the program the error occurred.) It did the same thing for

```
symptomCount = 5000
diseaseCount = 2500
```

---

* Tests were conducted on a Macintosh 867MHz PowerPC G4 running Eclipse.

On the other hand, the values

```
symptomCount = 2500
diseaseCount = 1200
```

produced a successful diagnosis in around 45 seconds.

How sensitive is the program to changes in the other quantities? Trying

```
symptomCount = 2500
diseaseCount = 1200
diseasesPerSymptom = 600
symptomsPerDisease = 300
patientSymptomCount = 300
```

results in an "out of memory" crash. Substituting

```
diseasesPerSymptom = 400
symptomsPerDisease = 200
patientSymptomCount = 200
```

produces a successful diagnosis in around 90 seconds.

Where are the bottlenecks?

There are two problems to address. One is the time to find a diagnosis, when the program is able to do so. The second, and we think the more important, is the out-of-memory problem. Real-world medical data bases ought to be able to handle more than 5000 symptoms.

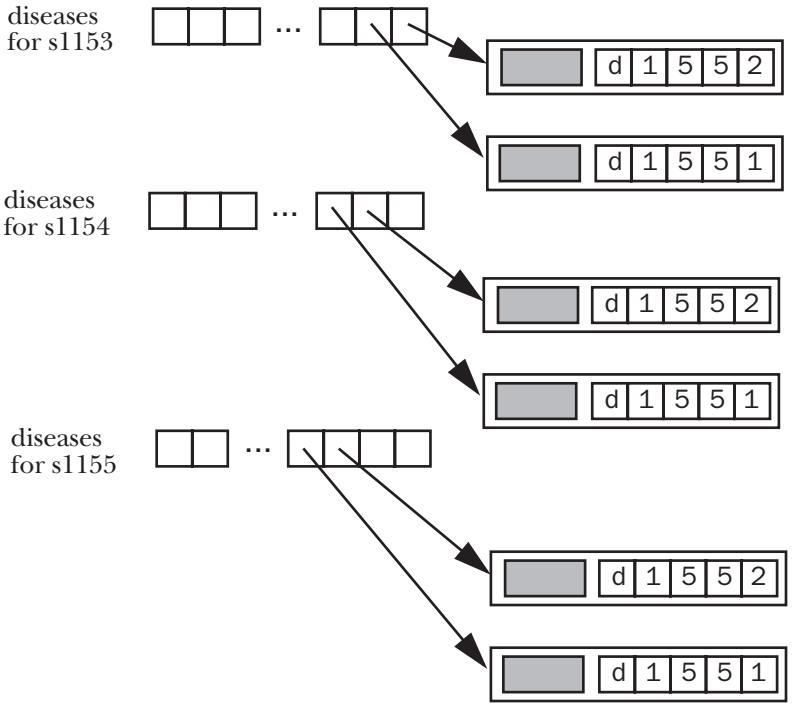
Further exploration of the crashes reveals that they occur in the MedicalDataBase constructor; there apparently is not enough memory even to store information about 5000 symptoms and 2500 diseases, much less to do any computation with them. How much memory is required? Here is a "back-of-the-envelope" calculation:

- Each line of symptom information contains one symptom name and diseasesPerSymptom disease names. When symptomCount is 5000 and diseasesPerSymptom is 200, that's a million strings.
- Similarly, diseaseCount = 2500 and symptomsPerDisease = 100 yields 250,000 strings.
- Most of the symptom and disease names in the test data base are four or five characters long. We don't know how much bookkeeping information is needed per String at run time. However, awareness that a String is probably implemented as an array of char suggests that the size of the array is probably a power of 2, and another integer is necessary to store the String length. The average symptom or disease name in the test data base thus requires at least storage for an integer plus eight characters.

