Java Cookbook

Matthias Baitsch

May 2009

Introduction

This document provides strategies how to solve common Java programming tasks. In detail, the following topics are covered:

1. using elementary Java classes for arrays and strings,
2. reading from a file and writing to a file,
3. some Swing classes,
4. using a table to render and manipulate structured data,
5. using linear algebra classes and solving linear systems,
6. plotting functions,
7. defining a mathematical function,
8. creating two-dimensional drawings using coordinate transformation,
9. managing a graphical application including animation.

For each problem, some background information and a straightforward application example that can be adapted to your actual problem is provided.

Since the examples in this document are based on various external libraries, the “ICEB MPCE” plug-in has to be installed for eclipse. Please consider the Internet pages for the “Modern Programming Concepts in Engineering” course for details.

1 Elementary Java classes

1.1 Working with arrays

Java ships with a class `java.util.Arrays` which provides handy and efficient methods for common array tasks. Furthermore, the use of so called vararg-arguments can simplify code significantly.

Printing In order to quickly print an array to the console, the `toString` and `deepToString` methods can be used:

```
double[] a1 = { 6, 3, 9.99, 1 };
double[][] a2 = { { 3, 6, 2, 99 }, { 22, -1 } };
System.out.println("a1 = " + Arrays.toString(a1));
System.out.println("a2 = " + Arrays.deepToString(a2));
```

where the output is

```
a1 = [6.0, 3.0, 9.99, 1.0]
a2 = [[3.0, 6.0, 2.0, 99.0], [22.0, -1.0]]
```

Note that the `deepToString` method applies the `toString` method recursively to arrays of arrays (which might again contain arrays).

Sorting In order to sort an array, the `sort` method can be applied

```
double[] a3 = { 66, -99.4, 6, 3, 9.99, 1 };
Arrays.sort(a3);
System.out.println("a3 = " + Arrays.toString(a3));
```

Note that in the output, the array is sorted:

```
a3 = [-99.4, 1.0, 3.0, 6.0, 9.99, 66.0]
```

The `sort` method uses a variant of the quicksort algorithm which offers $n \log n$ performance on many data sets that cause other quicksorts to degrade to quadratic performance.

Searching In order to find the index of an entry in a *sorted* array, the `binarySearch` method can be used:

```
double[] a4 = { -99.4, 1.0, 3.0, 6.0, 9.99, 66.0 };
System.out.println("Index of 6 in a4: " + Arrays.binarySearch(a4, 6));
System.out.println("Index of 7 in a4: " + Arrays.binarySearch(a4, 7));
```

The method returns the index of the specified element, if the element is present in the array. Otherwise, a negative number n is returned such that $-n-1$ is the index of the first element larger than the specified number. Consequently, the output of the above code is

```
Index of 6 in a3: 3
Index of 7 in a3: -5
```

Note that the result of `binarySearch` is unspecified if the input array is not sorted in ascending order. As the name of the method implies, a binary search algorithm is employed such that the time complexity is $\log n$.

Varargs parameters Passing an arbitrary number of items as parameters to a method requires array type parameters. Without the use of vararg parameters, the array has to be created before invoking the method (see first two lines of `main` method). Using varargs, an arbitrary number of parameters can be passed to a method (see lines three and four of `main`). In many situations, this can save a lot of typing and makes the code much more readable. In order to use varargs in a method, you just use `type...` instead of `type[]`, e.g. `int...` instead of `int[]`. The only limitation is that you can have only one vararg type parameter and that it must be the last parameter in the parameter list.

```
public class Varargs {

   public static void main(String[] args) {
      double[] a = { 1, 2, 3 };
      print("a1", a);
      print("a2", 1, 2, 3);
      print("a3", 1, 2, 3, 4);
   }

   private static void print(String l, double... x) {
      System.out.print(l + " = ");
      for (int i = 0; i < x.length; i++) {
         System.out.print(x[i] + " ");
      }
      System.out.println();
   }
}
```

1.2 Strings – conversion, formatting and splitting

Converting Strings into elementary data types. When reading working with files or graphical user interfaces, data is provided as an object of type **`String`** where an elementary data type (such as `int` or `double`) is required for further processing. This problem can be solved using the classes accomplishing elementary types:

```
int b = Integer.parseInt("187462897");
double c = Double.parseDouble("33.3");
long d = Long.parseLong("84987627854923874");
```

Note that it must be possible to convert the parameter passed to the parse-method into the corresponding type; otherwise an exception will be raised. For instance, the statement

```
double e = Double.parseDouble("abc");
```

results in a **NumberFormatException** to be thrown since there is no common convention how to convert “abc” into a number. Serious applications have to handle such a situation adequately.

