Chapter II

Introduction to Basic Local Alignment Search Tool

The Basic Local Alignment Search Tool or BLAST, as it is commonly referred to as, is a database search tool, developed and maintained by the National Center for Biotechnology Information (NCBI). The web-based tool for BLAST searches is available at:

```
http://www.ncbi.nlm.nih.gov/BLAST/
```

The BLAST suite of programs has been designed to find high scoring local alignments between sequences, without compromising the speed of such searches. BLAST uses a heuristic algorithm which seeks local as opposed to global alignments and is therefore able to detect relationships among sequences which share only isolated regions of similarity (Altschul et al., 1990). The first version of BLAST was released in 1990 and allowed users to perform ungapped searches only. The second version of BLAST, released is 1997, allowed gapped searches (Altschul et al., 1997).

The Purpose of BLAST

Why is BLAST so useful for biologists? It is not uncommon nowadays, especially with the large number of genomes being sequenced, that a researcher comes across a novel DNA or protein sequence for which no functional data is available. Some basic information on the sequence is necessary before a molecular biologist can take the new sequence into the laboratory and perform meaningful experiments with it. It would, for example, make the task of deciphering the biological function of a piece of DNA much easier if it were known that the new sequence encoded a *metabolic enzyme* or, indeed, a protein that is a putative member of a *superfamily* such as an *immunoglobulin*, a *kinase*, etc. Conversely, if the sequence was a Repetitive DNA Element, it would need an entirely different approach for its study.

This is where the power of database searching comes in handy. The principle aim of database searching, in general and with BLAST, in particular, is to reveal the existence of similarity between an input sequence (called 'query sequence') that a user wants to find more information about and other sequences (called 'target' sequences) that are stored in a biological database. This is usually the first step a researcher takes in determining the biological significance of an unknown sequence.

Given the size of biological sequence databases maintained by NCBI (the non-redundant set of sequences were estimated at 540 million residues in 2004), database searches usually reveal sequences that have some degree of similarity to the query sequence. These sequences from the database that come up with similarities with the input sequence are commonly referred to as 'hits'. Once such hits are found, users can draw inferences about the putative molecular function of the query sequence. A thumb rule for drawing inferences is that two sequences that share more than 50 per cent sequence identity are usually similar in structure and function. Under such conditions, the major sequence features of the two sequences can be easily aligned and identified. If there is only a 25 per cent sequence identity, there may be some structural homology, although in such situations, the domain correspondence between the two proteins may not be easily apparent. It is also generally accepted that sequences that are important for function (and therefore, for the survival of an organism or species) are generally conserved.

An example where a database search resulted in an important discovery was the finding reported by Doolittle et al. (1983) of the similarity between the *oncogene*, *v-sis*, of *Simian sarcoma virus* (an *RNA tumor virus*) and the gene encoding *human platelet-derived growth factor* (PDGF). The v-sis gene was the first oncogene to be identified with homology to a known cellular gene. This discovery provided an early insight into the critical role that growth factor signaling plays in the process of malignant transformation. Another example of the value of database searching was the discovery that the defective gene that caused *cystic fibrosis* formed a protein that had similarity to a family of proteins involved in the transport of hydrophilic molecules across the cytoplasmic membrane (Riordan, et. al., 1989). Cystic fibrosis is the most common inherited disease in the Caucasian population and affects the respiratory, digestive and reproductive systems. It is now known that mutations in the cystic fibrosis gene lead to loss of chloride transport across the cell membrane, which is the underlying cause of the disease.

Performing a BLAST Analysis

Before we can build a BLAST application, we need to understand how BLAST searches are performed using the NCBI BLAST service. BLAST is actually a suite of programs – the particular choice of program(s) depends on the type of input sequence (amino acid or nucleotide) and the type of the database to be searched against (protein or nucleotide). The most commonly used search programs and their applications are described in **Table 2.1.**

Table 2.1. BLAST programs

Program	Comparison	Application
BLASTN	DNA vs. DNA. Compares a nucleotide query sequence against a nucleotide sequence database.	Find DNA sequences that match the query
BLASTP	Protein vs. Protein. Compares an amino acid query sequence against a protein sequence database.	Find identical (homologous) proteins
BLASTX	DNA vs. Protein. Compares a nucleotide query sequence translated in all reading frames against a protein sequence database.	Find which protein the query sequence codes for
TBLASTN	Protein vs. DNA Compares a protein query sequence against a nucleotide sequence database dynamically translated in all reading frames.	Find genes in unknown DNA sequences
TBLASTX	DNA vs. DNA Compares the six-frame translations of a nucleotide query sequence against the six-frame translations of a nucleotide sequence database.	Discover gene structure. (Find degree of homology between the coding region of the query sequence and known genes in the database.)

In summary, the available BLAST options are:

1. For nucleotide sequences: BLASTN, BLASTX and TBLASTX

2. For amino acid sequences: BLASTP and TBLASTN

In the simplest case, we need the following pieces of information to perform a BLAST search using NCBI's web-based service (http://www.ncbi.nlm.nih.gov/BLAST/):

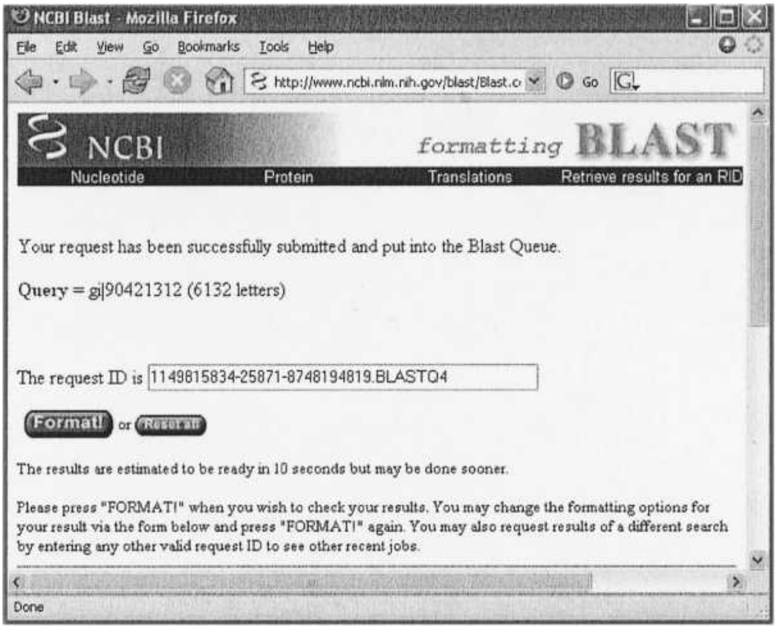
1. An input query sequence (this can be a nucleotide or amino acid)

2. The database to search against (this can be a nucleotide or protein database)

3. A database search program (any of the five available BLAST options)

Additional parameters such as the matrix and E-values also need to be set. Once the user submits the necessary information, the BLAST engine responds with a message informing the user that the request has been successfully submitted and placed in a queue. The server also provides an estimate of the time in which the results will become available for viewing. The BLAST output itself consists of a header that provides information on the specified BLAST parameters, the request ID for the search, the length of the query sequence and the database used. **Fig. 2.1 - 2.3** show the results immediately after initial submission of and the output of a BLAST search performed with the *human cystic fibrosis transmembrane conductance regulator* (CFTR) mRNA sequence (gi: 90421312). **Fig. 2.1** and **Fig. 2.2** show the request ID (RID) that uniquely identifies this particular search job that was submitted to the BLAST queue. We will learn more about RID in Chapter 3 when we build the functionality to perform BLAST searches using the NCBI QBlast service. **Fig. 2.2** provides a view of the header information present in the BLAST search results.

Below the header is a line up of sequences from the selected database ("hits") that match the query sequence along with the number of matches found (**Fig. 2.3**). A mouse-over on the first line reveals information on the origin of the sequence (for example, whether it is a human or a mouse sequence, the name of the gene, if known) and the score (**Fig. 2.4**). Sequences on the top are more significant (have better matches to sequences in the database and thus, have higher scores) than those at the bottom (lower scores).


NCBI Blast - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://www.ncbi.nlm.nih.gov/blast/Blast.c
Go
NCBI
formatting BLAST
Nucleotide
Protein
Translations
Retrieve results for an RID
Format!
Reset all
Done

Your request has been successfully submitted and put into the Blast Queue.

Query = gi|90421312 (6132 letters)

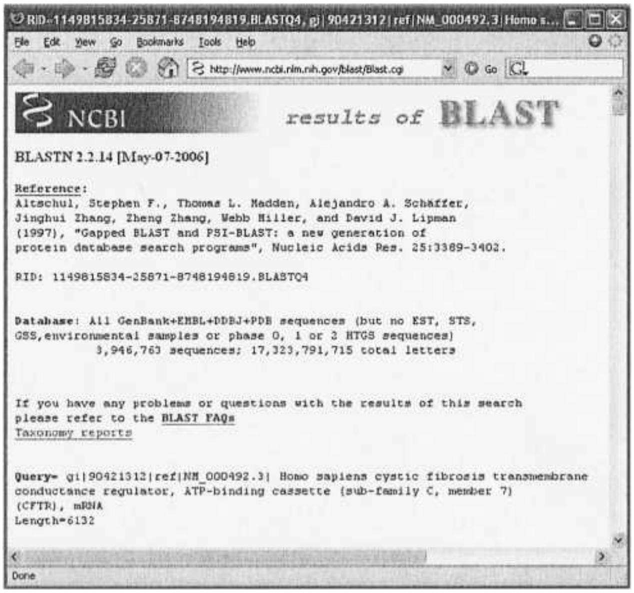
The request ID is 1149815834-25871-8748194819.BLASTQ4

Format! or Reset all

The results are estimated to be ready in 10 seconds but may be done sooner.

Please press "FORMAT!" when you wish to check your results. You may change the formatting options for your result via the form below and press "FORMAT!" again. You may also request results of a different search by entering any other valid request ID to see other recent jobs.

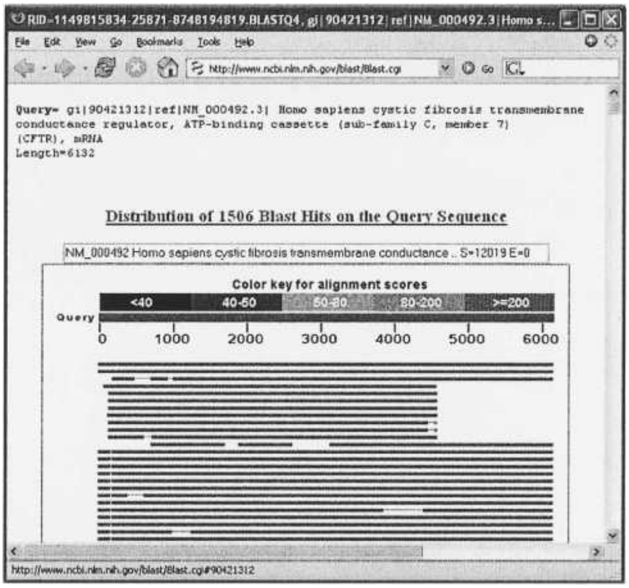
Fig. 2.1. Submission of a sequence to the BLAST queue


RID=1149815834-25871-8748194819.BLASTQ4, gi|90421312|ref|NM_000492.3|Homo s...
File Edit View Go Bookmarks Tools Help
http://www.ncbi.nlm.nih.gov/blast/Blast.cgi
Go
NCBI
results of BLAST
Done

BLASTN 2.2.14 [May-07-2006]

```
Reference:
Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schäffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman
(1997), "Gapped BLAST and PSI-BLAST: a new generation of
protein database search programs", Nucleic Acids Res. 25:3389-3402.

RID: 1149815834-25871-8748194819.BLASTQ4


Database: All GenBank+EMBL+DDBJ+PDB sequences (but no EST, STS,
GSS,environmental samples or phase 0, 1 or 2 HTGS sequences)
           3,946,763 sequences; 17,323,791,715 total letters



If you have any problems or questions with the results of this search
please refer to the BLAST FAQs
Taxonomy reports


Query= gi|90421312|ref|NM_000492.3| Homo sapiens cystic fibrosis transmembrane
conductance regulator, ATP-binding cassette (sub-family C, member 7)
(CFTR), mRNA
Length=6132
```

Fig. 2.2. Header information in BLAST search results


RID=1149815834-25871-8748194819.BLASTQ4, gi|90421312|ref|NM_000492.3|Homo s...
File Edit View Go Bookmarks Tools Help
http://www.ncbi.nlm.nih.gov/blast/Blast.cgi
Go

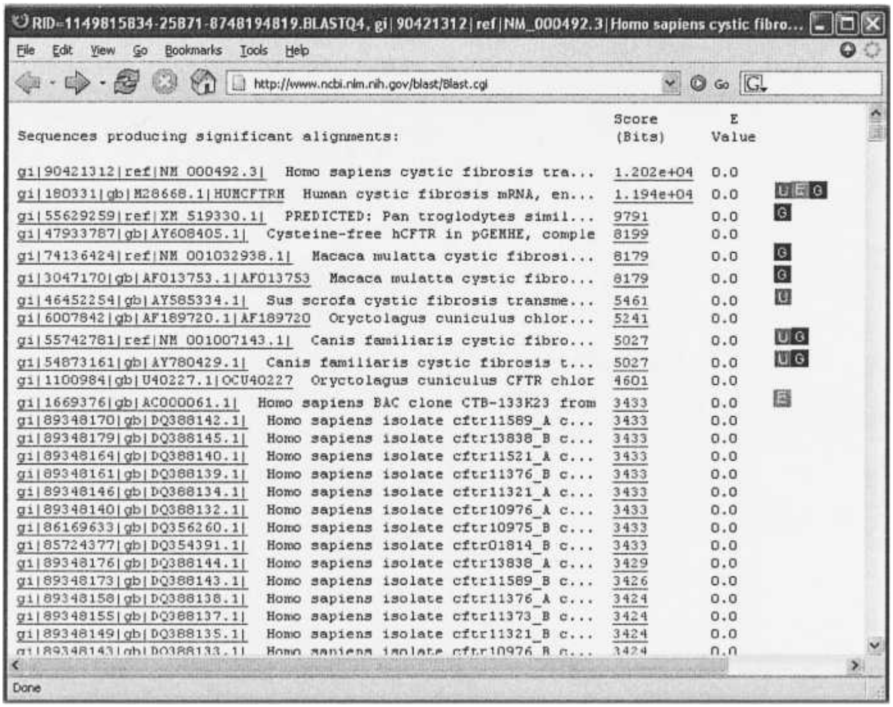
```
Query= gi|90421312|ref|NM_000492.3| Homo sapiens cystic fibrosis transmembrane
conductance regulator, ATP-binding cassette (sub-family C, member 7)
(CFTR), mRNA
Length=6132
```

Distribution of 1506 Blast Hits on the Query Sequence

NM_000492 Homo sapiens cystic fibrosis transmembrane conductance .. S=12019 E=0

Color key for alignment scores
<40
40-50
50-80
80-200
>=200
Query
0
1000
2000
3000
4000
5000
6000
http://www.ncbi.nlm.nih.gov/blast/Blast.cgi#90421312