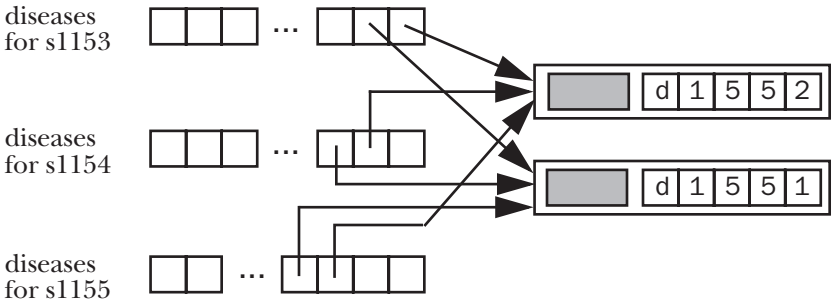
Together, the symptom information and the disease information use at least 15 million bytes (units of memory) for String objects alone. How can this total be reduced?

How might memory use be optimized?

It helps to draw a diagram of some of the string storage. For example, consider the diseases for symptoms `"s1553"`, `"s1554"`, and `"s1555"` diagrammed below. (The shaded area represents storage over and above what's needed to store the characters of the string.)



The diagram indicates that each disease set stores a *copy* of the characters in each relevant disease. However, this redundancy is unnecessary. All the copies of (say) the string `"d1552"` can be replaced by references to a single copy of `"d1552"`, as shown below.



Several questions remain. Where should the changes to implement this optimization be made? What structures will allow efficient maintenance of the string structure? To be continued in part 2 ...

Exercises

Analysis 27. Approximately how much memory will be saved by making the modification described in the narrative?

Reflection 28. The narrative describes a "back-of-the-envelope" estimate for the amount of memory used by strings. How close to the actual amount of memory should this estimate to be useful?

Modification 29. Rewrite the FileGen main method to accept the storage parameters as command-line arguments.

Analysis, Modification 30. The file generator is less efficient than necessary. Identify the source of the inefficiency, and fix it.

Analysis 31. What values for symptomCount, diseaseCount, diseasesPerSymptom, symptomsPerDisease, and patientSymptomCount would you expect in a real-world medical data base?

Outline of design and development questions

Class design

How might a programmer approach this problem?

What kinds of data collections are involved in a solution?

- What classes will represent the sets?
- What methods will the set classes provide?

Coding and testing for correctness

How should implementation of these classes proceed?

- How is the StringSet constructor tested?
- How is the add method tested?
- How can testing of addAll, containsAll, and difference be simplified?
  - How are the enumeration methods tested?
  - How are the enumeration methods coded?
- How are addAll, containsAll, and difference tested?
  - How is containsAll coded?
  - How is addAll tested?
  - How is difference tested?
  - How is difference coded?

How should MedicalDataBase methods be tested?

Performance testing

How should diagnosis and missingSymptoms be tested on a large data base?

What should the disease information look like?

How should the data generating program be organized to be as flexible as possible?

How are the data files used?

What are the results of testing?

Where are the bottlenecks?

How might memory use be optimized?

Programmers' Summary

This case study describes the design and development of a program that provides medical diagnosis advice. High-level design proceeds as in many object-oriented designs, first deriving *objects* in the program from *nouns* in the problem specification, then identifying the objects as instances of *classes* and determining their behavior. Recognition of two of the classes as common data types leads to decomposition of the program into classes representing *sets* of strings, sets of *associations*, and the medical data base itself, along with detailed pseudocode for the data base methods.

The intended design plan involves a series of *prototypes*, intermediate versions that lack some element of functionality. For the first prototype, correctness and ease of coding are most important. Once the first version is complete, the programmer cycles through several steps until an acceptable version results:

performance testing;
identification of bottlenecks;
removing the bottlenecks, typically by replacing a data structure with an alternative implementation.

The decomposition into set classes and client class should simplify the incremental improvement of the program, since it is likely that set operations will be the most significant contributor to program performance.

Next comes the development—coding and testing—of the first prototype. Development proceeds bottom up, first with the StringSet class, with then the AssociationSet class, and finally with the MedicalDataBase class that uses the set classes.

The technique of *test-driven development* is used throughout. *Unit tests*—tests of small areas of program functionality—are devised, the code to satisfy them is debugged, and the process is repeated. Several principles govern the organization of the groups of unit tests:

- when processing can be separated into a small number of cases, test them all;
- when processing collections of data, try collections of size 0, 1, 2, and perhaps 3;
- when processing sets in particular, check disjoint sets, sets that overlap, and identical sets.