Formatting numbers Outputting numbers in a nicely formatted fashion is in Java not straightforward. By default, the full number of digits is included when converting a number into a string, such that

```
double f = 100000.0 / 3.0;
System.out.println(”f=” + f);
```

gives

```
f=33333.333333333336
```

as output. Often, this large number number of displayed digits is not acceptable. To solve this problem, there basically exist two approaches: using a dedicated class or using a special print method.

The first solution is to use an object of type `java.text.NumberFormat`. Basic usage is

```
double f = 100000.0 / 3.0;
NumberFormat nf1 = DecimalFormat.getNumberInstance();
NumberFormat nf2 = new DecimalFormat(”0.000000E00#”);
System.out.println(”Format 1: ” + nf1.format(f));
System.out.println(”Format 2: ” + nf2.format(f));
```

where the corresponding output to the console is

```
Format 1: 33.333,333
Format 2: 3.3333333E04
```

The first number format object uses the default settings while for the second object, the formatting rules are specified explicitly. Note that the actual output might also depend on the language settings of your computer system.

The second option is to use the `printf` method defined in the **PrintStream** class. For example,

```
double f = 100000.0 / 3.0;
System.out.printf(”f=%15.2f\n”, f);
```

results in

```
f=        33333.33
```

The `printf` methods takes as parameters one string and an arbitrary number of further parameters. The first string describes how the remaining parameters are formatted. In the above example, we format the second parameter as floating point number reserving space for 15 characters while printing 2 decimal digits. This options is often suitable for printing numbers in tables but using it is not trivial and requires some reading of the documentation.

Splitting strings into pieces. A common task when reading files containing numerical data is to split one line containing many numbers into pieces each containing one number. The **`String`** class contains the handy method `split` for this purpose. The `split` method takes a string as input which describes the deliminator and returns an array of strings located in between the deliminators. Example: The statements

```
String s = "987 34.1 9982 66 90  7";
String[] a1 = Arrays.toString(s.split(" "));
String[] a2 = Arrays.toString(s.split("99"));
```

generate the arrays

```
a1 = ["987", "34.1", "9982", "66", "90", "", "7"]
a2 = ["987 34.1 ", "82 66 90 7"]
```

where the first string is split at blanks and the second one at the character sequence "99". If there are two adjacent deliminators, an empty string is inserted such as in the `a1` array between "90" and "7" because there are two blanks between 90 and 7 in the input string. For more complicated situations, it might be useful to know that the input string to the `split` method is interpreted as an Regular expression. For instance, in

```
String s2 = "987     ,  34.1, 9982,66        ,90 7";
String[] a3 = s2.split(" *, *")));
```

the deliminator is specified as an arbitrary number of blanks followed by a comma followed by an arbitrary number blanks. Thus, the resulting array is

```
a3 = ["987", "34.1", "9982", "66", "90 7"];
```

However, regular expressions are powerful but rather complex, so further reading is required in order to make more use of this concept.

2 File IO with Java

Reading and writing files in Java is not as straightforward as in other programming languages such as FORTRAN or C. There is a large number of classes and understanding the whole `java.io` package in detail is rather cumbersome. However, the most basic tasks – reading from a file and writing to a file – are easy if you know how.

Reading a file The demo application in Listing 1 reads its own source code and prints it to the console. Most of the work is done in the `list` method which takes the name of the file to list as input. In the first line of this method, a **`BufferedReader`** is created; the variable `br` points to this object. The **`BufferedReader`** class has basically one important method: `readLine`. It returns either one single line of the input file or `null` if the end of the file has been reached. Consequently, the code to process a whole file works as follows: First, we read the first line of the file and let the variable `line` point to it. Then, as long as the `line` is not `null`, we process the current line (in this example, it is just printed) first and then read another line. Note that the `list` method declares to throw an **`IOException`** since the specified file might not exist.