Fig. 2.3. Alignment of BLAST hits to the input sequence


RID=1149815834-25871-8748194819.BLASTQ4, gi|90421312|ref|NM_000492.3|Homo sapiens cystic fibro...
File Edit View Go Bookmarks Tools Help
http://www.ncbi.nlm.nih.gov/blast/Blast.cgi
Go
Done

```
                                                                   Score     E
Sequences producing significant alignments:                       (Bits)    Value

gi|90421312|ref|NM 000492.3|  Homo sapiens cystic fibrosis tra...  1.202e+04  0.0
gi|180331|gb|M28668.1|HUMCFTRM  Human cystic fibrosis mRNA, en...  1.194e+04  0.0    U E G
gi|55629259|ref|XM 519330.1|  PREDICTED: Pan troglodytes simil...  9791       0.0    G
gi|47933787|gb|AY608405.1|  Cysteine-free hCFTR in pGEMHE, comple  8199       0.0
gi|74136424|ref|NM 001032938.1|  Macaca mulatta cystic fibrosi...  8179       0.0    G
gi|3047170|gb|AF013753.1|AF013753  Macaca mulatta cystic fibro...  8179       0.0    G
gi|46452254|gb|AY585334.1|  Sus scrofa cystic fibrosis transme...  5461       0.0    U
gi|6007842|gb|AF189720.1|AF189720  Oryctolagus cuniculus chlor...  5241       0.0
gi|55742781|ref|NM 001007143.1|  Canis familiaris cystic fibro...  5027       0.0    U G
gi|54873161|gb|AY780429.1|  Canis familiaris cystic fibrosis t...  5027       0.0    U G
gi|1100984|gb|U40227.1|OCU40227  Oryctolagus cuniculus CFTR chlor  4601       0.0
gi|1669376|gb|AC000061.1|  Homo sapiens BAC clone CTB-133K23 from  3433       0.0    E
gi|89348170|gb|DQ388142.1|  Homo sapiens isolate cftr11589_A c...  3433       0.0
gi|89348179|gb|DQ388145.1|  Homo sapiens isolate cftr13838_B c...  3433       0.0
gi|89348164|gb|DQ388140.1|  Homo sapiens isolate cftr11521_A c...  3433       0.0
gi|89348161|gb|DQ388139.1|  Homo sapiens isolate cftr11376_B c...  3433       0.0
gi|89348146|gb|DQ388134.1|  Homo sapiens isolate cftr11321_A c...  3433       0.0
gi|89348140|gb|DQ388132.1|  Homo sapiens isolate cftr10976_A c...  3433       0.0
gi|86169633|gb|DQ356260.1|  Homo sapiens isolate cftr10975_B c...  3433       0.0
gi|85724377|gb|DQ354391.1|  Homo sapiens isolate cftr01814_B c...  3433       0.0
gi|89348176|gb|DQ388144.1|  Homo sapiens isolate cftr13838_A c...  3429       0.0
gi|89348173|gb|DQ388143.1|  Homo sapiens isolate cftr11589_B c...  3426       0.0
gi|89348158|gb|DQ388138.1|  Homo sapiens isolate cftr11376_A c...  3424       0.0
gi|89348155|gb|DQ388137.1|  Homo sapiens isolate cftr11373_B c...  3424       0.0
gi|89348149|gb|DQ388135.1|  Homo sapiens isolate cftr11321_B c...  3424       0.0
gi|89348143|gb|DQ388133.1|  Homo sapiens isolate cftr10976_B c...  3424       0.0
```

Fig. 2.4. Definition of database hits

Developing the SwingBlast Application

Now that we understand the significance and the working of the BLAST engine, we can begin our journey into the world of Java development by building a BLAST application, which we will call *SwingBlast*, from the ground up. In this Chapter, we will create the user interface elements using *Java Foundation Classes* or *JFC*, also known as *Abstract Windowing Toolkit* (*AWT)* and *Swing* classes. In Chapter 3, we will write the actual code to run the BLAST searches based on the NCBI BLAST engine. In each case, we will build the application in an iterative fashion thereby demonstrating a step-wise approach to building software - creating a basic program structure or framework and adding bits of code in an incremental fashion to enhance its functionality.

The steps for building Java applications from a software engineering point-of-view are as follows:

1. Develop use case scenarios
2. Define software modules
3. Define classes
4. Write the Java code (business logic)
5. Run and analyze output

We will begin by creating *use cases* that define the actions that a user may wish to perform on the application and the behavior that a user expects from the application in response to those actions. Use cases, simply stated, are individual scenarios that allow software developers to layout the behavior and functionality expected of the software. To create a Java based BLAST application that allows users to submit sequences and to retrieve the results of the search operation, we can envision the following use case scenarios:

1. User provides input information to the application
2. User submits the input information to the NCBI BLAST server

3. The application displays the selected BLAST results in graphical format

Fig. 2.5 provides a UML diagram that describes the interactions between the user and the application. The specific details about the expected input and output are as follows:

1. User provides input information to the NCBI BLAST engine: The input data can be a sequence or, if available, the corresponding sequence id from GenBank® (an annotated repository of all publicly available DNA sequences maintained by the NIH), which uniquely identifies a sequence within the GenBank database. The application behavior in either case is as follows:

   a. The input information is a nucleotide or protein sequence: In this case, after the sequence information is provided, the application automatically recognizes the sequence type, loads it in the Fasta format (**Fig. 2.6**) and presents the appropriate valid BLAST options (for example, BLASTN for nucleotide and BLASTP for protein etc., as explained in **Table 2.1**). The invalid BLAST options are disabled.

   b. The input information is a valid GenBank id (also called the GI number). In this case, the application downloads the sequence from GenBank and displays it in the appropriate format as stated above.

2. User submits the sequence to the NCBI BLAST server. Once the sequence becomes available to the application (either directly supplied by the user or downloaded from the GenBank id), the user selects the necessary BLAST parameters (the type of BLAST program, the database, the matrix, the E values, etc.) and hits the "Submit" button. This sends the sequence to the NCBI BLAST server for the search operation.

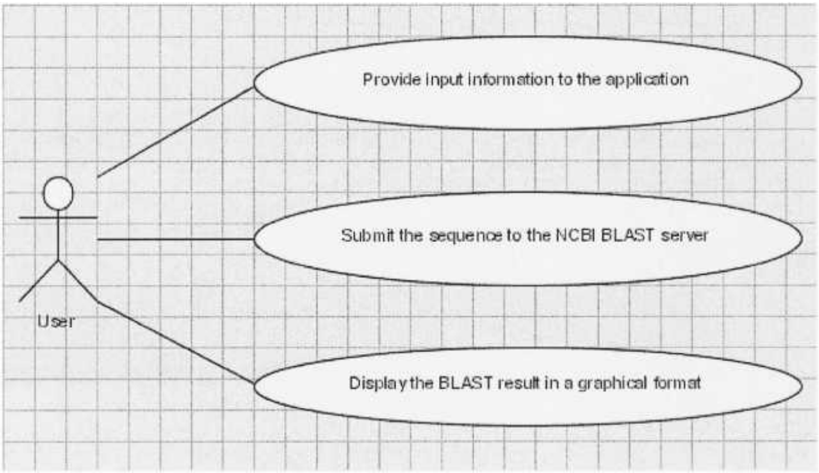
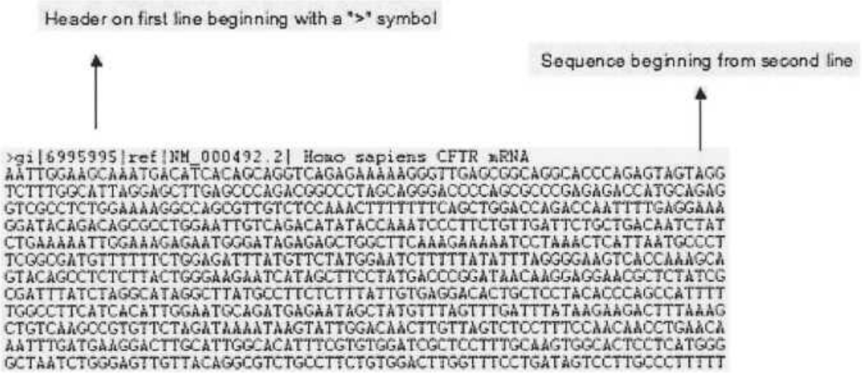

Provide input information to the application
Submit the sequence to the NCBI BLAST server
Display the BLAST result in a graphical format
User

Fig. 2.5. UML diagram for the SwingBlast use cases

The last use case ("User wants to browse the BLAST results in a graphical format") arises from a need to view the BLAST output, that is, the list of sequences from the database that matched the input sequence in a graphical and interactive fashion.


Header on first line beginning with a ">" symbol
Sequence beginning from second line

```
>gi|6995995|ref|NM_000492.2| Homo sapiens CFTR mRNA
AATTGGAAGCAAATGACATCACAGCAGGTCAGAGAAAAAGGGTTGAGCGGCAGGCACCCAGAGTAGTAGG
TCTTTGGCATTAGGAGCTTGAGCCCAGACGGCCCTAGCAGGGACCCCAGCGCCCGAGAGACCATGCAGAG
GTCGCCTCTGGAAAAGGCCAGCGTTGTCTCCAAACTTTTTTTCAGCTGGACCAGACCAATTTTGAGGAAA
GGATACAGACAGCGCCTGGAATTGTCAGACATATACCAAATCCCTTCTGTTGATTCTGCTGACAATCTAT
CTGAAAAATTGGAAAGAGAATGGGATAGAGAGCTGGCTTCAAAGAAAAATCCTAAACTCATTAATGCCCT
TCGGCGATGTTTTTTCTGGAGATTTATGTTCTATGGAATCTTTTTATATTTAGGGGAAGTCACCAAAGCA
GTACAGCCTCTCTTACTGGGAAGAATCATAGCTTCCTATGACCCGGATAACAAGGAGGAACGCTCTATCG
CGATTTATCTAGGCATAGGCTTATGCCTTCTCTTTATTGTGAGGACACTGCTCCTACACCCAGCCATTTT
TGGCCTTCATCACATTGGAATGCAGATGAGAATAGCTATGTTTAGTTTGATTTATAAGAAGACTTTAAAG
CTGTCAAGCCGTGTTCTAGATAAAATAAGTATTGGACAACTTGTTAGTCTCCTTTCCAACAACCTGAACA
AATTTGATGAAGGACTTGCATTGGCACATTTCGTGTGGATCGCTCCTTTGCAAGTGGCACTCCTCATGGG
GCTAATCTGGGAGTTGTTACAGGCGTCTGCCTTCTGTGGACTTGGTTTCCTGATAGTCCTTGCCCTTTTT
```

Fig. 2.6. A sequence represented in Fasta format

Designing the SwingBlast Java Application

The SwingBlast application involves data input from the user (the sequence or the GI number which identifies the sequence), manipulation of the input data ("BLASTing" the sequence against the selected databases), and visualization of the results of the database search (the BLAST output). Clearly, there are different parts to the application each of which performs a different function. We will follow the MVC framework we described in Chapter 1, while designing the various pieces of functionality of the SwingBlast application.

In line with the incremental approach to building the SwingBlast application, we will as a first step, create the basic framework application that will perform two basic functions - allow users to input a nucleotide sequence to the application and to format it in the Fasta format. The structure of the Java application we will build is shown in **Fig. 2.7** below.

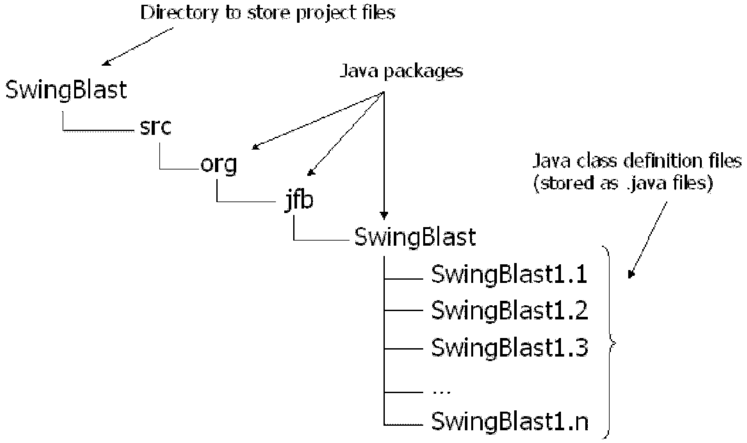


Fig. 2.7. Layout of the SwingBlast application

As depicted in **Fig. 2.7**, we define a project directory called SwingBlast to store the project files. We create a src (source) directory, in which we will create the packages org, org.jfb and org.jfb.SwingBlast to provide a default hierarchy for the class files. This layout also helps to group the necessary functionalities of the application, for example, by placing all the GUI classes in the SwingBlast package, all the source code files in the src directory and so on . SwingBlast1.1, ..., 1.n, etc., are the Java class definition files, where the numbers refer to versions of the software as we build functionality step-by-step. For the SwingBlast application, the package name we will use in our Java class definition files will be `org.jfb.SwingBlast`. After the package is declared, we name any `import` statements to be included in the program. Import statements load the classes that encapsulate functions necessary for the application to run. Since classes are contained in packages for the purpose of grouping common functionalities together, entire packages may be imported, if necessary. By using wildcards with import statements for example,

```
import java.awt.*;
```

we can ensure that all classes in the AWT package, which provide the Java graphical user interface elements, are available to the application.

As we mentioned earlier, the `SwingBlast` application takes data input from the user and responds to the input by taking appropriate actions. To make the application respond appropriately to user initiated actions, we need to add what are known as *event listeners* to the code. This functionality allows us to add *events* to menu buttons that respond to simple actions such as clear user input or quit the application, etc., as well as complex functionality, some of which we will demonstrate in this Chapter. To begin with, we will learn the basics of the Java event model and see how to add events and event listeners in the next few sections.

Java Event Model

The Java Event Model is based on the *Observer design pattern* also known as the *Publish-Subscribe design pattern* and a delegation model that allows a source to propagate an event to the relevant observer. The *Publish-Subscribe design pattern* is based on the Observer pattern where the *Observer* object listens for events from the *Subject* object. The Publish-Subscribe design pattern is similar to the Observer design pattern except for additional element called the *Event Channel* that separates the Observer (called *Subscriber* in the Publish-Subscribe design pattern) and the Subject (called *Publisher* in the Publish-Subscribe design pattern). The Event Channel performs the role of a messaging hub to broadcast events from Publishers to all the associated Subscribers.

Java uses what are known as *EventListener* objects to listen to changes to AWT or Swing components. Under this model, observers can be registered to listen to an object via *Listener* methods depending on the type of the listener or the kinds of events one is interested in. The general format for such methods is `addXXXListener()`, for example, `addMouseListener(MouseListener l)`, which is a method to listen to any mouse event generated by the object the listener is registered to. The listener object provides a *callback* method that is called by the object that is generating the event. The callback method will have the appropriate parameters that define such data as the source (for example, JButton, JPanel, or a main window, etc.) and type of event (for example, a mouse click event, or a focus event when selecting a particular Swing component or an action event, like pressing a submit button).

In Java, all events are executed in the same thread as the window painting event (via `paint()`). This thread is called the *event-dispatching thread.* For this reason, code in an event listener should be fast to execute to avoid interference with the drawing events.

Two types of events are defined in Java: *low-level* events and *semantic* events. Low-level events represent system related events that emerge from objects such as mouse and keyboard, etc., while semantic events arise from operations such as clicking on a button, selecting a text in a drop down box, etc. Depending on the situation, it is advisable to listen to semantic events whenever possible since they are more specific in nature - for example, listening for a button event inside the component that contains the button, rather than a mouse event, which can occur outside of a component.