Two other aspects of testing of the StringSet class are noteworthy. One is that sets are *unordered*; for example, the sets {"a", "b", "c"} and {"c", "b", "a"} are equivalent, and must be recognized as equal. The other is that JUnit tests are necessarily "black-box" tests, since the JUnit test case only has access to public StringSet information. The StringSet public interface was extended beyond those operations needed for the medical data base application, to include size and enumeration operations and thereby allow more thorough testing. These extensions are conventional operations for sets, and they don't introduce the extra complexity that would result from operations that would *change* the set.

Interesting features of the program code include the following.

- A nested Association class is used within the AssociationSet class, providing convenient access of the container class to its elements without exposing unnecessary information to client classes.
- The StringSet enumeration operations maintain an *invariant* property involving the index in the set of the next element in the enumeration to be returned.
- Arrays are used in the StringSetTest class to simplify the construction of correct values against which to compare the result of various set operations.

Development concludes with performance tests, which require a data base of realistic size. An auxiliary program is written to generate the data base. Underlying the design of this program are two simplifying assumptions: names of symptoms and diseases need not be realistic, but instead can be words such as "d231" and "s4507"; symptom-disease correspondences can be uniform, with a fixed number of potential diseases per symptom and necessary symptoms per disease. The program is also organized so as to allow easy generation of data bases of varying sizes.

Somewhat surprisingly, the use of simple data structures was not a performance bottleneck. Instead, the program's memory use was the key limitation. The narrative concludes with the explanation of the use of memory in the program, and a possible remedy for redundant storage of strings.

Exercises

Analysis 32. Comment on the advisability of having an additional StringSet constructor that initializes the string set from its argument, a line of symptoms or diseases.

Modification 33. Make the modification described in exercise 32.

Modification 34. Rewrite the StringSet and AssociationSet classes to *inherit* from classes in the java.util library.

Reflection 35. Compare JUnit use with testing in main.

Application 36. Describe an application where testing top down would be an appropriate approach.

Analysis 37. JUnit testing can only access public methods and variables of the class being tested, and therefore may be unable to catch errors such as a symptom name that matches a disease name, or unequal number of elements in symptomPossibleDiseases and legalSymptoms. List as many inconsistencies involving the MedicalDataBase private instance variables.

Application 38. Write a public audit method for the MedicalDataBase class that checks for the inconsistencies you listed in exercise 37. The audit method takes no arguments. It returns true if the private instance variables have reasonable and internally consistent values, and false if it finds some inconsistency.

Application 39. Design a test suite for a StringSet intersection method. A call to intersection with argument s should return a set whose elements are all those that are both in s and in this set.

Application 40. Provide the code for the intersection method.

Reflection 41. Describe the similarities and differences between intersection and difference.

Appendix A
StringSet.java, StringSetTest.java, AssociationSet.java, AssociationSetTest.java