From the `main` method, the `list` method is invoked and the file to be listed is specified as parameter. Here, the file name is specified relative to the working directory of the Java program; in eclipse, this is the project folder. Furthermore, it can be seen that the forward slash "/" is used as directory separator. This works on all platforms since Java correctly converts the "/" into the separator which is adequate for the actual platform. In this example, only basic exception handling is implemented; in a graphical application, it would be adequate to use a message box (see Section 3.3) in order to handle an **`IOException`**.

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

public class ReadFileDemo {

   public static void main(String[] args) throws IOException {
      list("src/ReadFileDemo.java");
   }

   private static void list(String filename) throws IOException {
      BufferedReader br = new BufferedReader(new FileReader(filename));
      String line = br.readLine();

      while (line != null) {
         System.out.println(line);
         line = br.readLine();
      }
   }
}
```

Listing 1: Reading from a file

Writing to a file In order to write to a file, the **`PrintWriter`** class is very convenient. The example in Listing 2 shows basic usage of this class. First, a print writer is created; the filename is specified in the constructor. Then, several lines are written to the file and finally, the file is closed. Since usually data is buffered in memory before it is written to a file (in order to enhance performance), the `close` method should be invoked in order to make sure that every single byte has been written to disk.

```
import java.io.IOException;
import java.io.PrintWriter;

public class WriteFileDemo {

   public static void main(String[] args) throws IOException {
      write("test.txt");
   }

   private static void write(String filename) throws IOException {
      PrintWriter pw = new PrintWriter(filename);

      for (int i = 0; i < 100000; i++) {
         pw.println("Line " + i);
      }
      pw.close();
   }
}
```

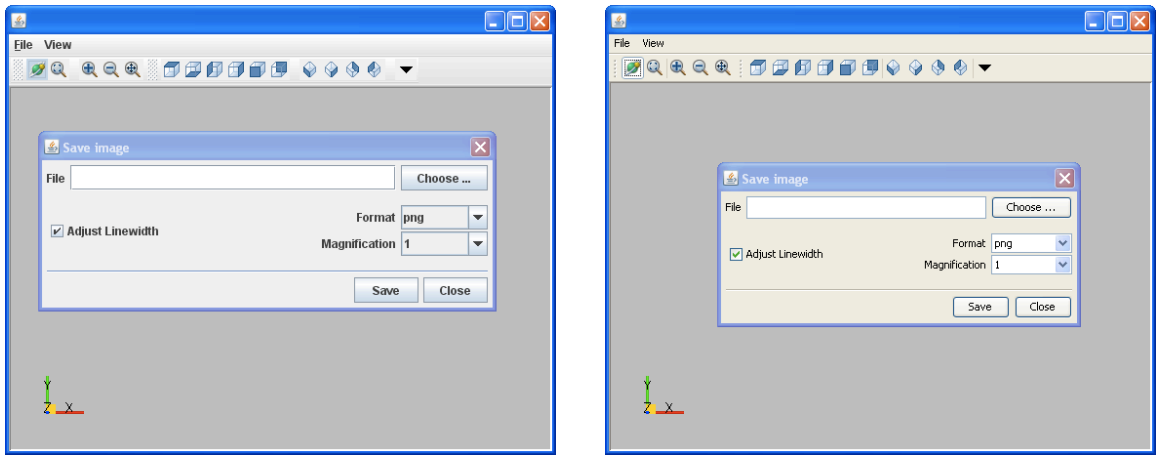
Listing 2: Writing to a file

3 Swing programming

3.1 Setting the look and feel

The Swing package lets programmers choose between different styles of graphical user interfaces, so called look and feels (L&F). On Windows systems, a look and feel called “Metal” is used as default, however, this style does not match everybody’s taste. Using the **`UIManager`** class, the L&F can be changed (preferably in a static block) before the actual program is started. In the following example, the system L&F is used, which is supposed to have an appearance similar to native applications on the actual system. Figure 1 shows a comparison of Metal and the system L&F on a Windows XP system.