Adding Events to Applications

To add events to applications, we will need to add two import statements at the beginning of our code:

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;
```

These Java packages provide the classes that are needed for triggering and handling events. Let's take the example of making the SwingBlast application respond to actions initiated by the user by clicking on the `Quit` button under the `SwingBlast` Menu. To create the Quit button, we create an object called quitItem of type JMenuItem with the following piece of code:

```
quitItem = new JMenuItem("Quit");
```

To associate quitItem with a mouse click event that leads to closing the application, we first instantiate an ActionListener. Next we register the new listener to receive events from this button by calling the button's addActionListener method:

```
quitItem.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
            System.exit(0);
     }
});
```

Actions triggered by mouse events such as a button click will also call the actionPerformed method from that listener and pass it an ActionEvent object as shown in the code above. That ActionEvent object contains all the properties of this event. In the early days, in C, you would have to catch the system interrupts and analyze the interrupt number received to figure out the type of event (viz., a keyboard or a mouse action or a USB port sending or receiving information, etc.). In Java, Swing does that for you by encapsulating all the hardware interactions into its event framework. This is undoubtedly much easier and means less work for the Java coder. Inside that actionPerformed method, all we need to do is to simply read the ActionEvent properties and code the appropriate action to respond to the event.

The code to handle events associated with the Clear button is constructed in a similar manner. The text box to enter sequences was earlier created as an object of type JTextArea using the code:

```
sequenceArea = new JTextArea();
```

The event handling code for the Clear button is similar, except that the exact action specified is that the text in the sequenceArea box is set to nothing (""):

```
clearButton.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
            sequenceArea.setText("");
     }
});
```

Designing the SwingBlast GUI

We can now create the first version (1.1) of the SwingBlast application. SwingBlast version 1.1 will have a text box to enter sequence data, a Clear button to delete the entered sequence and a menu bar for quitting the application (**Fig. 2.8**).

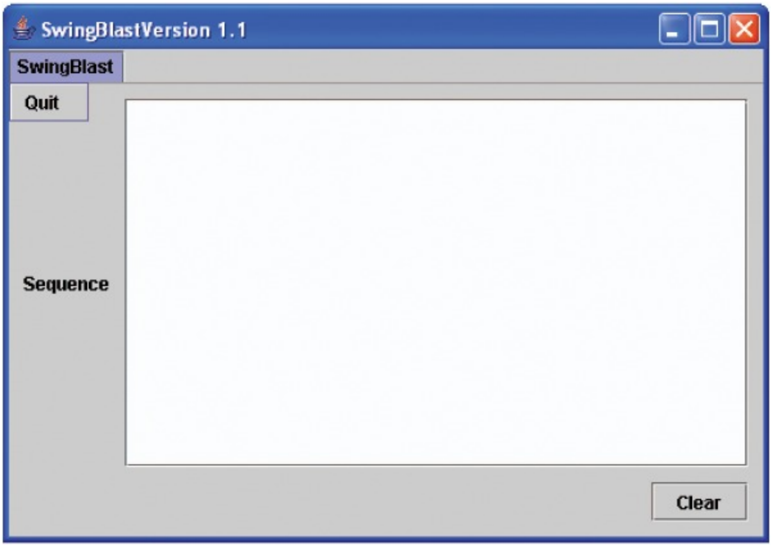


Fig. 2.8. SwingBlast Version 1.1

Let's now write the code that will create SwingBlast version 1.1. At the most basic level, our code will look like **Listing 2.1**.

Listing 2.1. Coding SwingBlast version 1.1

```
  package org.jfb.SwingBlast;

  import javax.swing.*;
  import java.awt.*;

  public class SwingBlast1_1 extends JFrame {
      private static final String APP_NAME = "SwingBlast";
      private  static  final  String  APP_VERSION  =  "Version
1.1";
      private  static  final  Dimension  APP_WINDOW_SIZE  =  new
Dimension(500, 300);

      private JComponent newContentPane;
      private JTextArea sequenceArea;
      private JScrollPane scrollPaneArea;
      private JButton clearButton;
      private JMenuItem quitItem;

      public SwingBlast1_1() {
          super(APP_NAME + APP_VERSION);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          newContentPane = new JPanel();
          newContentPane.setLayout(new BorderLayout());
          setContentPane(newContentPane);

          JMenuBar menu = new JMenuBar();
          JMenu swingBlastMenu = new JMenu(APP_NAME);
          quitItem = new JMenuItem("Quit");
          swingBlastMenu.add(quitItem);
          menu.add(swingBlastMenu);
          setJMenuBar(menu);

          // The sequence pane
          JPanel sequencePanel = new JPanel();
          JLabel sequence = new JLabel("Sequence");
          sequenceArea = new JTextArea();
          sequenceArea.setLineWrap(true);
          scrollPaneArea = new JScrollPane(sequenceArea);
          sequencePanel.setLayout(new
BoxLayout(sequencePanel, BoxLayout.LINE_AXIS));
          sequencePanel.add(sequence);
          sequencePanel.add(Box.createRigidArea(new
Dimension(10, 0)));
          sequencePanel.add(scrollPaneArea);

sequencePanel.setBorder(BorderFactory.createEmptyBorder(10,
0, 10, 0));

          //Lay out the buttons from left to right
JPanel buttonPane = new JPanel();
          clearButton = new JButton("Clear");

          buttonPane.setLayout(new BoxLayout(buttonPane,
BoxLayout.LINE_AXIS));
          buttonPane.add(Box.createHorizontalGlue());
          buttonPane.add(Box.createRigidArea(new
Dimension(10, 0)));
          buttonPane.add(clearButton);

          JPanel jPanel = new JPanel();
          jPanel.setLayout(new BorderLayout());
          jPanel.setBorder(BorderFactory.createEmptyBorder(0,
10, 10, 10));
          jPanel.add(sequencePanel, BorderLayout.CENTER);
          jPanel.add(buttonPane, BorderLayout.SOUTH);

          newContentPane.add(jPanel, BorderLayout.CENTER);
          newContentPane.setPreferredSize(APP_WINDOW_SIZE);
          //Display the window
          pack();
          Dimension                screenSize                  =
Toolkit.getDefaultToolkit().getScreenSize();
          setLocation((screenSize.width                        -
APP_WINDOW_SIZE.width) / 2,
                      (screenSize.height                       -
APP_WINDOW_SIZE.height) / 2);
          setVisible(true);

      }

      public static void main(String[] args) {
          SwingUtilities.invokeLater(new Runnable() {
              public void run() {
                  final    SwingBlast1_1    view    =    new
SwingBlast1_1();
              }
          });
      }
  }
```

As described earlier, we begin by declaring a package, which in this case is named after the `SwingBlast` application that we are building. The common prefix `jfb` is short for Java for Bioinformatics. Since we are creating a Swing based GUI to manage sequence input and analysis, we have named the class "SwingBlast". The suffix 1_1 at the end of the class name reflects the fact that this is version 1.1 of the `SwingBlast` application.

A simplified general format of the class declaration is as follows:

```
class_modifiers class <class_name> extends <superclass_name>
       {
             /* list of class data fields */
             /* list of class methods     */
       }
```

In our case, the modifier for the `SwingBlast1_1` class is *public* which means other methods or classes outside of this class may access this class:

```
  public class SwingBlast1_1 extends JFrame {
    ...
  }
```

By convention, there can be only one *public* class in a Java file; further, the name of the Java file must match the name of the *public* class. For this reason, the code in **Listing 2.1** must be stored in a file called SwingBlast1_1.java.

Note the use of the extends keyword in the class declaration. The extends keyword indicates that the SwingBlast1_1 class inherits methods from the class *JFrame*. In object oriented terminology, SwingBlast1_1 is called the *sub* or *child* class while *JFrame* which it derives from is called the *parent* (or *super*) class. The extends keyword obviates the need for instantiating *JFrame* separately in the SwingBlast1_1 class to access its methods. Inside the SwingBlast1_1 class, we can call any of the methods available in the parent *JFrame* class.

JFrame is a *Swing container* that serves as the top-level or main application window. Top-level Swing containers provide space within which other Swing components can position and draw themselves. Swing components are also called "*lightweight components*" because they are written in Java versus *AWT* components or "*heavyweight components*" which are native components (written in C or C++, etc.) wrapped into Java classes. It is important to know what class of components are being used. As a rule of thumb, Swing and AWT components should not be mixed or used together in the same application, as this may lead to unpredictable behavior during repainting, and may make the application hard to debug.

The Swing framework provides a mechanism for interactions with individual components through *event handling*. This is what the two import statements at the top of our code in **Listing 2.1** do:

```
import javax.swing.*;

import java.awt.*;
```

The first package provides a set of lightweight components while the second contains all classes for dealing with graphics, events, images, etc. **Fig. 2.9** shows the superclass hierarchy of the JFrame class where each subclass is shown below its parent class. According to this scheme, the *JFrame* class is derived from the *Frame* class, which in turn is derived from the *Window* class and so on. The *Frame* class defines a top-level window with a title and a border and has methods such as getTitle, setTitle etc., which respectively get and set the title of the frame. By definition, the JFrame class derives these methods from the *Frame* class (and other superclasses). Every main window that contains Swing components should be implemented with a JFrame. Examples of other containers are *JApplet*, *JWindow* and *JDialog*. In our application, JFrame will serve as the top-level container. JFrame in turn will provide the framework to contain other components like for example *JPanel*, *JButton* and *JMenu*, etc.

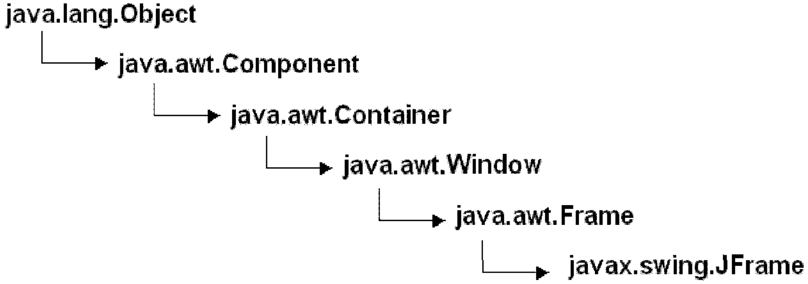


Fig. 2.9. Class hierarchy of the JFrame class

The next three lines of code define constants for setting the name (`SwingBlast`), version (1.1) and the window size (500 x 300 pixels) of the application. We will use upper case names separated with underscores '_', as a naming convention for our constants :

```
private static final String APP_NAME = "SwingBlast";
```

private limits the accessibility of the variable called `APP_NAME` to other objects within the same class. The keyword *Static* means that the value of the variable is shared by any object of that same class (this also defines what is known as the class variable). This means that if one object modifies it, the other object can see the new value. A non-static variable, on the other hand, is modifiable only by the object instantiated from within the same class. The keyword *final* means that the variable cannot be changed and therefore it is a constant. The constants `APP_NAME` and `APP_VERSION` are of type *String* as indicated in the code. To summarize, `APP_NAME` is a constant accessible only from within the class and it has the same value for any object belonging to this class.

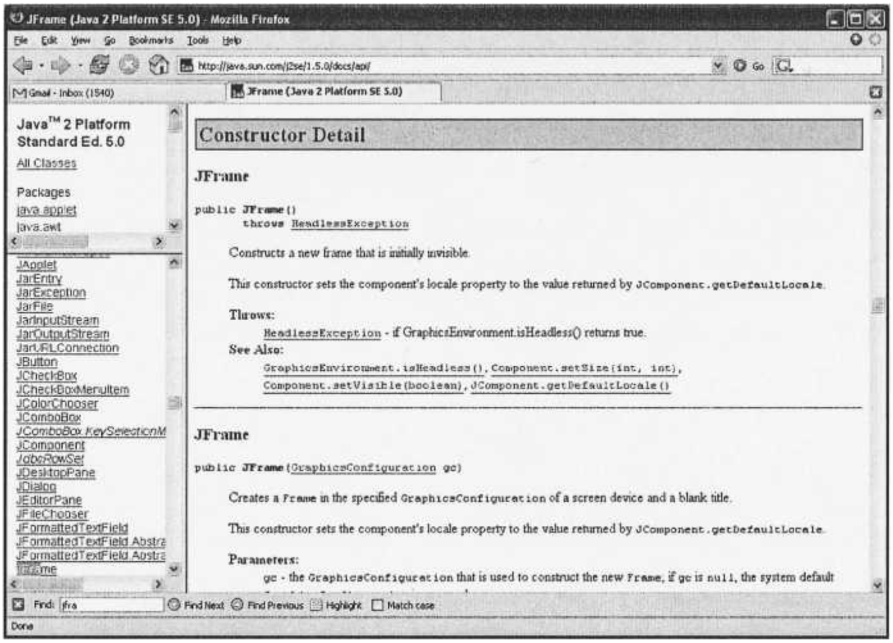
The next 5 lines declare Swing components of the types *JComponent*, *JTextArea*, *JScrollPane*, *JButton* and *JMenuItem* respectively. All Swing components (except top-level containers) whose names begin with "J" are derived from and inherit from the *JComponent* class such as *JTextArea*, *JPanel*, *JScrollPane*, *JButton*, and *JmenuItem*. *JComponent* is thus the base class for all these Swing components.

The next line:

```
public SwingBlast1_1() {
```

defines the constructor for the `SwingBlast1_1` class. Note that it is declared *public*, has the same name as the class itself and does not return anything. The `SwingBlast1_1` constructor also does not accept any parameters and therefore is the default constructor for the `SwingBlast1_1` class.

The *super* keyword in the `SwingBlast1_1` constructor calls the constructor of the superclass (hence the use of the term "*super*") - which in this case is JFrame, since `SwingBlast1_1` "extends" JFrame. Next it passes the *String* variables `APP_NAME` and `APP_VERSION` to the JFrame constructor to set the name and version of the application. The description of the JFrame constructor that is used is shown below. This information is available from the Java 2 API documentation (**Fig. 2.10**).



Constructor Detail

JFrame

```
public JFrame()
       throws HeadlessException
```

Constructs a new frame that is initially invisible.

This constructor sets the component's locale property to the value returned by `JComponent.getDefaultLocale`.

Throws:
`HeadlessException` - if GraphicsEnvironment.isHeadless() returns true.

See Also:
`GraphicsEnvironment.isHeadless()`, `Component.setSize(int, int)`, `Component.setVisible(boolean)`, `JComponent.getDefaultLocale()`

JFrame

```
public JFrame(GraphicsConfiguration gc)
```

Creates a `Frame` in the specified `GraphicsConfiguration` of a screen device and a blank title.

This constructor sets the component's locale property to the value returned by `JComponent.getDefaultLocale`.

Parameters:
`gc` - the `GraphicsConfiguration` that is used to construct the new `Frame`, if `gc` is `null`, the system default

Fig. 2.10. Java 2 API documentation on JFrame

```
  Name: JFrame(String title)
  Description: Creates a new, initially invisible Frame with
the specified title.
```

The same result can also be achieved by explicitly setting the title as follows:

```
setTitle(APP_NAME + " " + APP_VERSION);
```

The line:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

uses the `setDefaultCloseOperation` method from the `JFrame` class which is defined as follows:

```
public void setDefaultCloseOperation(int operation)
```

and sets the operation that we want the application to perform by default when the user attempts to close the frame. We have specified the operation to exit the application by using the `EXIT_ON_CLOSE` option.

Coding the SwingBlast GUI

The general scheme for creating and adding Swing components to an application consists of the following steps:

1. Create an instance of a top-level container such as Jframe
2. Use a *layout manager* to specify the location and size of the components
3. Specify the top-level container's content pane to hold the individual GUI elements

To begin with, we create an instance of the JPanel class (called `newContentPane`), which defines a generic container as the top-level container. We will use this container to hold our GUI elements. Components are positioned inside a top-level container using what are known as *layout managers* in Java. The area within a top-level container where individual components (labels, buttons, etc.) are placed is called the *content pane.* To specify the *content pane* of the `newContentPane` component as the *content pane* for storing the visible elements of the

SwingBlast application, we use the top-level container's setContentPane method:

```
        newContentPane = new JPanel();
        newContentPane.setLayout(new BorderLayout());
          setContentPane(newContentPane);
```

Here we have used the *BorderLayout layout manager* to align and position the components. Next we add the menu bar (called SwingBlast) and a single menu item ("Quit"):

```
            JMenuBar menu = new JMenuBar();
            JMenu swingBlastMenu = new JMenu(APP_NAME);
            quitItem = new JMenuItem("Quit");
            swingBlastMenu.add(quitItem);
            menu.add(swingBlastMenu);
            setJMenuBar(menu);
```

Note that components are added using the add method as shown here for the SwingBlast menu:

```
            menu.add(swingBlastMenu);
```

Next we create the sequence pane and add a component called sequenceArea of the type *JTextArea* that simply defines an area for entering text:

```
            // The sequence pane
            JPanel sequencePanel = new JPanel();
            JLabel sequence = new JLabel("Sequence");
            sequenceArea = new JTextArea();
            sequenceArea.setLineWrap(true);
            scrollPaneArea = new JScrollPane(sequenceArea);
            sequencePanel.setLayout(new
BoxLayout(sequencePanel, BoxLayout.LINE_AXIS));
            sequencePanel.add(sequence);
            sequencePanel.add(Box.createRigidArea(new
Dimension(10, 0)));
            sequencePanel.add(scrollPaneArea);

sequencePanel.setBorder(BorderFactory.createEmptyBorder(10,
0, 10, 0));
```

To provide scrolling capabilities inside the text area (especially for large sequences), we have associated the *JScrollPane* object with the sequenceArea. The clear button is added in a similar fashion. Finally, we add the *main()* method to the program. The *main()* method actually performs the job of creating an instance of the class and running the application. The *Java Virtual Machine (JVM)* calls this `main()` method when we pass the class name to it. Every Java application must contain a *main()* method whose signature looks like this:

```
public static void main(String[] args) {
      // statements;
}
```

The JVM would eventually complain about a class if the `main()` method was missing. The simplified general format for a method in Java is:

```
method_modifier return_type method_name (arguments) {
    body of the method;
}
```

In our case, the method looks like this:

```
public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      final SwingBlast1_1 view = new SwingBlast1_1();
    }
  });
 }
}
```

The line:

```
SwingUtilities.invokeLater(new Runnable() { }
```

indicates that the painting of the GUI takes place in a separate *thread* (the *AWT thread* or *the event-dispatching thread*) and is a way of separating the GUI processes from the business processes (such as a BLAST operation) as strongly advised in the Java guidelines.

> A *thread* is a process that is capable of running concurrently alongside other *threads* or processes.
>
> The *event-dispatching thread* is the thread responsible for handling events and repainting of components. It is therefore very important to avoid any running heavy resource consuming code in the *event-dispatching thread.*

The keyword *Runnable* defines the type of object that will run in a new thread. The *invokeLater()* method causes the event-dispatching thread to call the `run()` method of the Runnable object which is passed to *invokeLater()* method after all pending events (such as repainting a component, etc.) are processed. The `run()` method of the Runnable object is in charge for creating a SwingBlast object through the constructor method of SwingBlast, which in turn performs all the specified actions, such as creating the top level window, setting its name and laying out the GUI elements, etc.

Compile and run the code shown in **Listing 2.1**. As you will notice, the basic framework as described above does not do anything useful apart from displaying the graphical interface as shown in **Fig. 2.8**. The only events the application can respond to so far are the default Minimize, Maximize and Close operations through icons located on the top right of the application window.

Coding the SwingBlast Business Logic

We will begin the process of building the business logic into the application by adding code that will format the user entered sequence into the commonly used Fasta format. We will simultaneously add code that will calculate and display the size of the input sequence. We will then incorporate a simple algorithm to determine the sequence type – that is, if the user entered sequence is nucleotide or protein.

The Fasta format as defined earlier contains a header that begins with the greater than symbol (>) and contains information about the sequence such as sequence identifiers and size, etc. (which may be delimited by separators such as vertical bars or spaces) on the first line and is followed on the second line with the actual sequence (**Fig. 2.6**).

So how do we get the sequence entered in the text area to rearrange itself in the Fasta format? As with any programming language there are more than one ways of achieving this. We will use a method based on *Focus events* to implement this. *Focus events* are triggered whenever a component such as text area gains or loses focus. *Focus events* associated with a particular component can be obtained by registering a *FocusListener* with the component. When the component gains or loses focus, the relevant method in the *listener* object (*focusGained* or *focusLost*, respectively) is invoked, and the *FocusEvent* is passed to it. The general method to do this is shown in **Listing 2.2**.

Listing 2.2. Adding Focus events and listeners to SwingBlast

```
sequenceArea.addFocusListener(new FocusListener() {
  public void focusGained(FocusEvent e) {

  }

  public void focusLost(FocusEvent e) {
    // add statements here
  }
});
```

We will design the code such that after a sequence has been added to the text area, it will be converted into the Fasta format as soon as the text area loses focus (for example, when a user navigates away from the text area to another part of the application). Conversely, no action will be performed when the `sequenceArea` component gains focus. We therefore want to add program logic in the *focusLost* method, which gets activated after a component loses focus, to achieve this. **Listing 2.3** shows how to implement this.

Listing 2.3. Programming the `focusLost` method

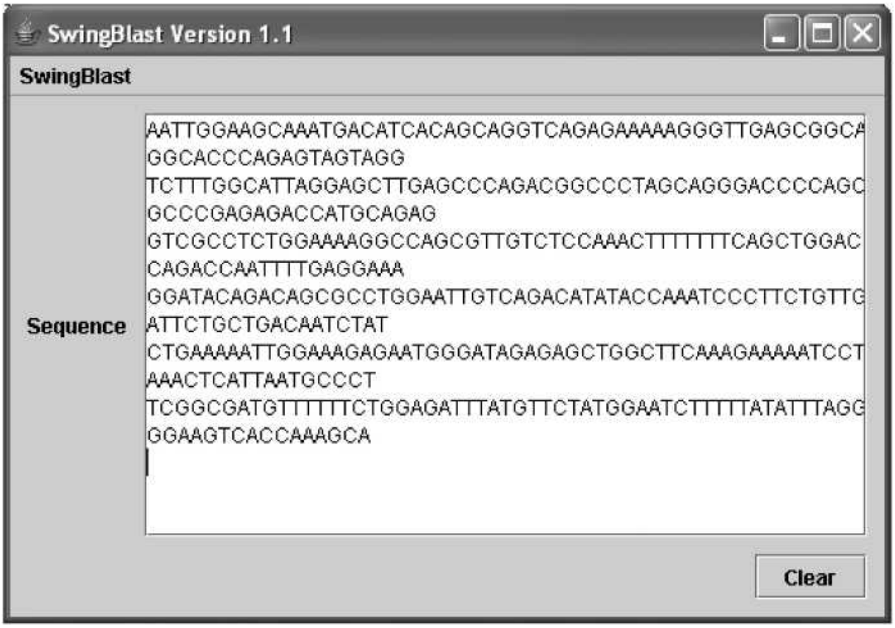
```
sequenceArea.addFocusListener(new FocusListener() {
  public void focusGained(FocusEvent e) {

  }

  public void focusLost(FocusEvent e) {
    // Retrieve the sequence in the text area
    String seqText = sequenceArea.getText();

    // Convert the sequence into Fasta format
    String header = null;
    int seqLength = 0;
    String sequence = "";
    String fastaSeq = "";

    seqText = seqText.replaceAll("\\s", "");
    sequence = seqText.toLowerCase();
    header = "> Sequence1";
    seqLength = seqText.length();
    fastaSeq = header + "|" + seqLength + "\n" +
sequence;
```

```
            sequenceArea.setText(fastaSeq);
          }

        });
```

For the header part of the Fasta sequence, we will add a generic label (called "sequence1") to represent the name of the raw sequence entered by the user followed by a vertical bar and the size of the sequence for the purpose of illustration. Plug this into the main code and test the application by pasting a sequence (such as the first few hundred bases of the CFTR gene sequence shown below) into it.

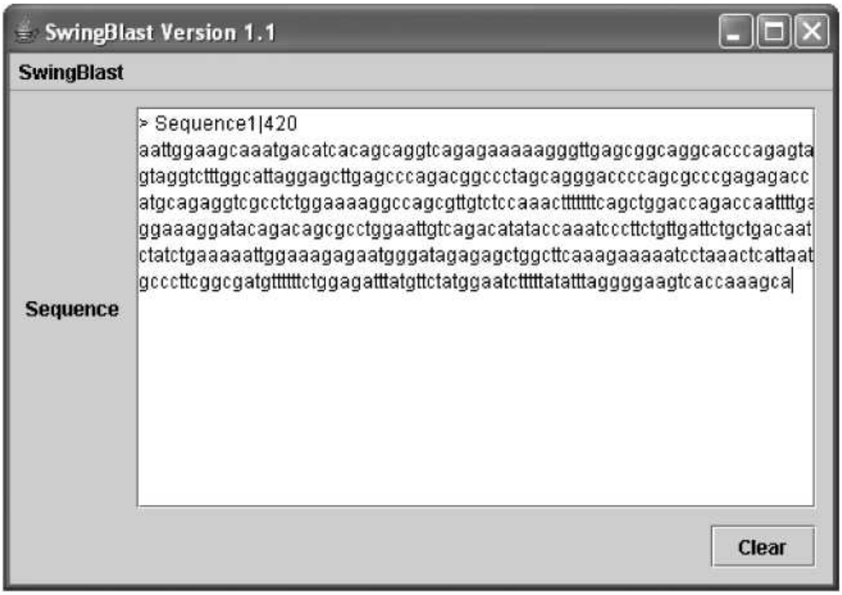
```
AATTGGAAGCAAATGACATCACAGCAGGTCAGAGAAAAAGGGTTGAGCGGCAGGCACCCAG
AGTAGTAGGTCTTTGGCATTAGGAGCTTGAGCCCAGACGGCCCTAGCAGGGACCCCAGCGC
CCGAGAGACCATGCAGAGGTCGCCTCTGGAAAAGGCCAGCGTTGTCTCCAAACTTTTTTTC
AGCTGGACCAGACCAATTTTGAGGAAAGGATACAGACAGCGCCTGGAATTGTCAGACATAT
ACCAAATCCCTTCTGTTGATTCTGCTGACAATCTATCTGAAAAATTGGAAAGAGAATGGGA
TAGAGAGCTGGCTTCAAAGAAAAATCCTAAACTCATTAATGCCCTTCGGCGATGTTTTTTC
TGGAGATTTATGTTCTATGGAATCTTTTTATATTTAGGGGAAGTCACCAAAGCA
```

You will see that once the text area loses focus, for example, by clicking on the `SwingBlast` menu, the sequence is converted into lower case and formatted into the Fasta format (**Fig. 2.11** and **Fig. 2.12**).



```
AATTGGAAGCAAATGACATCACAGCAGGTCAGAGAAAAAGGGTTGAGCGGCA
GGCACCCAGAGTAGTAGG
TCTTTGGCATTAGGAGCTTGAGCCCAGACGGCCCTAGCAGGGACCCCAGC
GCCCGAGAGACCATGCAGAG
GTCGCCTCTGGAAAAGGCCAGCGTTGTCTCCAAACTTTTTTTCAGCTGGAC
CAGACCAATTTTGAGGAAA
GGATACAGACAGCGCCTGGAATTGTCAGACATATACCAAATCCCTTCTGTTG
ATTCTGCTGACAATCTAT
CTGAAAAATTGGAAAGAGAATGGGATAGAGAGCTGGCTTCAAAGAAAAATCCT
AAACTCATTAATGCCCT
TCGGCGATGTTTTTTCTGGAGATTTATGTTCTATGGAATCTTTTTATATTTAGG
GGAAGTCACCAAAGCA
```

Fig. 2.11. Unformatted nucleotide sequence



```
> Sequence1|420
aattggaagcaaatgacatcacagcaggtcagagaaaaagggttgagcggcaggcacccagagta
gtaggtctttggcattaggagcttgagcccagacggccctagcagggaccccagcgcccgagagacc
atgcagaggtcgcctctggaaaaggccagcgttgtctccaaacttttttcagctggaccagaccaattttga
ggaaaggatacagacagcgcctggaattgtcagacatataccaaatcccttctgttgattctgctgacaat
ctatctgaaaaattggaaagagaatgggatagagagctggcttcaaagaaaaatcctaaactcattaat
gcccttcggcgatgttttttctggagatttatgttctatggaatctttttatatttaggggaagtcaccaaagca
```

Fig. 2.12. Fasta formatting of sequences (Text area loses focus)

In addition, a header line is added as specified in the code along with the length of the sequence. Although the logic to convert raw sequence into Fasta format does work as described, we need to incorporate a way to tell the *FocusEvent* method not to take any action if the sequence is already in the Fasta format (either because the sequence was pasted in the Fasta format or because it was formatted by the user formatted by the user using the FocusLost method) and therefore does not need formatting. This is easily done by checking for the presence of the ">" character at the beginning of the sequence as shown in **Listing 2.4** below.

Listing 2.4. Checking for Fasta formatting of sequences

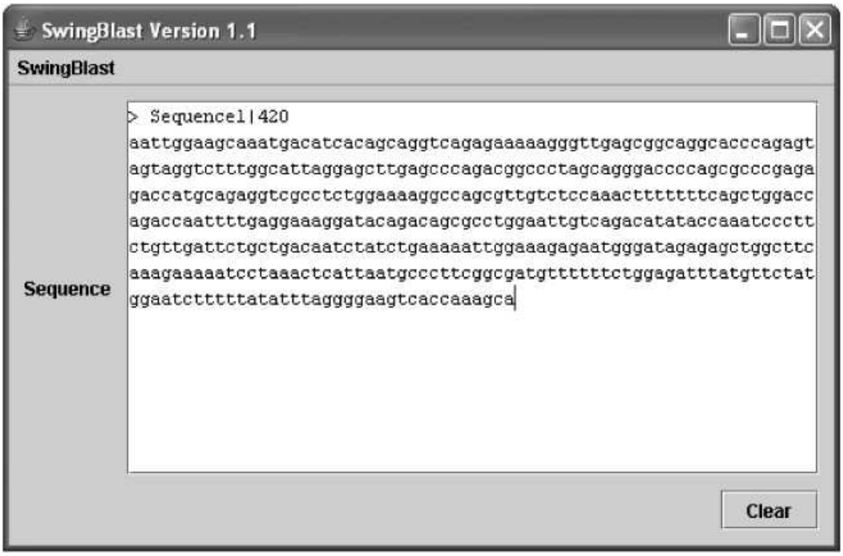
```
sequenceArea.addFocusListener(new FocusListener() {
  public void focusGained(FocusEvent e) {

 }

 public void focusLost(FocusEvent e) {
   // Retrieve the sequence in the text area
   String seqText = sequenceArea.getText();

   int idx = seqText.indexOf(">");
   boolean fastaFormatted = idx != -1;
```

```
          String header = null;
          int seqLength = 0;
          String sequence = "";
          String fastaSeq = "";
  // Check if sequence is in Fasta format
          if (fastaFormatted) {
            int returnIdx = seqText.indexOf("\n");
            header = seqText.substring(0, returnIdx);
            fastaSeq = seqText.substring(returnIdx + 1,
  seqText.length()).replaceAll("\\s", "").toLowerCase();
            fastaSeq = seqText;
          } else {
            seqText = seqText.replaceAll("\\s", "");
            fastaSeq = seqText.toLowerCase();
            header = "> Sequence1";
            seqLength = seqText.length();
          }

  // Convert the sequence into Fasta format if not Fasta
//formatted
          if (!fastaFormatted) {
            fastaSeq = header + "|" + seqLength + "\n" +
fastaSeq;

          }
          sequenceArea.setText(fastaSeq);
        }
```

To make the sequence align properly, we will use a monospace font such as Courier. The code to do this is as follows:

```
final Font sf = sequenceArea.getFont();
Font f = new Font("Monospaced", sf.getStyle(), sf.getSize());
sequenceArea.setFont(f);
```

Run the code again. This time the sequence is properly aligned (**Fig. 2.13**).