StringSet.java

```
import java.util.*;

public class StringSet {

    private ArrayList<String> strings;

    // Initialize this set to empty.
    public StringSet ( ) {
        strings = new ArrayList<String> ( );
    }

    // Add the given string to this set if it's not already there.
    public void add (String s) {
        if (!strings.contains (s)) {
            strings.add (s);
        }
    }

    // Add all the strings in the given set to this set, i.e. produce the union of the two sets.
    public void addAll (StringSet set) {
        set.initEnumeration ( );
        while (set.hasMoreElements ( )) {
            add (set.nextElement ( ));
        }
    }

    // Reset this set to empty.
    public void reset ( ) {
        strings = new ArrayList<String> ( );
    }

    // Return true if this set contains s; return false otherwise.
    public boolean contains (String s) {
        return strings.contains (s);
    }

    // Return true if this set contains all the strings in the given set,
    // i.e. this set is a superset of the given set; return false otherwise.
    public boolean containsAll (StringSet set) {
        set.initEnumeration();
        while (set.hasMoreElements()) {
            if (!strings.contains(set.nextElement())) {
                return false;
            }
        }
        return true;
    }

    // Return those elements of this set that aren't contained in the given set.
    public StringSet difference (StringSet set) {
```

```
        StringSet rtn = new StringSet ( );
        initEnumeration ( );
        while (hasMoreElements ( )) {
            String s = nextElement ( );
            if (!set.contains (s)) {
                rtn.add (s);
            }
        }
        return rtn;
    }

    // Return the number of elements in this set.
    public int size ( ) {
        return strings.size ( );
    }

    private int nextIndex;                              // index of the next string to be enumerated

    public void initEnumeration ( ) {
        nextIndex = 0;
    }

    // Return true if there are more elements to be enumerated;
    // return false if all elements of this set have been enumerated.
    public boolean hasMoreElements ( ) {
        return nextIndex < strings.size();
    }

    // Return the next element in the enumeration.
    // Precondition: hasMoreElements ( )
    public String nextElement ( ) {
        String s = strings.get (nextIndex);
        nextIndex++;
        return s;
    }

    // Return the result of converting this set to an array.
    public String [ ] toArray ( ) {
        String [ ] rtn = new String [strings.size ( )];
        for (int k=0; k<strings.size ( ); k++) {
            rtn[k] = strings.get (k);
        }
        return rtn;
    }
}
```