```
import javax.swing.UIManager;

public class LookAndFeel {

   static {
      try {
         UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
      } catch (Exception e) {}
   }

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



(a) Metal L&F

(b) System L&F

Figure 1: Swing look and feels on Windows XP

3.2 Selecting files

In order to let users of an application select a file to open or to save to, the **`JFileChooser`** class can be used. Using this class basically involves three steps as shown in the listing below:

1. Creating the **`JFileChooser`** object. Several constructurs can be used (see Javadoc); in the example we tell the file chooser to use the working directory of the actual program.

2. Let the file chooser appear on screen. Depending on whether you want to open a file or save to a file, you either use `showOpenDialog` or `showSaveDialog`. Both methods return an integer as return code.

3. Further processing depends on the value of the return code. If the user approved the dialog, the selected file is retrieved from the file chooser using the `getSelectedFile` method. Otherwise, no specific action is taken.

```
public class FileChooser extends JFrame implements ActionListener {

   public static void main(String[] args) {
      new FileChooser().setVisible(true);
   }

   private JButton b1_ = new JButton("Open...");
   private JButton b2_ = new JButton("Save as...");

   public FileChooser() {
      b1_.addActionListener(this);
      b2_.addActionListener(this);
      setLayout(new GridLayout(1, 2));
      add(b1_);
      add(b2_);
      pack();
   }

   public void actionPerformed(ActionEvent e) {
      int rc;
      JFileChooser fc = new JFileChooser(System.getProperty("user.dir"));
      if (e.getSource() == b1_) {
         rc = fc.showOpenDialog(this);
      } else {
         rc = fc.showSaveDialog(this);
      }
      if (rc == JFileChooser.APPROVE_OPTION) {
         File file = fc.getSelectedFile();
         System.out.println(file);
      }
   }
}
```

Listing 3: File chooser

3.3 Standard dialogs

In various situation, applications have to inform users about a specific situation or ask for confirmation of a certain action. The **`JOptionPane`** class provides convenient static methods to display corresponding dialogs (see Table 1). An example for basic usage of the of the **`JOptionPane`** is given in Listing 4. More information can be found in the Javadoc or in the excellent Swing tutorial under `http://java.sun.com/docs/books/tutorial/uiswing/components/dialog.html`.

```
public class StandardDialogsDemo extends JFrame implements ActionListener {

   public static void main(String[] args) {
      new StandardDialogsDemo().setVisible(true);
   }

   private JButton b1_ = new JButton("Show confirm dialog...");
   private JButton b2_ = new JButton("Show input dialog");
   private JButton b3_ = new JButton("Show message dialog");

   public StandardDialogsDemo() {
      b1_.addActionListener(this);
      b2_.addActionListener(this);
      b3_.addActionListener(this);
      setLayout(new GridLayout(0, 1));
      add(b1_);
      add(b2_);
      add(b3_);
      pack();
   }

   public void actionPerformed(ActionEvent e) {
      if (e.getSource() == b1_) {
         int rc = JOptionPane.showConfirmDialog(this,
               "File exists. Override?");
         if (rc == JOptionPane.YES_OPTION) {
            System.out.println("Yes clicked");
         } else if (rc == JOptionPane.NO_OPTION) {
            System.out.println("No clicked");
         } else {
            System.out.println("Cancelled");
         }
      } else if (e.getSource() == b2_) {
         String input = JOptionPane.showInputDialog("What is your name?");
         System.out.println("You entered: " + input);
      } else if (e.getSource() == b3_) {
         JOptionPane.showMessageDialog(this, "Message in a bottle");
      }
   }
}
```

Listing 4: Standard dialogs

Method Name	Description
showConfirmDialog	Asks a confirming question, like yes/no/cancel
showInputDialog	Prompt for some input
showMessageDialog	Tell the user about something that has happened

Table 1: Static methods of the **`JOptionPane`** class

4 How to use a `JTable` to manage structured data

The **`JTable`** class is useful when it comes to displaying and editing structured data. As many Swing components, **`JTable`** uses the MVC pattern. As shown in Figure 2, a table is associated with an object that implements the **`TableModel`** interface. This object is the model. On the other side, the table is at the same time view and controller.

If you want to use **`JTable`** in your own application, it is often practical to subclass **`AbstractTableModel`**. This class implements some basic functionality of a table model. Often, such subclasses act as adapter providing the application data in a format required by **`JTable`**.