```
> Sequence1|420
aattggaagcaaatgacatcacagcaggtcagagaaaaagggttgagcggcaggcacccagagt
agtaggtctttggcattaggagcttgagcccagacggccctagcagggaccccagcgcccgaga
gaccatgcagaggtcgcctctggaaaaggccagcgttgtctccaaactttttttcagctggacc
agaccaattttgaggaaaggatacagacagcgcctggaattgtcagacatataccaaatccctt
ctgttgattctgctgacaatctatctgaaaaattggaaagagaatgggatagagagctggcttc
aaagaaaaatcctaaactcattaatgcccttcggcgatgtttttctggagatttatgttctat
ggaatcttttatatttaggggaagtcaccaaagca
```

Fig. 2.13. Using monospace font to format sequences

Determining Sequence Type: Nucleotide or Protein?

Now that we have formatted the sequence and calculated its size, lets plug in functionality into the SwingBlast application that will determine if the entered sequence is nucleotide (DNA or RNA) or protein. Note that RNA, like DNA is a polymer composed of four nucleotides. The difference between RNA and DNA is the nature of the sugar moiety: RNA has the ribose sugar, while DNA has the deoxyribose sugar. RNA has the same purine bases as DNA: adenine (A) and guanine (G) and the same pyrimidine cytosine (C), but instead of thymine (T), it uses the pyrimidine uracil (U).

Determination of sequence type is done with an algorithm that takes into account information on the natural composition of nucleotide and protein sequences. According to the algorithm, if:

1. Total number of nculeotides (that is, sum of A, T, G and C's) divided by the total length of the sequence is greater that 0.85, it is a DNA sequence

2. Total number of A, T, G, C and U's divided by the total length of the sequence is greater that 0.85, it is an RNA sequence

If neither of these two conditions is met, the sequence is assumed to be a protein sequence. Note that we are not using the extended DNA/RNA alphabet that includes symbols for sequence ambiguity as defined in the International Union of Pure and Applied Chemistry (IUPAC) and International Union of Biochemistry (IUB) nucleotide and amino acid nomenclature. Instead, we are assuming the DNA alphabet to be composed of the four bases A (adenine), T (thymine), G (guanine), C (cytosine) and N, the RNA alphabet to be composed of A (adenine), U (uridine), G (guanine), C (cytosine) and N (where N is any nucleotide base) and the amino acid alphabet to be composed of A (alanine), C (cysteine), D (aspartate), E (glutamic acid), F (phenylalanine), G (glycine), H (histidine), I (isoleucine), K (lysine), L (leucine), M (methionine), N (asparagine), P (proline), Q (glutamine), R (arginine), S (serine), T (threonine), V (valine), W (tryptophan) and Y (tyrosine).

Let's see how this algorithm works with an example. Take the partial mRNA sequence of the human CFTR gene (gi: 90421312) as shown below:

```
AAUUGGAAGCAAAUGACAUCACAGCAGGUCAGAGAAAAAGGGUUGAGCGGCAGGCACCCAG
AGUAGUAGGUCUUUGGCAUUAGGAGCUUGAGCCCAGACGGCCCUAGCAGGGACCCCAGCGC
CCGAGAGACCAUGCAGAGGUCGCCUCUGGAAAAGGCCAGCGUUGUCUCCAAACUUUUUUUC
AGCUGGACCAGACCAAUUUUGAGGAAAGGAUACAGACAGCGCCUGGAAUUGUCAGACAUAU
ACCAAAUCCCUUCUGUUGAUUCUGCUGACAAUCUAUCUGAAAAAUUGGAAAGAGAAUGGGA
UAGAGAGCUGGCUUCAAAGAAAAAUCCUAAACUCAUUAAUGCCCUUCGGCGAUGUUUUUC
UGGAGAUUUAUGUUCUAUGGAAUCUUUUUAUAUUUAGGGGAAGUCACCAAAGCAGUACAGC
CUCUCUUACUGGGAAGAAUCAUAGCUUCCUAUGACCCGGAUAACAAGGAGGAACGCUCUAU
CGCGAUUUAUCUAGGCAUAGGCUUAUGCCUUCUCUUUAUUGUGAGGACACUGCUCCUACAC
CCAGCCAUUUUUGGCCUUCAUCACAUUGGAAUGCAGAUGAGAAUAGCUAUGUUUAGUUUGA
UUUAUAAGAAGACUUUAAAGCUGUCAAGCCGUGUUCUAGAUAAAAUAAGUAUUGGACAACU
UGUUAGUCUCCUUUCCAACAACCUGAACAAAUUUGAUGAAGGACUUGCAUUGGCACAUUUC
GUGUGGAUCGCUCCUUUGCAAGUGGCACUCCUCAUGGGGCUAAUCUGGGAGUUGUUACAGG
CGUCUGCCUUCUGUGGACUUGGUUUCCUGAUAGUCCUUGCCCUUUUU
```

We will call this sequence with a size of 840 bases “S1”. Lets start by removing all A, T, G and C's from the sequence. The length of the sequence without A, T, G and C's is 237; lets call this sequence S2.

Number of A, T, G and C's in the sequence = S1 – S2 = 603. Next we remove all the U's from the sequence that remain after removing the A, T, G and C's (that is, the sequence S2). The length of the sequence after removing all the U's is zero (since all we had left were U's). Lets call this S3. The total number of U's in the sequence is therefore S2 – S3 is 237.

Now let's calculate the relative proportions of DNA and RNA alphabets in the sequence.

$$(A + T + G + C)/\text{Total} = 603/840 = 0.72$$

According to the algorithm, since this is less than 0.85, it cannot be a DNA sequence.

$$(A + T + G + C + U)/\text{Total} = (603 + 237)/840 = 1$$

Since this is > 0.85, this is an RNA sequence. We can now write the code using the above reasoning. Since we will use *regular expression* matching to parse the sequence, we will first import the appropriate libraries to do so:

```
import org.apache.regexp.RE;

import org.apache.regexp.RESyntaxException;
```

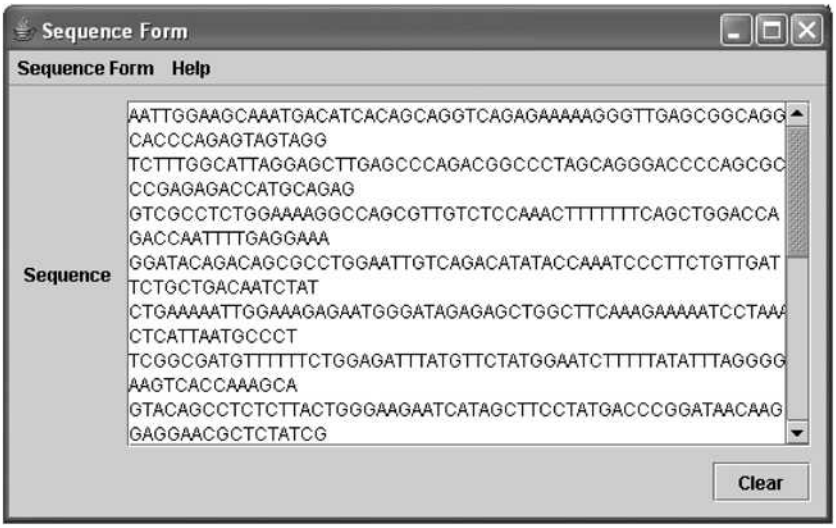
We declare the magic 0.85 number as a threshold:

```
private static final double SEQ_THRESHOLD = 0.85;
```

The `getSequenceType()` method that implements the algorithm is as follows:

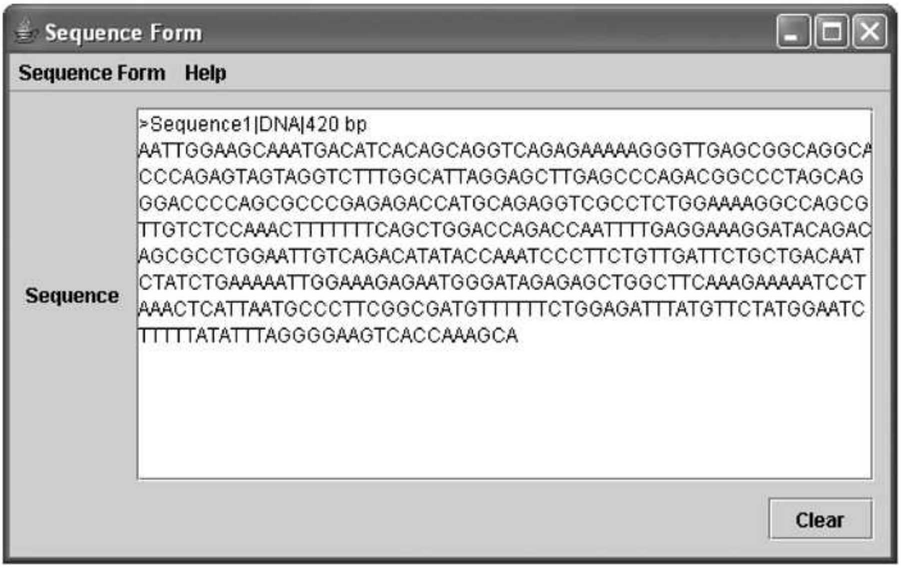
```
    public static int getSequenceType(String sequence) throws
RESyntaxException {
        RE re = new RE("[actgnACGTN]+");
        String[] strings = re.split(sequence);
        int numbOfLettersOtherThanATGCNs = 0;

        for (int i = 0; i < strings.length; i++) {
          numbOfLettersOtherThanATGCNs                          +=
strings[i].length();
        }
        int length = sequence.length();
        int        numbOfACGTNs        =        length         -
numbOfLettersOtherThanATGCNs;

        re = new RE("[uU]+");
        strings = re.split(sequence);
        int numbOfLettersOtherThanUs = 0;

        for (int i = 0; i < strings.length; i++) {
          numbOfLettersOtherThanUs += strings[i].length();
        }
        int numbOfUs = sequence.length() - 
numbOfLettersOtherThanUs;

        if (numbOfACGTNs / (double) length > SEQ_THRESHOLD) {
          return TYPE_DNA;
        } else if ((numbOfACGTNs + numbOfUs) / (double) 
length > SEQ_THRESHOLD) {
          return TYPE_RNA;
        } else {
          return TYPE_PROTEIN;
        }
      }
```

With this code in place, we get the following results for the partial sequences of the human CFTR nucleotide (**Fig. 2.14** and **Fig. 2.15**) and protein (**Fig. 2.16** and **Fig. 2.17**).



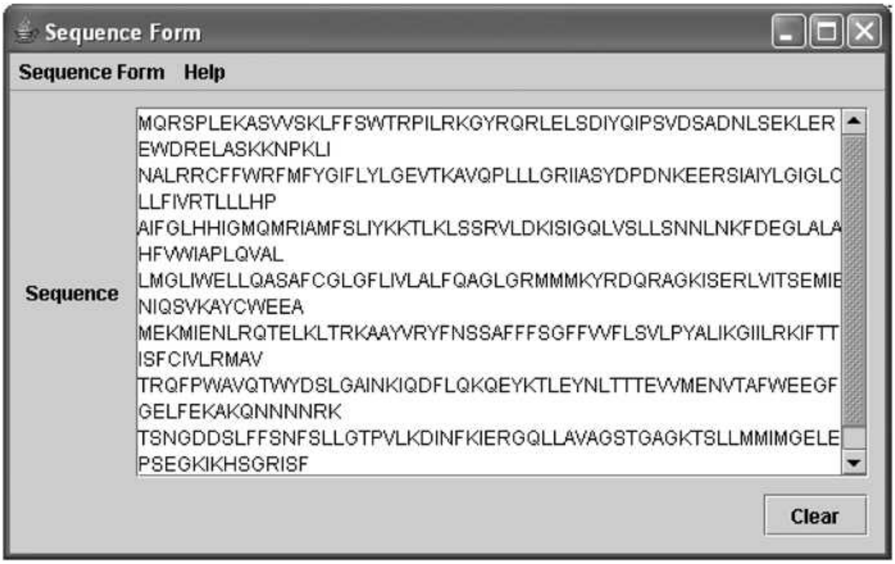
```
AATTGGAAGCAAATGACATCACAGCAGGTCAGAGAAAAAGGGTTGAGCGGCAGG
CACCCAGAGTAGTAGG
TCTTTGGCATTAGGAGCTTGAGCCCAGACGGCCCTAGCAGGGACCCCAGCGC
CCGAGAGACCATGCAGAG
GTCGCCTCTGGAAAAGGCCAGCGTTGTCTCCAAACTTTTTTTCAGCTGGACCA
GACCAATTTTGAGGAAA
GGATACAGACAGCGCCTGGAATTGTCAGACATATACCAAATCCCTTCTGTTGAT
TCTGCTGACAATCTAT
CTGAAAAATTGGAAAGAGAATGGGATAGAGAGCTGGCTTCAAAGAAAAATCCTAAA
CTCATTAATGCCCT
TCGGCGATGTTTTTTCTGGAGATTTATGTTCTATGGAATCTTTTTATATTTAGGGG
AAGTCACCAAAGCA
GTACAGCCTCTCTTACTGGGAAGAATCATAGCTTCCTATGACCCGGATAACAAG
GAGGAACGCTCTATCG
```

Fig. 2.14. Determining sequence type - CFTR nucleotide sequence



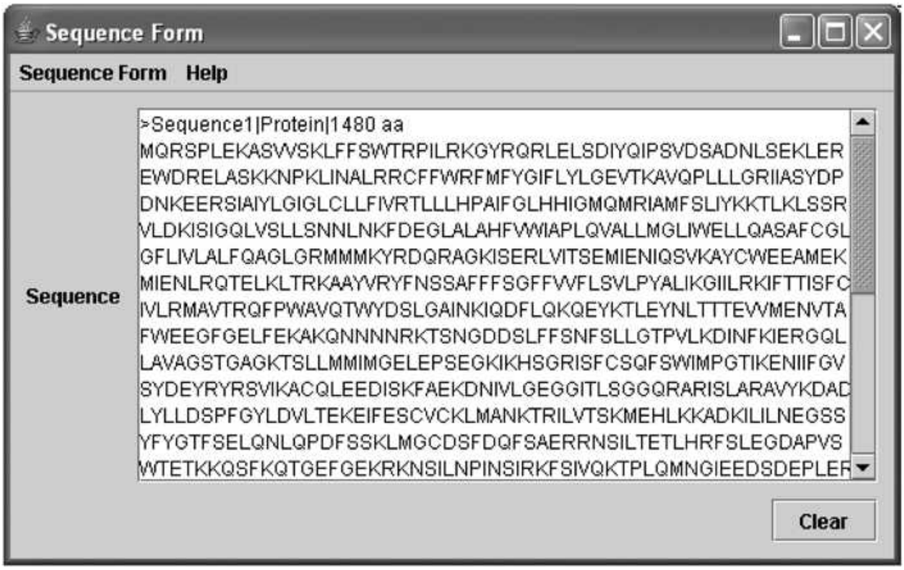
```
>Sequence1|DNA|420 bp
AATTGGAAGCAAATGACATCACAGCAGGTCAGAGAAAAAGGGTTGAGCGGCAGGCA
CCCAGAGTAGTAGGTCTTTGGCATTAGGAGCTTGAGCCCAGACGGCCCTAGCAG
GGACCCCAGCGCCCGAGAGACCATGCAGAGGTCGCCTCTGGAAAAGGCCAGCG
TTGTCTCCAAACTTTTTTTCAGCTGGACCAGACCAATTTTGAGGAAAGGATACAGAC
AGCGCCTGGAATTGTCAGACATATACCAAATCCCTTCTGTTGATTCTGCTGACAAT
CTATCTGAAAAATTGGAAAGAGAATGGGATAGAGAGCTGGCTTCAAAGAAAAATCCT
AAACTCATTAATGCCCTTCGGCGATGTTTTTTCTGGAGATTTATGTTCTATGGAATC
TTTTTATATTTAGGGGAAGTCACCAAAGCA
```

Fig. 2.15. Determining sequence type - CFTR nucleotide sequence