StringSetTest.java

```
import junit.framework.TestCase;

public class StringSetTest extends TestCase {

    public void testEmpty ( ) {
        StringSet set = new StringSet ( );
        assertTrue (set.size ( ) == 0);
        assertFalse (set.contains ("foo"));
    }
```

```
public void test1element ( ) {
    StringSet set = new StringSet ( );
    set.add ("x");
    assertTrue (set.size ( ) == 1);
    assertTrue (set.contains ("x"));
    assertFalse (set.contains ("y"));
}

public void testAddAndReset ( ) {
    StringSet set = new StringSet ( );
    set.add ("x");
    set.add ("y");
    assertTrue (set.size ( ) == 2);
    assertTrue (set.contains ("x"));
    assertTrue (set.contains ("y"));
    set.reset ( );
    assertTrue (set.size ( ) == 0);
    assertFalse (set.contains ("x"));                          // any leftovers?
    assertFalse (set.contains ("y"));
    set.add("x");
    assertTrue (set.size ( ) == 1);
    assertTrue (set.contains ("x"));
    assertFalse (set.contains ("y"));
    set.add("x");                                  // set should remain unchanged
    assertTrue (set.size ( ) == 1);
    assertTrue (set.contains ("x"));
    assertFalse (set.contains ("y"));
    set.reset ( );
    assertTrue (set.size ( ) == 0);
    assertFalse (set.contains ("x"));
    assertFalse (set.contains ("y"));
}

public void testEnumeration ( ) {
    // 0-element set
    StringSet set = new StringSet ( );
    set.initEnumeration ( );
    assertFalse (set.hasMoreElements ( ));
    assertFalse (set.hasMoreElements ( ));
    // 1-element set
    set.add ("x");
    set.initEnumeration ( );
    assertTrue (set.hasMoreElements ( ));
    assertTrue (set.hasMoreElements ( ));
    String s;
    s = set.nextElement ( );
    assertTrue (s.equals ("x"));
    assertFalse (set.hasMoreElements ( ));
    assertFalse (set.hasMoreElements ( ));
    set.add ("x");                                 // this shouldn't change the set
    set.initEnumeration ( );
    assertTrue (set.hasMoreElements ( ));
    assertTrue (set.hasMoreElements ( ));
    s = set.nextElement ( );
    assertTrue (s.equals ("x"));
    assertFalse (set.hasMoreElements ( ));
    assertFalse (set.hasMoreElements ( ));
    // 2-element set
    set.add ("y");
```

```
    set.initEnumeration ( );
    assertTrue (set.hasMoreElements());
    assertTrue (set.hasMoreElements());
    String s1 = set.nextElement();
    assertTrue (set.hasMoreElements());
    assertTrue (set.hasMoreElements());
    String s2 = set.nextElement();
    assertFalse (set.hasMoreElements());
    assertFalse (set.hasMoreElements());
    assertTrue ((s1.equals("x") && s2.equals("y"))
           || ((s1.equals("y") && s2.equals("x"))));
    set.initEnumeration ( );
    s1 = set.nextElement ( );
    s2 = set.nextElement ( );
    assertTrue ((s1.equals("x")&& s2.equals("y"))
        || ((s1.equals("y") && s2.equals("x"))));}

public void testContainsAll ( ) {
    // equal sets, 1 element each
    StringSet s1 = new StringSet ( );
    s1.add ("x");
    StringSet s2 = new StringSet ( );
    s2.add ("x");
    assertTrue (s1.containsAll (s2));
    assertTrue (s2.containsAll (s1));
    // 1-element vs. 2-element set
    s2.add ("z");
    assertFalse (s1.containsAll (s2));
    assertTrue (s2.containsAll (s1));
    // empty set vs. 1-element set
    StringSet s3 = new StringSet ( );
    assertFalse (s3.containsAll (s1));
    assertTrue (s1.containsAll (s3));
    // 1-element vs. 1-element set (not the same element)
    s3.add ("z");
    assertFalse (s3.containsAll (s1));
    assertFalse (s1.containsAll (s3));
}

public void testAddAll ( ) {
    // add to an empty set
    StringSet s1 = new StringSet ( );
    StringSet s2 = new StringSet ( );
    s2.add ("x");
    s2.add ("y");
    s1.addAll (s2);
    String [ ] xy = { "x", "y" };
    testSetEquals (s1, xy);

    // add an empty set
    StringSet empty = new StringSet ( );
    s1.addAll (empty);
    testSetEquals (s1, xy);

    // add with duplicates
    StringSet s3 = new StringSet ( );
    s3.add ("y");
    s3.add ("z");
    s1.addAll (s3);
```

```
        String [ ] xyz = { "x", "y", "z" };
        testSetEquals (s1, xyz);

        // add with no duplicates
        StringSet s4 = new StringSet ( );
        s4.add ("a");
        s4.add ("b");
        s1.addAll (s4);
        String [ ] abxyz = { "a", "b", "x", "y", "z" };
        testSetEquals (s1, abxyz);
    }

    public void testSetEquals (StringSet set, String [ ] arg) {
        assertTrue (set.size ( ) == arg.length);
        for (int k = 0; k < arg.length; k++) {
            assertTrue (set.contains (arg[k]));
        }
    }

    public void testDifference ( ) {
        StringSet s1 = new StringSet ( );
        StringSet s2 = new StringSet ( );
        s2.add ("b");
        s2.add ("c");
        StringSet diff;
        String [ ] empty = {};
        String [ ] bc = { "b", "c" };
        testSetEquals (s1.difference (s2), empty);
        testSetEquals (s2.difference (s1), bc);
        testSetEquals (s1.difference (s1), empty);
        testSetEquals (s2.difference (s2), empty);
        String [ ] a = { "a" };
        String [ ] d = { "d" };
        s2.add ("d");                                    // now contains {"b", "c", "d"}
        StringSet s3 = new StringSet ( );
        s3.add ("a");
        s3.add ("b");
        s3.add ("c");
        testSetEquals (s2.difference (s3), d);
        testSetEquals (s3.difference (s2), a);
    }

    public void testToArray ( ) {
        StringSet s = new StringSet ( );
        testSetEquals (s, s.toArray ( ));
        s.add ("a");
        testSetEquals (s, s.toArray ( ));
        s.add ("b");
        testSetEquals (s, s.toArray ( ));
    }
}
```

AssociationSet.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class AssociationSet {

    private ArrayList<Association> myAssociations;

    private class Association {
        public String myKey;
        public StringSet myValues;

        public Association (String s, StringSet strings) {
            myKey = s;
            myValues = strings;
        }
    }

    public AssociationSet ( ) {
        myAssociations = new ArrayList<Association> ( );
    }