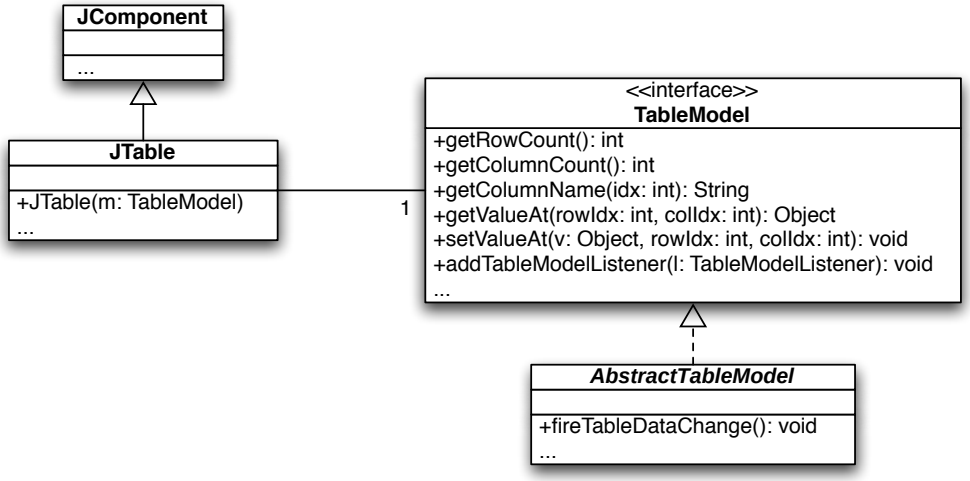


Figure 2: Architecture of **`JTable`**

4.1 A table for polygons

Consider for instance the case where you want to use a **`JTable`** to edit a polygon. Using the table, the user should be able to add vertices, to remove vertices and to modify vertices. Our solution basically involves three data types: The i) **`Polygon`** class represents the model. Objects that implement the ii) **`PolygonView`** interface are views on the model. The iii) **`PolygonToTableModelAdapter`** class adapts the polygon to the **`JTable`** data model.

Figure 3 shows the corresponding classes in a UML class diagram. The **`PolygonToTableModelAdapter`** class implements the **`PolygonView`** interface and is thus notified upon changes of the **`Polygon`** object it belongs to. In the `update` method, the **`fireTableDataChange`** method defined in the superclass is invoked such that the **`JTable`** can update itself.

The following pages contain the implementation of the classes in Figure 3. If your problem is similar to the polygon example, use the code below as a starting point for your own application. Replace the **`Polygon`** class with your model and modify the adapter class according to your needs.

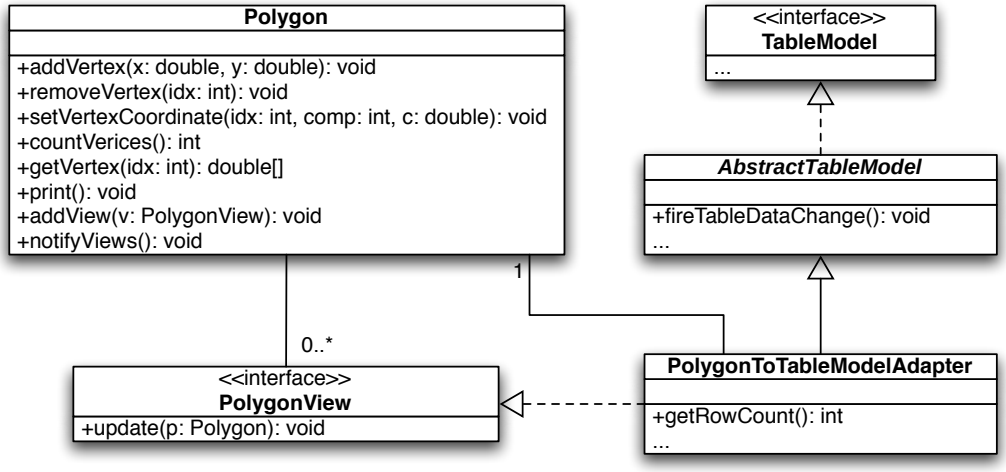


Figure 3: Adapter which bridges a **`Polygon`** object to a **`TableModel`**

Finally, the above classes are embedded in a simple application. First, a **`PolygonEditorPanel`** is introduced that contains the table and three additional buttons that i) print the table, ii) add a row and iii) remove a row. Note that in the `actionPerformed` method, the polygon is modified and then asked to notify its views. Thus, the **`PolygonEditorPanel`** acts as controller.