```
MQRSPLEKASVVSKLFFSWTRPILRKGYRQRLELSDIYQIPSVDSADNLSEKLER
EWDRELASKKNPKLI
NALRRCFFWRFMFYGIFLYLGEVTKAVQPLLLGRIIASYDPDNKEERSIAIYLGIGLC
LLFIVRTLLLHP
AIFGLHHIGMQMRIAMFSLIYKKTLKLSSRVLDKISIGQLVSLLSNNLNKFDEGLALA
HFVWIAPLQVAL
LMGLIWELLQASAFCGLGFLIVLALFQAGLGRMMMKYRDQRAGKISERLVITSEMIE
NIQSVKAYCWEEA
MEKMIENLRQTELKLTRKAAYVRYFNSSAFFFSGFFVVFLSVLPYALIKGIILRKIFTT
ISFCIVLRMAV
TRQFPWAVQTWYDSLGAINKIQDFLQKQEYKTLEYNLTTTEVVMENVTAFWEEGF
GELFEKAKQNNNNRK
TSNGDDSLFFSNFSLLGTPVLKDINFKIERGQLLAVAGSTGAGKTSLLMMIMGELE
PSEGKIKHSGRISF
```

Fig. 2.16. Determining sequence type - CFTR protein sequence



```
>Sequence1|Protein|1480 aa
MQRSPLEKASVVSKLFFSWTRPILRKGYRQRLELSDIYQIPSVDSADNLSEKLER
EWDRELASKKNPKLINALRRCFFWRFMFYGIFLYLGEVTKAVQPLLLGRIIASYDP
DNKEERSIAIYLGIGLCLLFIVRTLLLHPAIFGLHHIGMQMRIAMFSLIYKKTLKLSSR
VLDKISIGQLVSLLSNNLNKFDEGLALAHFVWIAPLQVALLMGLIWELLQASAFCGL
GFLIVLALFQAGLGRMMMKYRDQRAGKISERLVITSEMIENIQSVKAYCWEEAMEK
MIENLRQTELKLTRKAAYVRYFNSSAFFFSGFFVVFLSVLPYALIKGIILRKIFTTISFC
IVLRMAVTRQFPWAVQTWYDSLGAINKIQDFLQKQEYKTLEYNLTTTEVVMENVTA
FWEEGFGELFEKAKQNNNNRKTSNGDDSLFFSNFSLLGTPVLKDINFKIERGQL
LAVAGSTGAGKTSLLMMIMGELEPSEGKIKHSGRISFCSQFSWIMPGTIKENIIFGV
SYDEYRYRSVIKACQLEEDISKFAEKDNIVLGEGGITLSGGQRARISLARAVYKDAD
LYLLDSPFGYLDVLTEKEIFESCVCKLMANKTRILVTSKMEHLKKADKILILNEGSS
YFYGTFSELQNLQPDFSSKLMGCDSFDQFSAERRNSILTETLHRFSLEGDAPVS
WTETKKQSFKQTGEFGEKRKNSILNPINSIRKFSIVQKTPLQMNGIEEDSDEPLER
```

Fig. 2.17. Determining sequence type: CFTR protein sequence

We will call this `SwingBlast` version 1.2. The complete code is described in **Listing 2.5**.

Listing 2.5. Determining sequence type

```
package org.jfb.SwingBlast;

import org.apache.regexp.RE;
import org.apache.regexp.RESyntaxException;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.FocusEvent;
import java.awt.event.FocusListener;

public class SwingBlastl_2 extends JFrame {
  private static final String APP_NAME = "Sequence Form";
  private static final String APP_VERSION = "Version 1_2";

  private static final Dimension APP_WINDOW_SIZE = new
Dimension(450, 350);

  private static final int TYPE_DNA = 0;
  private static final int TYPE_RNA = 1;
  private static final int TYPE_PROTEIN = 2;

  private JComponent newContentPane;
```

```
    private JTextArea sequenceArea;
    private JScrollPane scrollPaneArea;
    private JButton clear;

    private JMenuItem aboutItem;
    private JMenuItem quitItem;
    private static final double SEQ_THRESHOLD = 0.85;

    public SwingBlast1_2() {
      super();
      seqFormInit();
    }

    private void seqFormInit() {
      setTitle(APP_NAME);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      newContentPane = new JPanel();
      newContentPane.setOpaque(true);
      newContentPane.setLayout(new BorderLayout());

      setContentPane(newContentPane);

      // Create the menu bar
      JMenuBar menu = new JMenuBar();
      JMenu swingBlastMenu = new JMenu(APP_NAME);
      quitItem = new JMenuItem("Quit");
      swingBlastMenu.add(quitItem);
      menu.add(swingBlastMenu);

      JMenu helpMenu = new JMenu("Help");
      aboutItem = new JMenuItem("About");
      helpMenu.add(aboutItem);
      menu.add(helpMenu);
      setJMenuBar(menu);

      // Create the sequence pane
      JPanel sequencePanel = new JPanel();
      JLabel sequence = new JLabel("Sequence");
      sequenceArea = new JTextArea();
      Font font = sequence.getFont();
      sequenceArea.setFont(new  Font("Courier",  Font.PLAIN,
font.getSize()));
      sequenceArea.setLineWrap(true);
      scrollPaneArea = new JScrollPane(sequenceArea);

      sequencePanel.setLayout(new   BoxLayout(sequencePanel,
BoxLayout.LINE_AXIS));
      sequencePanel.add(sequence);
      sequencePanel.add(Box.createRigidArea(new Dimension(10,
0)));
      sequencePanel.add(scrollPaneArea);

sequencePanel.setBorder(BorderFactory.createEmptyBorder(10,
0, 10, 0));

      // Lay out the buttons from left to right
      JPanel buttonPane = new JPanel();
      clear = new JButton("Clear");

      buttonPane.setLayout(new          BoxLayout(buttonPane,
BoxLayout.LINE_AXIS));
      buttonPane.add(Box.createHorizontalGlue());
      buttonPane.add(Box.createRigidArea(new    Dimension(10,
0)));
      buttonPane.add(clear);

      JPanel jPanel = new JPanel();
      jPanel.setLayout(new BorderLayout());
      jPanel.setBorder(BorderFactory.createEmptyBorder(0, 10,
10, 10));
      jPanel.add(sequencePanel, BorderLayout.CENTER);
      jPanel.add(buttonPane, BorderLayout.SOUTH);

      newContentPane.add(jPanel, BorderLayout.CENTER);
      newContentPane.setPreferredSize(APP_WINDOW_SIZE);

      // Display the window
      pack();
      Dimension                  screenSize                  =
Toolkit.getDefaultToolkit().getScreenSize();
      setLocation((screenSize.width  -  APP_WINDOW_SIZE.width)
/ 2,
        (screenSize.height - APP_WINDOW_SIZE.height) / 2);
      setVisible(true);

      addListeners();
    }

    private void addListeners() {
      quitItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          System.exit(0);
        }
      });

      aboutItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          JOptionPane.showMessageDialog(SwingBlast1_2.this,
APP_NAME + " " + APP_VERSION,
            "About          "          +          APP_NAME,
JOptionPane.INFORMATION_MESSAGE);
        }
      });

      clear.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          sequenceArea.setText("");
        }
      });

      sequenceArea.addFocusListener(new FocusListener() {
        public void focusGained(FocusEvent e) {

        }

        public void focusLost(FocusEvent e) {
          // Check if the sequence is DNA, RNA or protein

          String text = sequenceArea.getText();

  // Format the sequence in FASTA format and retrieve the
  // sequence the user entered
          int idx = text.indexOf(">");
          boolean fastaFormatted = idx != -1;
          String seqText = null;
          String header = null;
          int seqLength = 0;
          String sequence = "";

          if (fastaFormatted) {
            int returnIdx = text.indexOf("\n");
            header = text.substring(0, returnIdx);
            sequence    =    text.substring(returnIdx    +    1,
text.length()).replaceAll("\\s", "").toLowerCase();
            seqText = text;
          } else {
            text = text.replaceAll("\\s", "");
            sequence = text.toLowerCase();
            header = ">Sequence1|";
            seqLength = text.length();
          }

          // Determine the sequence type
          int typeOfSequence = -1;
          try {
            typeOfSequence = getSequenceType(sequence);
          } catch (RESyntaxException e1) {
            e1.printStackTrace();
          }

          String type = null;
          String unitOfLength = null;

          switch (typeOfSequence) {
            case TYPE_DNA:
              type = "DNA";
              unitOfLength = " bp";
              break;
            case TYPE_RNA:
```

```
       type = "RNA";
       unitOfLength = " bp";
       break;
      case TYPE_PROTEIN:
       type = "Protein";
       unitOfLength = " aa";
       break;
      default:
       type = "N/A";
       unitOfLength = " N/A";
     }

     if (!fastaFormatted) {
      seqText  =  header  +  type  +  "|"  +  seqLength  +
unitOfLength + "\n" + sequence.toUpperCase();
     }

     // Display the results in sequence text area
     sequenceArea.setText(seqText);
    }
   });
  }

  public static int getSequenceType(String sequence) throws
RESyntaxException {
    RE re = new RE("[actgnACGTN]+");
    String[] strings = re.split(sequence);
    int numbOfLettersOtherThanATGCNs = 0;

    for (int i = 0; i < strings.length; i++) {
     numbOfLettersOtherThanATGCNs                    +=
strings[i].length();
    }
    int length = sequence.length();
    int       numbOfACGTNs       =       length       -
numbOfLettersOtherThanATGCNs;

    re = new RE("[uU]+");
    strings = re.split(sequence);
    int numbOfLettersOtherThanUs = 0;

    for (int i = 0; i < strings.length; i++) {
     numbOfLettersOtherThanUs += strings[i].length();
    }
    int      numbOfUs      =      sequence.length()      -
numbOfLettersOtherThanUs;

    if (numbOfACGTNs / (double) length > SEQ_THRESHOLD) {
     return TYPE_DNA;
    }  else  if  ((numbOfACGTNs  +  numbOfUs)  /  (double)
length > SEQ_THRESHOLD) {
     return TYPE_RNA;
    } else {
```

```
                return TYPE_PROTEIN;
            }
        }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                final SwingBlast1_2 view = new SwingBlast1_2();
            }
        });
    }
}
```

Note how we have handled the creation of the GUI elements in SwingBlast version 1.2 (**Listing 2.5**):

```
    public SwingBlast1_2() {
        super();
        seqFormInit();
    }
```

We first created a method called `seqFormInit()` containing all the code to layout the components and then called the method in the code shown above. Earlier, for SwingBlast Version 1.1, we had instead bundled all the code within the main class (**Listing 2.1**):

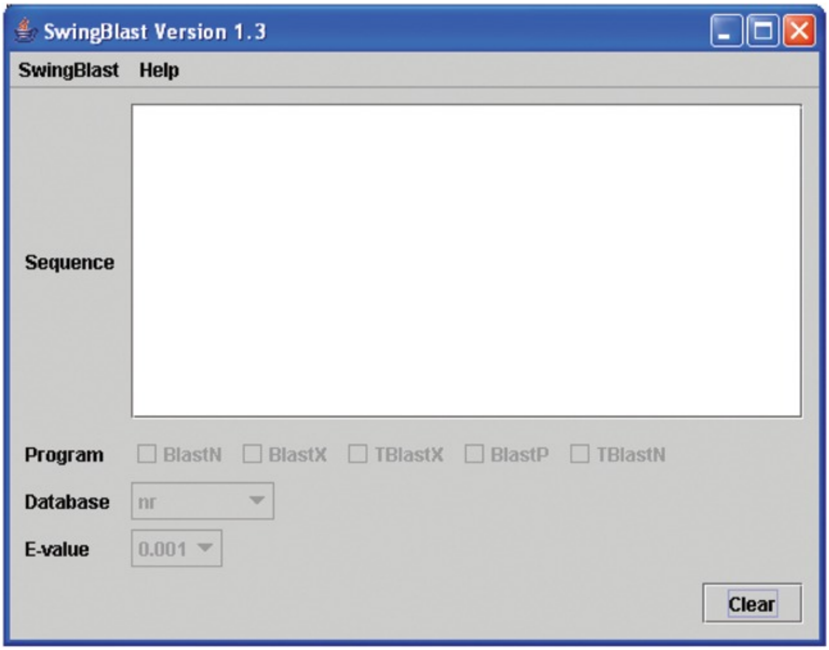
```
public SwingBlast1_1() {

    setTitle(APP_NAME + " " + APP_VERSION);

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    ...
}
```

Using a separate method to build the GUI makes the code easier to read by separating the widget part from the implementation aspect.

Displaying Valid BLAST Options

The next step, now that we have accurately determined the type of sequence the user has entered in the text area, is determine which BLAST options to display for the particular type of input sequence. The purpose of this is to enable the application to automatically present only the valid BLAST algorithms appropriate for the input sequence provided by the user. Currently, if a user selects Nucleotide-nucleotide BLAST (BLASTN) on the NCBI BLAST server and supplies a protein sequence or a GenBank Id corresponding to a protein sequence, an error message pointing the mismatch is displayed; however, the BLAST server does not automatically present the valid options based on user input. Recall from **Table 2.1** that the valid BLAST options for nucleotide sequences are BLASTN, BLASTX and TBLASTX and the valid options for amino acid sequences are BLASTP and TBLASTN.

We will begin by adding the needed GUI elements to the `SwingBlast` application. The GUI elements we will need are five checkboxes for the five BLAST algorithms (BLASTN, BLASTP, BLASTX, TBLASTN and TBLASTX), a drop-down menu to select the databases to search the input sequence against and the E-value to specify the stringency of search. The application at this stage should appear as shown in **Fig. 2.18**. We will program these GUI elements to be inactivated upon launch of the application since no sequence is available for analysis. We will call this version 1.3 of the SwingBlast application.



Sequence

Program □ BlastN □ BlastX □ TBlastX □ BlastP □ TBlastN

Database nr

E-value 0.001

Fig. 2.18. Adding BLAST options to SwingBlast

The code to add the BLAST programs as check boxes is as follows. We first create the required array variables: BLAST_PROGRAMS_DNA, BLAST_PROGRAMS_PROTEIN, DATABASES and EVALUES to hold the appropriate allowed values for each of the parameters. Note that we are illustrating this application with a few BLAST parameters. The user can add more parameters as per individual requirements.

```
    private static final String[] BLAST_PROGRAMS_DNA = new
String[]{"BlastN", "BlastX", "TBlastX"};
    private static final String[] BLAST_PROGRAMS_PROTEIN =
new String[]{"BlastP", "TBlastN"};
    private static final String[] DATABASES = new
String[]{"nr", "est_human"};
    private static final String[] EVALUES = new
String[]{"0.001", "0.01", "0.1", "1", "10", "100"};
```

We then create the necessary widgets: check boxes for the DNA and protein BLAST options and combo boxes for the database and E-values.