    // Add an association to this set.
    // Precondition: there is not already an association in this set with the given key.
    public void put (String key, StringSet values) {
        myAssociations.add (new Association (key, values));
    }

    // Return the value associated with the given key. If no such association exists, return null.
    public StringSet get (String key) {
        for (Association a: myAssociations) {
            if (a.myKey.equals (key)) {
                return a.myValues;
            }
        }
        return null;
    }
}
```

AssociationSetTest.java

```
import junit.framework.TestCase;

public class AssociationSetTest extends TestCase {

    public void testGetPut ( ) {
        AssociationSet set = new AssociationSet ( );
        assertTrue (set.get("a") == null);
        StringSet value1 = new StringSet ( );
        value1.add("hello");
        set.put("a", value1);
        assertTrue (set.get("a") == value1);
        assertTrue (set.get("b") == null);
        StringSet value2 = new StringSet ( );
        value2.add("world");
        set.put("b", value2);
        assertTrue (set.get("b") == value2);
    }
}
```

Appendix B
MedicalDataBase.java, MedicalDataBaseTest.java

MedicalDataBase.java

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

public class MedicalDataBase {

    private AssociationSet symptomPossibleDiseases;
    private AssociationSet diseaseNecessarySymptoms;
    private StringSet legalSymptoms;
    private StringSet patientSymptoms;

    // Initialize the data base from the given symptom and disease information.
    public MedicalDataBase (BufferedReader symptomInfo,
      BufferedReader diseaseInfo) throws IOException {
        symptomPossibleDiseases = new AssociationSet ( );
        diseaseNecessarySymptoms = new AssociationSet ( );
        legalSymptoms = new StringSet ( );
        patientSymptoms = new StringSet ( );

        String line;
        Scanner sc;

        // Each symptom information line starts with a symptom and continues
        // with one or more suggested diseases separated by white space.
        symptomPossibleDiseases = new AssociationSet ( );
        line = symptomInfo.readLine ( );
        while (line != null) {
            sc = new Scanner (line);
            String symptom = sc.next ( );
            StringSet diseases = new StringSet ( );
            while (sc.hasNext ( )) {
                diseases.add (sc.next ( ));
            }
            legalSymptoms.add (symptom);
            symptomPossibleDiseases.put (symptom, diseases);
            line = symptomInfo.readLine ( );
        }

        // Each disease information line starts with a disease and continues
        // with zero or more necessary symptoms separated by white space.
        diseaseNecessarySymptoms = new AssociationSet ( );
        line = diseaseInfo.readLine ( );
        while (line != null) {
            sc = new Scanner (line);
            String disease = sc.next ( );
            StringSet symptoms = new StringSet ( );
            while (sc.hasNext ( )) {
                symptoms.add (sc.next ( ));
            }
            diseaseNecessarySymptoms.put (disease, symptoms);
            line = diseaseInfo.readLine ( );
        }
```

```
    patientSymptoms = new StringSet ( );
}

// Add a symptom to those already collected for the patient.
public void addSymptom (String symptom)
  throws UnrecognizedSymptomException {
    if (!legalSymptoms.contains (symptom)) {
        throw new UnrecognizedSymptomException (symptom);
    }
    patientSymptoms.add (symptom);
}

// Reset the set of patient symptoms.
public void reset ( ) {
    patientSymptoms = new StringSet ( );
}

// Produce a diagnosis, that is, a list of diseases consistent with the given patient symptoms.
public String [ ] diagnosis ( ) {
    StringSet possibleDiseases = new StringSet ( );
    patientSymptoms.initEnumeration ( );
    while (patientSymptoms.hasMoreElements ( )) {
        possibleDiseases.addAll (symptomPossibleDiseases.get
            (patientSymptoms.nextElement ( )));
    }
    StringSet result = new StringSet ( );
    possibleDiseases.initEnumeration ( );
    while (possibleDiseases.hasMoreElements ( )) {
        String candidate = possibleDiseases.nextElement ( );
        if (patientSymptoms.containsAll
          (diseaseNecessarySymptoms.get (candidate))) {
            result.add (candidate);
        }
    }
    return result.toArray ( );
}

// If the given disease is not among those suggested by the current symptoms, return null.
// Otherwise, return a list of the symptoms required by the given disease
// but not among those displayed by the patient.
public String [ ] missingSymptoms (String disease) {
    StringSet possibleDiseases = new StringSet ( );
    patientSymptoms.initEnumeration ( );
    while (patientSymptoms.hasMoreElements ( )) {
        possibleDiseases.addAll (symptomPossibleDiseases.get
            (patientSymptoms.nextElement ( )));
    }
    if (!possibleDiseases.contains (disease)) {
        return null;
    }
    StringSet necessarySymptoms = diseaseNecessarySymptoms.get (disease);
    StringSet result = necessarySymptoms.difference (patientSymptoms);
    return result.toArray ( );
}

public static void main (String [ ] args) throws Exception {
    // arg 0 is name of symptom info file
    // arg 1 is name of disease info file
```

```
        // arg 2 is name of patient symptom file
        BufferedReader symptomInfo = new BufferedReader (new FileReader (args[0]));
        BufferedReader diseaseInfo = new BufferedReader (new FileReader (args[1]));
        MedicalDataBase db = new MedicalDataBase (symptomInfo, diseaseInfo);
        BufferedReader patientSymptoms = new BufferedReader (new FileReader (args[2]));
        String symptom = patientSymptoms.readLine ( );
        while (symptom != null && symptom.length ( ) > 0) {
            db.addSymptom (symptom);
            symptom = patientSymptoms.readLine();
        }
        String [ ] diag = db.diagnosis ( );
        for (int k=0; k<diag.length; k++) {
            System.out.println (diag[k]);
        }
    }
}
```