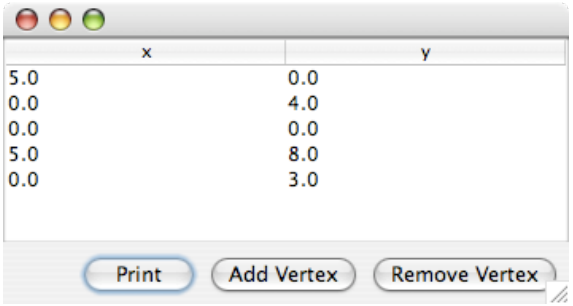


Figure 4: The polygon editor application

The application's main class is **`PolygonEditor`**. Figure 4 shows the running application after some vertices have been added.

```
public interface PolygonView {
   public void update(Polygon p);
}
```

Listing 5: The **`PolygonView`** interface

```
import java.util.*;

public class Polygon {
   private List<double[]> vertices_ = new LinkedList<double[]>();
   private List<PolygonView> views_ = new LinkedList<PolygonView>();
```

```
   public void addVertex(double x, double y) {
      vertices_.add(new double[] { x, y });
   }

   public void removeVertex(int idx) {
      vertices_.remove(idx);
   }

   public void setVertexCoordinate(int idx, int comp, double val) {
      getVertex(idx)[comp] = val;
   }

   public int countVertices() {
      return vertices_.size();
   }

   public double[] getVertex(int idx) {
      return vertices_.get(idx);
   }

   public void print() {
      for (int i = 0; i < vertices_.size(); i++) {
         System.out.println(Arrays.toString(vertices_.get(i)));
      }
   }

   public void addView(PolygonView view) {
      views_.add(view);
      view.update(this);
   }

   public void notifyViews() {
      for (int i = 0; i < views_.size(); i++) {
         views_.get(i).update(this);
      }
   }
}
```

Listing 6: The **`Polygon`** class

```
import javax.swing.table.*;

public class PolygonToTableModelAdapter extends AbstractTableModel implements
      PolygonView {

   private Polygon polygon_;

   public PolygonToTableModelAdapter(Polygon polygon) {
      polygon_ = polygon;
   }

   public int getColumnCount() {
      return 2;
   }
```

```
   public int getRowCount() {
      return polygon_.countVertices();
   }

   public String getColumnName(int columnIdx) {
      if (columnIdx == 0) {
         return "x";
      }
      return "y";
   }

   public Object getValueAt(int rowIdx, int columnIdx) {
      return polygon_.getVertex(rowIdx)[columnIdx];
   }

   public boolean isCellEditable(int rowIndex, int columnIndex) {
      return true;
   }

   public void setValueAt(Object value, int rowIdx, int columnIdx) {
      double coord = Double.parseDouble(value.toString());

      polygon_.setVertexCoordinate(rowIdx, columnIdx, coord);
   }

   public void update(Polygon p) {
      fireTableDataChanged();
   }
}
```

Listing 7: The **`PolygonToTableModelAdapter`** class

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

public class PolygonEditorPanel extends JPanel implements ActionListener {

   private JButton addVertexB_ = new JButton("Add Vertex");
   private JButton printB_ = new JButton("Print");
   private JButton removeVertexB_ = new JButton("Remove Vertex");
   private JTable table_;
   private Polygon polygon_;
   private PolygonToTableModelAdapter adapter_;

   public PolygonEditorPanel(Polygon p) {
      polygon_ = p;
      adapter_ = new PolygonToTableModelAdapter(polygon_);
      table_ = new JTable(adapter_);
      polygon_.addView(adapter_);

      printB_.addActionListener(this);
      addVertexB_.addActionListener(this);
      removeVertexB_.addActionListener(this);
```

```
      setLayout(new BorderLayout());
      add(new JScrollPane(table_), BorderLayout.CENTER);