```
    private JCheckBox[] cbDna;
    private JCheckBox[] cbProtein;
    private JComboBox comboDbs;
    private JComboBox comboEvalues;
```

We create a method called createProgramPanel() that draws the BLAST program panel, the database panel and the E-value panel (**Listing 2.6**).

Listing 2.6. Laying out the BLAST widgets

```
    private JPanel createProgramPanel() {
      // Create the program panel
      JPanel programPanel = new JPanel();
      JLabel program = new JLabel("Program");
      program.setPreferredSize(LABEL_PREFERRED_SIZE);
      cbDna = new JCheckBox[BLAST_PROGRAMS_DNA.length];
      String blastProgram;
      for (int i = 0; i < BLAST_PROGRAMS_DNA.length; i++) {
        blastProgram = BLAST_PROGRAMS_DNA[i];
        cbDna[i] = new JCheckBox(blastProgram);
        cbDna[i].setMaximumSize(COMBO_PREFERRED_SIZE);
      }
      cbProtein = new
JCheckBox[BLAST_PROGRAMS_PROTEIN.length];
      for (int i = 0; i < BLAST_PROGRAMS_PROTEIN.length; i++)
{
        blastProgram = BLAST_PROGRAMS_PROTEIN[i];
        cbProtein[i] = new JCheckBox(blastProgram);
        cbProtein[i].setMaximumSize(COMBO_PREFERRED_SIZE);
      }

      programPanel.setLayout(new BoxLayout(programPanel,
BoxLayout.LINE_AXIS));
      programPanel.add(program);
      programPanel.add(Box.createRigidArea(new Dimension(10,
0)));
      for (int i = 0; i < cbDna.length; i++) {
        programPanel.add(cbDna[i]);
        programPanel.add(Box.createRigidArea(new Dimension(5,
0)));
      }
      for (int i = 0; i < cbProtein.length; i++) {
        programPanel.add(cbProtein[i]);
        if (i + 1 < cbProtein.length)
          programPanel.add(Box.createRigidArea(new
Dimension(5, 0)));
      }
      programPanel.add(Box.createHorizontalGlue());
      JPanel paramPanel = new JPanel();
      paramPanel.setLayout(new BoxLayout(paramPanel,
BoxLayout.PAGE_AXIS));

      paramPanel.add(programPanel);
      paramPanel.add(Box.createRigidArea(new Dimension(0,
5)));

      // Create the database panel
      JPanel databasePanel = new JPanel();
      JLabel database = new JLabel("Database");
      database.setPreferredSize(LABEL_PREFERRED_SIZE);
      comboDbs = new JComboBox(DATABASES);
      comboDbs.setMaximumSize(COMBO_PREFERRED_SIZE);

      databasePanel.setLayout(new BoxLayout(databasePanel,
BoxLayout.LINE_AXIS));
      databasePanel.add(database);
      databasePanel.add(Box.createRigidArea(new Dimension(10,
0)));
      databasePanel.add(comboDbs);
      databasePanel.add(Box.createHorizontalGlue());
      paramPanel.add(databasePanel);
      paramPanel.add(Box.createRigidArea(new Dimension(0,
5)));

      // Create the E-Value panel
      JPanel evaluePanel = new JPanel();
      JLabel eValue = new JLabel("E-value");
      eValue.setPreferredSize(LABEL_PREFERRED_SIZE);
      comboEvalues = new JComboBox(EVALUES);
      comboEvalues.setMaximumSize(COMBO_PREFERRED_SIZE);
      evaluePanel.setLayout(new BoxLayout(evaluePanel,
BoxLayout.LINE_AXIS));
      evaluePanel.add(eValue);
      evaluePanel.add(Box.createRigidArea(new Dimension(10,
0)));
      evaluePanel.add(comboEvalues);
      evaluePanel.add(Box.createHorizontalGlue());
      paramPanel.add(evaluePanel);
      paramPanel.add(Box.createRigidArea(new Dimension(0,
5)));

      enableFunctions(TYPE_UNKNOWN);
      return paramPanel;
    }
```

The `enableFunctions()` method takes an `int` parameter (`typeOfSequence`) and is responsible for setting the check boxes for the BLAST programs to enable or disable them based on the type of sequence entered by the user. We will use the `setEnabled()` function to enable (or disable) a button. The `setEnabled()` method takes a parameter of type Boolean which can be set to `true` to enable the button and `false` to disable the button.

In case of a nucleotide sequence, we want the three check boxes for BLASTN, BLASTX and TBLASTX to be available. Simultaneously, we want the database and the E-value combo boxes to become enabled as soon as the user enters a sequence. This logic is implemented in the following manner:

```
    private void enableFunctions(int typeOfSequence) {
      if (typeOfSequence == TYPE_DNA || typeOfSequence ==
TYPE_RNA) {
        setChb(chbDna, true);
        setChb(chbProtein, false);
        setCob(cobDbs, true);
        setCob(cobEvalues, true);
      } else if (typeOfSequence == TYPE_PROTEIN) {
        setChb(chbProtein, true);
        setChb(chbDna, false);
        setCob(cobDbs, true);
        setCob(cobEvalues, true);
      } else {
        setChb(chbProtein, false);
        setChb(chbDna, false);
        setCob(cobDbs, false);
        setCob(cobEvalues, false);
      }
    }
```

In the code shown above, we define the `setChb()` and `setCob()` methods to change the settings of the check boxes (chbProtein for protein searches, chbDNA for nucleotide searches) and the combo boxes (cobDbs for database type and cobEvalues for E-values) respectively. These methods take the object type as the first parameter (check or combo box whose state needs to be set) and a `Boolean` parameter (true/false) as illustrated below:

```
    private static void setChb(JCheckBox[] boxes, boolean
value) {
      for (int i = 0; i < boxes.length; i++) {
        boxes[i].setEnabled(value);
        boxes[i].setSelected(false);
      }
    }
```

In the above method, we iterate over the check boxes, set them to enabled or disabled and ensure that they are not selected by default. For example, when the following method is called:

```
        setChb(cbDna, true);
```

the method changes only the DNA check boxes to true (enables them) since we have set cbDNA to hold the array of check boxes for only the two nucleotide related BLAST programs in the code:

```
    private static final String[] BLAST_PROGRAMS_DNA = new
String[]{"BlastN", "BlastX", "TBlastX"};

  cbDna = new JCheckBox[BLAST_PROGRAMS_DNA.length];
```

Similarly, the `setCob()` function sets the values for the combo boxes for the database and the E-values:

```
    private static void setCob(JComboBox component, boolean
value) {
      component.setEnabled(value);
      component.setSelectedIndex(0);
    }
```

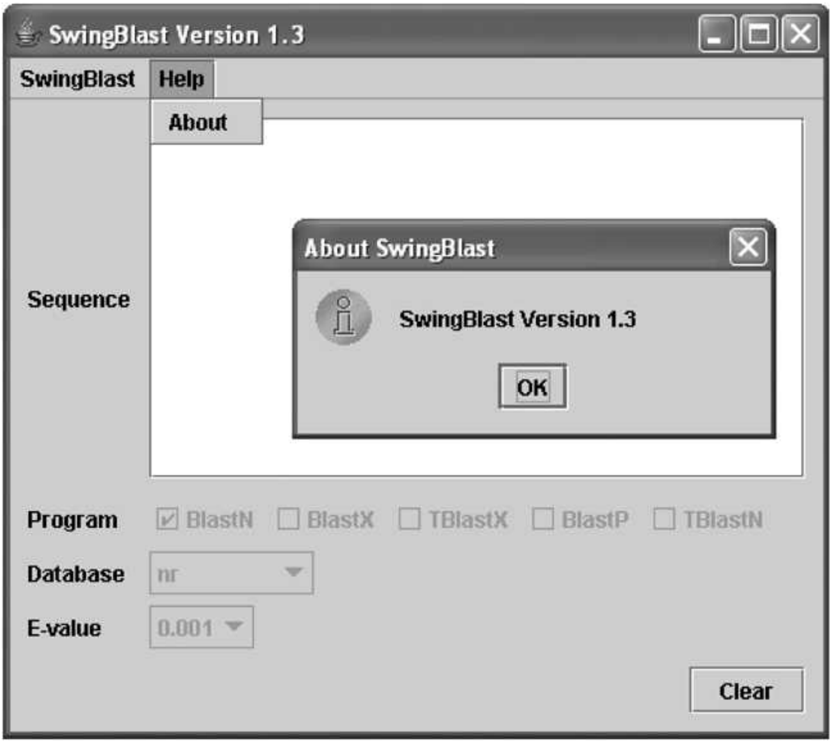
Conversely, for a protein sequence, we want the BLASTP and TBLASTN check boxes and the database and the E-value combo boxes to become enabled and the check boxes for BLASTN, BLASTX and TBLASTX disabled. The method with this logic included is as follows:

```
    private void enableFunctions(int typeOfSequence) {
      if  (typeOfSequence  ==  TYPE_DNA  ||  typeOfSequence  ==
TYPE_RNA) {
        setChb(cbDna, true);
        setChb(cbProtein, false);
        setCob(comboDbs, true);
        setCob(comboEvalues, true);
      } else if (typeOfSequence == TYPE_PROTEIN) {
        setChb(cbProtein, true);
        setChb(cbDna, false);
        setCob(comboDbs, true);
        setCob(comboEvalues, true);
      } else {
        setChb(cbProtein, false);
        setChb(cbDna, false);
        setCob(comboDbs, false);
        setCob(comboEvalues, false);
      }
    }
```

We will also add a `Help` menu item. The code to add that is fairly simple:

```
      JMenu helpMenu = new JMenu("Help");
      aboutItem = new JMenuItem("About");
      helpMenu.add(aboutItem);
      menu.add(helpMenu);
```

The `Help` → `About` simply describes the current `SwingBlast` version (**Fig. 2.19**). The complete code for the application is described in **Listing 2.7**.



SwingBlast Version 1.3

Program ☑ BlastN □ BlastX □ TBlastX □ BlastP □ TBlastN

Database nr

E-value 0.001

Fig. 2.19. Help About Menu information

Listing 2.7. SwingBlast version 1.3

```
  package org.jfb.SwingBlast;

  import org.apache.regexp.RE;
  import org.apache.regexp.RESyntaxException;

  import javax.swing.*;
  import java.awt.*;
  import java.awt.event.ActionEvent;
  import java.awt.event.ActionListener;
  import java.awt.event.FocusEvent;
  import java.awt.event.FocusListener;

  public class SwingBlast1_3 extends JFrame {
    private static final String APP_NAME = "SwingBlast";
    private static final String APP_VERSION = "Version 1.3";

    private static final Dimension LABEL_PREFERRED_SIZE = new
Dimension(57, 16);
```

```
    private static final Dimension COMBO_PREFERRED_SIZE = new
Dimension(60, 25);
    private static final Dimension CP_PREF_SIZE = new
Dimension(450, 350);

    private static final int TYPE_DNA = 0;
    private static final int TYPE_RNA = 1;
    private static final int TYPE_PROTEIN = 2;

    private static final String[] BLAST_PROGRAMS_DNA = new
String[]{"BlastN", "BlastX", "TBlastX"};
    private static final String[] BLAST_PROGRAMS_PROTEIN =
new String[]{"BlastP", "TBlastN"};
    private static final String[] DATABASES = new
String[]{"nr", "est_human"};
    private static final String[] EVALUES = new
String[]{"0.001", "0.01", "0.1", "1", "10", "100"};

    private JComponent newContentPane;
    private JTextArea sequenceArea;
    private JScrollPane scrollPaneArea;

    private JCheckBox[] chbDna;
    private JCheckBox[] chbProtein;
    private JComboBox cobDbs;
    private JComboBox cobEvalues;

    private JButton clear;

    private JMenuItem aboutItem;
    private JMenuItem quitItem;
    private static final double SEQ_THRESHOLD = 0.85;
    private static final int TYPE_UNKNOWN = -1;

    public SwingBlast1_3() {
      super();
      seqFormInit();
    }

    private void seqFormInit() {
      setTitle(APP_NAME + " " + APP_VERSION);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      newContentPane = new JPanel();
      newContentPane.setOpaque(true);
      newContentPane.setLayout(new BorderLayout());

      setContentPane(newContentPane);

      // Create the menu bar
      JMenuBar menu = new JMenuBar();
      JMenu swingBlastMenu = new JMenu(APP_NAME);
      quitItem = new JMenuItem("Quit");
      swingBlastMenu.add(quitItem);
```

```
        menu.add(swingBlastMenu);

        JMenu helpMenu = new JMenu("Help");
        aboutItem = new JMenuItem("About");
        helpMenu.add(aboutItem);
        menu.add(helpMenu);
        setJMenuBar(menu);

        // Create the sequence pane
        JPanel sequencePanel = new JPanel();
        JLabel sequence = new JLabel("Sequence");
        sequenceArea = new JTextArea();
        sequenceArea.setLineWrap(true);
        scrollPaneArea = new JScrollPane(sequenceArea);
        scrollPaneArea.setPreferredSize(new Dimension(300,
200));

        sequencePanel.setLayout(new BoxLayout(sequencePanel,
BoxLayout.LINE_AXIS));
        sequencePanel.add(sequence);
        sequencePanel.add(Box.createRigidArea(new Dimension(10,
0)));
        sequencePanel.add(scrollPaneArea);

sequencePanel.setBorder(BorderFactory.createEmptyBorder(10,
0, 10, 0));

        // Lay out the buttons from left to right
        JPanel buttonPane = new JPanel();
        clear = new JButton("Clear");

        buttonPane.setLayout(new BoxLayout(buttonPane,
BoxLayout.LINE_AXIS));
        buttonPane.add(Box.createHorizontalGlue());
        buttonPane.add(Box.createRigidArea(new Dimension(10,
0)));
        buttonPane.add(clear);

        JPanel jPanel = new JPanel();
        jPanel.setLayout(new BorderLayout());
        jPanel.setBorder(BorderFactory.createEmptyBorder(0, 10,
10, 10));
        jPanel.add(sequencePanel, BorderLayout.NORTH);
        jPanel.add(createProgramPanel(), BorderLayout.CENTER);
        jPanel.add(buttonPane, BorderLayout.SOUTH);

        newContentPane.add(jPanel, BorderLayout.CENTER);
        newContentPane.setPreferredSize(CP_PREF_SIZE);

        // Display the window
        pack();
        Dimension screenSize =
Toolkit.getDefaultToolkit().getScreenSize();
```

```
      setLocation((screenSize.width - CP_PREF_SIZE.width) /
2,
        (screenSize.height - CP_PREF_SIZE.height) / 2);
      setVisible(true);
      addListeners();
    }

    private JPanel createProgramPanel() {
      // Create the program panel
      JPanel programPanel = new JPanel();
      JLabel program = new JLabel("Program");
      program.setPreferredSize(LABEL_PREFERRED_SIZE);
      chbDna = new JCheckBox[BLAST_PROGRAMS_DNA.length];
      String blastProgram;
      for (int i = 0; i < BLAST_PROGRAMS_DNA.length; i++) {
        blastProgram = BLAST_PROGRAMS_DNA[i];
        chbDna[i] = new JCheckBox(blastProgram);
        chbDna[i].setMaximumSize(COMBO_PREFERRED_SIZE);
      }
      chbProtein = new
JCheckBox[BLAST_PROGRAMS_PROTEIN.length];
      for (int i = 0; i < BLAST_PROGRAMS_PROTEIN.length; i++)
{
        blastProgram = BLAST_PROGRAMS_PROTEIN[i];
        chbProtein[i] = new JCheckBox(blastProgram);
        chbProtein[i].setMaximumSize(COMBO_PREFERRED_SIZE);
      }

      programPanel.setLayout(new BoxLayout(programPanel,
BoxLayout.LINE_AXIS));
      programPanel.add(program);
      programPanel.add(Box.createRigidArea(new Dimension(10,
0)));
      for (int i = 0; i < chbDna.length; i++) {
        programPanel.add(chbDna[i]);
        programPanel.add(Box.createRigidArea(new Dimension(5,
0)));
      }
      for (int i = 0; i < chbProtein.length; i++) {
        programPanel.add(chbProtein[i]);
        if (i + 1 < chbProtein.length)
          programPanel.add(Box.createRigidArea(new
Dimension(5, 0)));
      }
      programPanel.add(Box.createHorizontalGlue());
      JPanel paramPanel = new JPanel();
      paramPanel.setLayout(new BoxLayout(paramPanel,
BoxLayout.PAGE_AXIS));

      paramPanel.add(programPanel);
      paramPanel.add(Box.createRigidArea(new Dimension(0,
5)));
```

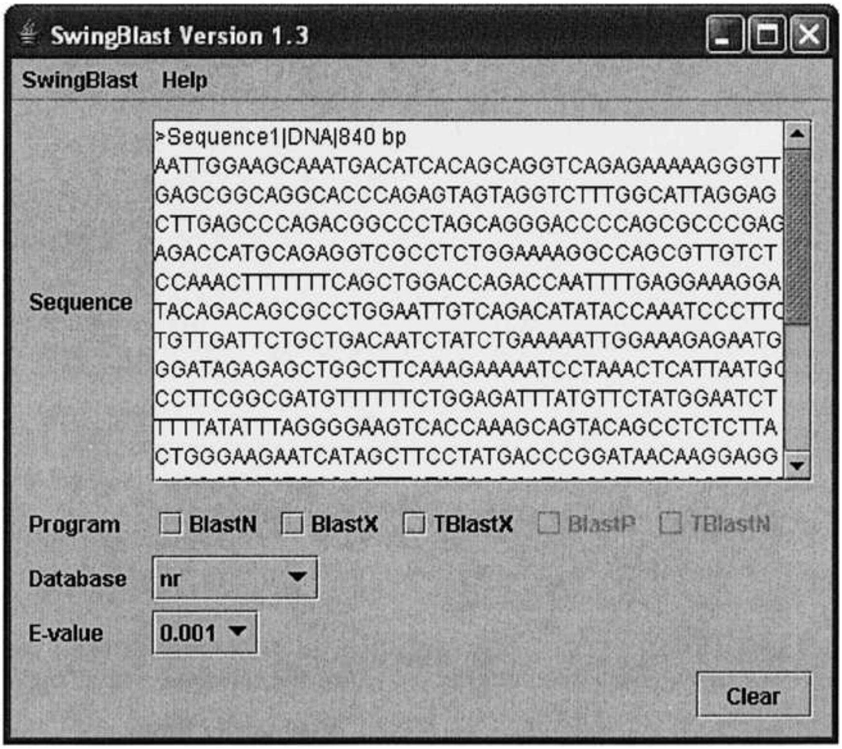
```
      // Create the database panel
      JPanel databasePanel = new JPanel();
      JLabel database = new JLabel("Database");
      database.setPreferredSize(LABEL_PREFERRED_SIZE);
      cobDbs = new JComboBox(DATABASES);
      cobDbs.setMaximumSize(COMBO_PREFERRED_SIZE);

      databasePanel.setLayout(new BoxLayout(databasePanel,
BoxLayout.LINE_AXIS));
      databasePanel.add(database);
      databasePanel.add(Box.createRigidArea(new Dimension(10,
0)));
      databasePanel.add(cobDbs);
      databasePanel.add(Box.createHorizontalGlue());
      paramPanel.add(databasePanel);
      paramPanel.add(Box.createRigidArea(new Dimension(0,
5)));