MedicalDataBaseTest.java

```
import junit.framework.TestCase;
import java.util.*;
import java.io.*;

public class MedicalDataBaseTest extends TestCase {

    private MedicalDataBase db;

    protected void setUp ( ) throws IOException {
        FileWriter symptomInfoFile = new FileWriter ("symptomInfo");
        symptomInfoFile.write ("fever ague plague\n");
        symptomInfoFile.write ("chill ague plague\n");
        symptomInfoFile.write ("shivering ague\n");
        symptomInfoFile.write ("buboes plague\n");
        symptomInfoFile.write ("shortness_of_breath all-nighter"
          + " caffeine_overdose exercise\n");
        symptomInfoFile.write ("sleepiness all-nighter" + " boring_lecturer\n");
        symptomInfoFile.close ( );

        FileWriter diseaseInfoFile = new FileWriter ("diseaseInfo");
        diseaseInfoFile.write ("plague buboes\n");
        diseaseInfoFile.write ("ague fever chill\n");
        diseaseInfoFile.write ("all-nighter\n");
        diseaseInfoFile.write ("caffeine_overdose\n");
        diseaseInfoFile.write ("exercise shortness_of_breath\n");
        diseaseInfoFile.write ("boring_lecturer\n");
        diseaseInfoFile.close ( );
    }

    public void testExample ( ) throws IOException {
        db = new MedicalDataBase
          (new BufferedReader (new FileReader ("symptomInfo")),
                new BufferedReader (new FileReader ("diseaseInfo")));
        try {
            db.addSymptom ("double_vision");
            fail ("failed to throw exception for unknown symptom");
        } catch (UnrecognizedSymptomException e){

        }
```

```
// possible results of diagnosis and missingSymptoms
String [ ] diagList, missingList;
String [ ] empty = { };
String [ ] buboes = {"buboes"};
String [ ] chill = {"chill"};
String [ ] ague = {"ague"};
String [ ] fourDiseases = {"ague", "all-nighter",
  "caffeine_overdose", "exercise"};

try {
    db.addSymptom ("fever");
} catch (UnrecognizedSymptomException e) {
    fail ("unrecognized symptom incorrectly thrown for \"fever\"");
}
diagList = db.diagnosis ( );
testArraySetEquals(diagList, empty);
missingList = db.missingSymptoms ("flu");
assertTrue (missingList == null);
missingList = db.missingSymptoms ("all-nighter");
assertTrue (missingList == null);
missingList = db.missingSymptoms ("plague");
testArraySetEquals(missingList, buboes);
missingList = db.missingSymptoms ("ague");
testArraySetEquals(missingList, chill);

try {
    db.addSymptom ("chill");
} catch (UnrecognizedSymptomException e1) {
    fail ("unrecognized symptom incorrectly thrown for \"chill\"");
}
diagList = db.diagnosis ( );
testArraySetEquals (diagList, ague);
missingList = db.missingSymptoms ("all-nighter");
assertTrue (missingList == null);
missingList = db.missingSymptoms ("plague");
testArraySetEquals (missingList, buboes);
missingList = db.missingSymptoms ("ague");
testArraySetEquals (missingList, empty);

try {
    db.addSymptom ("shortness_of_breath");
} catch (UnrecognizedSymptomException e) {
    fail ("unrecognized symptom incorrectly thrown"
      + " for \"shortness_of_breath\"");
}
diagList = db.diagnosis ( );
testArraySetEquals (diagList, fourDiseases);
missingList = db.missingSymptoms ("all-nighter");
testArraySetEquals (missingList, empty);
missingList = db.missingSymptoms ("plague");
testArraySetEquals (missingList, buboes);
missingList = db.missingSymptoms ("ague");
testArraySetEquals (missingList, empty);
missingList = db.missingSymptoms ("exercise");
testArraySetEquals (missingList, empty);

db.reset();
diagList = db.diagnosis();
testArraySetEquals (diagList, empty);
```

```
        missingList = db.missingSymptoms("plague");
        assertTrue (missingList == null);
    }

    private void testArraySetEquals (String [ ] set, String [ ] arg) {
        assertTrue (set.length == arg.length);
        for (int k = 0; k < arg.length; k++) {
            assertTrue (contains (set, arg[k]));
        }
    }

    private boolean contains (String [ ] set, String s) {
        for (int k=0; k<set.length; k++) {
            if (set[k].equals (s)) {
                return true;
            }
        }
        return false;
    }
}
```