      // Create the E-Value panel
      JPanel evaluePanel = new JPanel();
      JLabel eValue = new JLabel("E-value");
      eValue.setPreferredSize(LABEL_PREFERRED_SIZE);
      cobEvalues = new JComboBox(EVALUES);
      cobEvalues.setMaximumSize(COMBO_PREFERRED_SIZE);

      evaluePanel.setLayout(new BoxLayout(evaluePanel,
BoxLayout.LINE_AXIS));
      evaluePanel.add(eValue);
      evaluePanel.add(Box.createRigidArea(new Dimension(10,
0)));
      evaluePanel.add(cobEvalues);
      evaluePanel.add(Box.createHorizontalGlue());
      paramPanel.add(evaluePanel);
      paramPanel.add(Box.createRigidArea(new Dimension(0,
5)));

      // Set it up disabled
      enableFunctions(TYPE_UNKNOWN);
      return paramPanel;
    }

    private void addListeners() {
      quitItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          System.exit(0);
        }
      });

      aboutItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          JOptionPane.showMessageDialog(SwingBlastl_3.this,
APP_NAME + " " + APP_VERSION,
            "About " + APP_NAME,
JOptionPane.INFORMATION_MESSAGE);
        }
      });

      clear.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          sequenceArea.setText("");
          enableFunctions(-1);
        }
      });

      sequenceArea.addFocusListener(new FocusListener() {
        public void focusGained(FocusEvent e) {
        }

        public void focusLost(FocusEvent e) {
          // Check if sequence is DNA, RNA or protein
          String text = sequenceArea.getText();

          // Format sequence in FASTA format and retrieve the
// entered sequence
          int idx = text.indexOf(">");
          boolean fastaFormatted = idx != -1;
          String seqText = null;
          String header = null;
          int seqLength = 0;
          String sequence = "";

          if (fastaFormatted) {
            int returnIdx = text.indexOf("\n");

            if (returnIdx != -1) {
              header = text.substring(0, returnIdx);
              sequence = text.substring(returnIdx + 1,
text.length()).replaceAll("\\s", "").toLowerCase();
              seqText = text;
            }
          } else {
            text = text.replaceAll("\\s", "");
            RE re = null;
            try {
              re = new RE("[0-9]+");
            } catch (RESyntaxException e1) {
              e1.printStackTrace();
            }

            boolean isGenBankID = re.match(text);

            if (isGenBankID) {
              GenbankSequenceDB genbankSequenceDB = new
GenbankSequenceDB();
              header = "GI:" + text;
              Sequence seqObject = null;
```

```
       try {
         seqObject =
genbankSequenceDB.getSequence(text);
         SeqIOTools.writeGenbank(System.out,
seqObject);
       } catch (Exception e1) {
         e1.printStackTrace();
       }
       sequence = seqObject.seqString();
      } else {
       sequence = text.toLowerCase();
       header = ">Sequence1|";
       seqLength = text.length();
      }
     }

     // Check if sequence has been entered
     if (sequence.length() == 0)
      return;

     // Determine sequence type
     int typeOfSequence = TYPE_UNKNOWN;
     try {
      typeOfSequence = getSequenceType(sequence);
     } catch (RESyntaxException e1) {
      e1.printStackTrace();
     }

     String type = null;
     String unitOfLength = null;

     switch (typeOfSequence) {
      case TYPE_DNA:
       type = "DNA";
       unitOfLength = " bp";
       break;
      case TYPE_RNA:
       type = "RNA";
       unitOfLength = " bp";
       break;
      case TYPE_PROTEIN:
       type = "Protein";
       unitOfLength = " aa";
       break;
      default:
       type = "N/A";
       unitOfLength = " N/A";
     }

     if (!fastaFormatted) {
      seqText = header + type + "|" + seqLength +
unitOfLength + "\n" + sequence.toUpperCase();
```

```
          }

          // Display results
          sequenceArea.setText(seqText);

          enableFunctions(typeOfSequence);
        }
      });
    }

    private void enableFunctions(int typeOfSequence) {
      if (typeOfSequence == TYPE_DNA || typeOfSequence ==
TYPE_RNA) {
        setChb(chbDna, true);
        setChb(chbProtein, false);
        setCob(cobDbs, true);
        setCob(cobEvalues, true);
      } else if (typeOfSequence == TYPE_PROTEIN) {
        setChb(chbProtein, true);
        setChb(chbDna, false);
        setCob(cobDbs, true);
        setCob(cobEvalues, true);
      } else {
        setChb(chbProtein, false);
        setChb(chbDna, false);
        setCob(cobDbs, false);
        setCob(cobEvalues, false);
      }
    }

    private static void setChb(JCheckBox[] boxes, boolean
value) {
      for (int i = 0; i < boxes.length; i++) {
        boxes[i].setEnabled(value);
      }
    }

    private static void setCob(JComponent component, boolean
value) {
      component.setEnabled(value);
    }

    public static int getSequenceType(String sequence) throws
RESyntaxException {
      RE re = new RE("[actgnACGTN]+");
      String[] strings = re.split(sequence);
      int numbOfLettersOtherThanATGCNs = 0;

      for (int i = 0; i < strings.length; i++) {
        numbOfLettersOtherThanATGCNs += strings[i].length();
      }
      int length = sequence.length();
      int numbOfACGTNs = length -
```

```
numbOfLettersOtherThanATGCNs;

      re = new RE("[uU]+");
      strings = re.split(sequence);
      int numbOfLettersOtherThanUs = 0;

      for (int i = 0; i < strings.length; i++) {
        numbOfLettersOtherThanUs += strings[i].length();
      }
      int numbOfUs = sequence.length() -
numbOfLettersOtherThanUs;

      if (numbOfACGTNs / (double) length > SEQ_THRESHOLD) {
        return TYPE_DNA;
      } else if ((numbOfACGTNs + numbOfUs) / (double) length
> SEQ_THRESHOLD) {
        return TYPE_RNA;
      } else {
        return TYPE_PROTEIN;
      }
    }

    public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          final SwingBlast1_3 view = new SwingBlast1_3();
        }
      });
    }
  }
```

Fig. 2.20 and **Fig. 2.21** show the behavior of the application for a nucleotide and a protein sequence respectively that is entered in the text area. In both cases, the correct set of BLAST programs are selected (BLASTN, BLASTX and TBLASTX for nucleotide sequence and BLASTP and TBLASTN for protein sequence). Simultaneously, the drop-down menu boxes for the databases and the E-value are activated for selection by the user.



Sequence

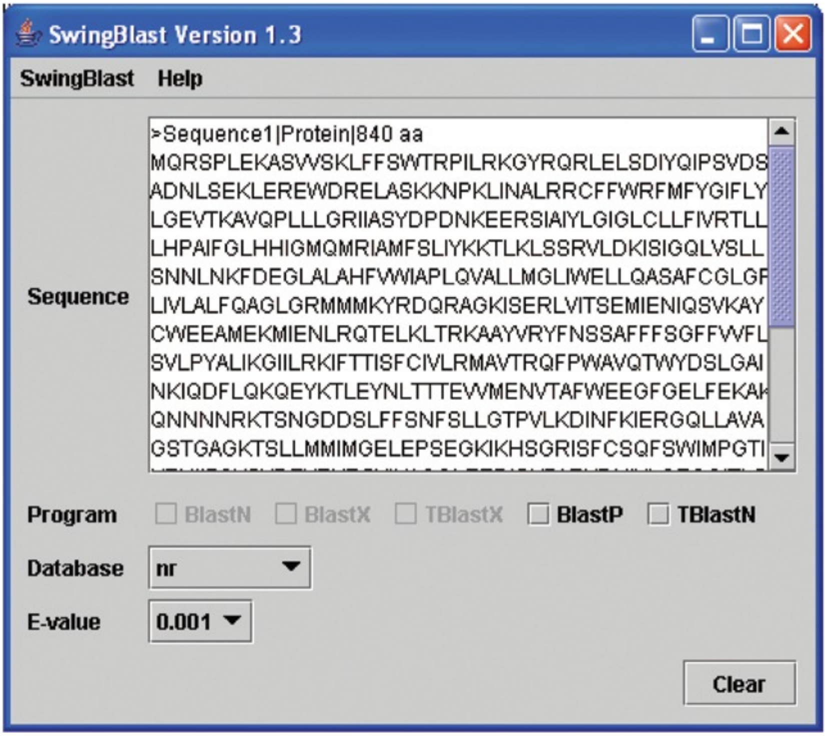
```
>Sequence1|DNA|840 bp
AATTGGAAGCAAATGACATCACAGCAGGTCAGAGAAAAAGGGTT
GAGCGGCAGGCACCCAGAGTAGTAGGTCTTTGGCATTAGGAG
CTTGAGCCCAGACGGCCCTAGCAGGGACCCCAGCGCCCGAG
AGACCATGCAGAGGTCGCCTCTGGAAAAGGCCAGCGTTGTCT
CCAAACTTTTTTTCAGCTGGACCAGACCAATTTTGAGGAAAGGA
TACAGACAGCGCCTGGAATTGTCAGACATATACCAAATCCCTTC
TGTTGATTCTGCTGACAATCTATCTGAAAAATTGGAAAGAGAATG
GGATAGAGAGCTGGCTTCAAAGAAAAATCCTAAACTCATTAATGC
CCTTCGGCGATGTTTTTTCTGGAGATTTATGTTCTATGGAATCT
TTTTATATTTAGGGGAAGTCACCAAAGCAGTACAGCCTCTCTTA
CTGGGAAGAATCATAGCTTCCTATGACCCGGATAACAAGGAGG
```

Program □ BlastN □ BlastX □ TBlastX □ BlastP □ TBlastN

Database nr

E-value 0.001

Fig. 2.20. Displaying BLAST options for a nucleotide sequence



Sequence

```
>Sequence1|Protein|840 aa
MQRSPLEKASVVSKLFFSWTRPILRKGYRQRLELSDIYQIPSVDS
ADNLSEKLEREWDRELASKKNPKLINALRRCFFWRFMFYGIFLY
LGEVTKAVQPLLLGRIIASYDPDNKEERSIAIYLGIGLCLLFIVRTLL
LHPAIFGLHHIGMQMRIAMFSLIYKKTLKLSSRVLDKISIGQLVSLL
SNNLNKFDEGLALAHFVWIAPLQVALLMGLIWELLQASAFCGLGF
LIVLALFQAGLGRMMMKYRDQRAGKISERLVITSEMIENIQSVKAY
CWEEAMEKMIENLRQTELKLTRKAAYVRYFNSSAFFFSGFFVVFL
SVLPYALIKGIILRKIFTTISFCIVLRMAVTRQFPWAVQTWYDSLGAI
NKIQDFLQKQEYKTLEYNLTTTEVVMENVTAFWEEGFGELFEKAK
QNNNNRKTSNGDDSLFFSNFSLLGTPVLKDINFKIERGQLLAVA
GSTGAGKTSLLMMIMGELEPSEGKIKHSGRISFCSQFSWIMPGTI
```

Program □ BlastN □ BlastX □ TBlastX □ BlastP □ TBlastN

Database nr

E-value 0.001

Fig. 2.21. Displaying BLAST options for a protein sequence

Summary

In this Chapter, we created a Swing based application that allows users to prepare sequences for BLAST searches by performing simple formatting tasks such as conversion into the Fasta format and determining the sequence type and length. Along the way we introduced how to write code to respond to events taking place in response to user initiated actions. We created the GUI elements and wrote the code that enables the elements to respond to the sequence type and present only the valid BLAST options that are available for the entered sequence type. The rationale for building these features into the application was to make it more functional and to simplify its use for the end-users, given the many potentially confusing parameters a user has to supply when performing a search operation. In the next Chapter, we will extend the `SwingBlast` application to actually perform the BLAST search operation.

Questions and Exercises

1. Enhance the SwingBlast application interface to accept multiple sequences, for example, by incorporating the ability to upload a multiple Fasta file. Next incorporate code to add checkboxes against each uploaded sequence to allow users to select specific sequences for further analysis. Develop the use cases that fulfill the above user requirements.

2. Explore the BLAST algorithms in further detail by visiting the tutorial site listed below. How do you determine the statistical significance of BLAST hits? What are bit scores and p-values?

3. Download the sequence for simian sarcoma virus v-sis oncogene gene from GenBank and perform a BLAST against the nr database. What BLAST program(s) would you use to find similarities between v-sis and existing nucleotide and protein sequences? What are the top ten hits that BLAST returns? Which human and other vertebrate homologs can you identify?

Additional Resources

- BLAST tutorial - http://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-1.html

- GenBank - http://www.ncbi.nlm.nih.gov/Genbank/index.html

- Java^{TM} 2 Platform Standard Edition 5.0 API Specification - http://java.sun.com/j2se/1.5.0/docs/api/

Selected Reading

Simian sarcoma virus onc gene, v-sis, is derived from the gene (or genes) encoding a platelet-derived growth factor. Doolittle RF, Hunkapiller MW, Hood LE, Devare SG, Robbins KC, Aaronson SA, Antoniades HN. Science. 1983 Jul 15;221(4607):275–277.

Identification of the cystic fibrosis gene: cloning and characterization of complementary DNA. Riordan JR, Rommens JM, Kerem B, Alon N, Rozmahel R, Grzelczak Z, Zielenski J, Lok S, Plavsic N, Chou JL, et al. Science. 1989 Sep 8;245(4922):1066-73.

Basic local alignment search tool. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. J Mol Biol. 1990 Oct 5;215(3):403-10.

Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Altschul SF, Madden TL, Schaffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ. Nucleic Acids Res. 1997 Sep 1;25(17):3389-402.