Appendix C
FileGen.java

FileGen.java

```
import java.io.*;

public class FileGen {

    private final static int symptomCount = 10000;
    private final static int diseaseCount = 5000;                    // symptomCount/2
    private final static int diseasesPerSymptom = 400;    // symptomCount/100 for large set
    private final static int symptomsPerDisease = 200;          // diseasesPerSymptom/2
    private final static int patientSymptomCount = 200;

    public static void main (String [ ] args) throws Exception {
        // arg 0 is name of symptom info file
        // arg 1 is name of disease info file
        // arg 2 is name of patient symptoms

        boolean [ ][ ] table = new boolean [symptomCount] [diseaseCount];
        for (int i = 0; i < symptomCount; i++) {
            for (int j = 0; j < diseaseCount; j++) {
                table[i][j] = false;
            }
        }
        PrintWriter symptomInfoFile = new PrintWriter (new FileWriter (args[0]), true);
        for (int i = 0; i < symptomCount; i++) {
            symptomInfoFile.print ("s" + i);
            for (int j = 0; j < diseasesPerSymptom; j++) {
                symptomInfoFile.print (" d" + (i + j) % diseaseCount);
                table[i][ (i + j) % diseaseCount] = true;
            }
            symptomInfoFile.println ( );
        }
        PrintWriter diseaseInfoFile = new PrintWriter (new FileWriter (args[1]), true);
        for (int j = 0; j < diseaseCount; j++) {
            diseaseInfoFile.print ("d" + j);
            int n = 0;
            for (int i = 0; i < symptomCount; i++) {
                if (table[i][j]) {
                    diseaseInfoFile.print (" s" + i);
                    n++;
                    if (n == symptomsPerDisease) {
                        break;
                    }
                }
            }
            diseaseInfoFile.println ( );
        }
        PrintWriter patientSymptomFile
            = new PrintWriter (new FileWriter (args[2]), true);
        for (int i = 0; i < patientSymptomCount; i++) {
            patientSymptomFile.println ("s" + i);
        }
    }
}
```