      JPanel p1 = new JPanel(new FlowLayout(FlowLayout.RIGHT));
      p1.add(printB_);
      p1.add(addVertexB_);
      p1.add(removeVertexB_);
      add(p1, BorderLayout.SOUTH);
   }

   public void actionPerformed(ActionEvent e) {
      if (e.getSource() == printB_) {
         polygon_.print();
      } else if (e.getSource() == addVertexB_) {
         polygon_.addVertex(0, 0);
         polygon_.notifyViews();

      } else if (e.getSource() == removeVertexB_) {
         int row = table_.getSelectedRow();

         if (row != -1) {
            polygon_.removeVertex(row);
            polygon_.notifyViews();
         }
      }
   }
}
```

Listing 8: The **`PolygonEditorPanel`** class

```
import javax.swing.JFrame;

public class PolygonEditor extends JFrame {

   public static void main(String[] args) {
      new PolygonEditor().setVisible(true);
   }

   public PolygonEditor() {
      Polygon p = new Polygon();
      PolygonEditorPanel polygonEditorPanel = new PolygonEditorPanel(p);

      add(polygonEditorPanel);
      setSize(500, 500);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
   }
}
```

Listing 9: The **`PolygonEditor`** class

5 Linear algebra and systems of linear equations

For linear algebra and the solution of linear systems of equations, you can use classes from the `inf.jlinalg` package. The most important classes are

`IVector` Interface that defines operations for vectors like get/set entry, get size and so on.

`ArrayVector` An implementation of the **`IVector`** interface that uses a simple Java array.

`Vector3D` A class that represents vectors with three components. A set of arithmetical operations for vectors (add, subtract, dot product, vector product and so on) is provided.

`IMatrix` This interface defines operations for $m \times n$ matrices. Operations include: get the number of rows and columns, set and get individual entries and add a value to an entry. Note that no arithmetical operations are defined.

`Array2DMatrix` This class is an implementation of the **`IMatrix`** interface that is backed by a two-dimensional array.

`BLAM` A set of Basic Linear Algebra Methods (vector and matrix multiplication, addition and so on) is provided by this class.

`GeneralMatrixLSESolver` A solver for systems of linear equations that uses a LU-factorization of the coefficient matrix.

`MatrixFormat` Using this class, you can format a matrix for printing.

While using the classes, you will have to consider the online documentation extensively. In the following sections, the classes are introduced by three examples.

5.1 Matrix multiplication

We compute the matrix product

$$\mathbf{C} = \mathbf{A}^T\mathbf{B}\,\mathbf{A}$$

for the matrices

$$\mathbf{A} = \begin{bmatrix} 1 & -6 & 3 \\ -4 & 2 & 5 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -1 & 3 \\ 9 & 2 \end{bmatrix}.$$

In Listing 10 first the matrix objects are constructed by specifying the size. After initializing **A** and **B**, first the result of $\mathbf{A}^T\mathbf{B}$ is stored in the variable `tmp`. In the second step, **C** is computed. For an explanation of the parameters for `BLAM.multiply` consider the online documentation.

```
import inf.jlinalg.*;

public class BlamExample {

    public static void main(String[] args) {
        IMatrix a = new Array2DMatrix(2, 3);
        IMatrix b = new Array2DMatrix(2, 2);
        IMatrix tmp = new Array2DMatrix(3, 2);
        IMatrix c = new Array2DMatrix(3, 3);
        // initialize A and B
        a.set(0, 0, 1);
        a.set(0, 1, -6);
        a.set(0, 2, 3);
        a.set(1, 0, -4);
        a.set(1, 1, 2);
        a.set(1, 2, 5);
        b.set(0, 0, -1);
        b.set(0, 1, 3);
        b.set(1, 0, 9);
        b.set(1, 1, 2);
        // compute
        BLAM.multiply(1.0, BLAM.TRANSPOSE, a,
                BLAM.NO_TRANSPOSE, b, 0.0, tmp);
        BLAM.multiply(1.0, BLAM.NO_TRANSPOSE, tmp,
                BLAM.NO_TRANSPOSE, a, 0.0, c);
        // print
        System.out.println("Matrix A");
        System.out.println(MatrixFormat.format(a));
        System.out.println("Matrix B");
        System.out.println(MatrixFormat.format(b));
        System.out.println("Matrix C = A^T B A");
        System.out.println(MatrixFormat.format(c));
    }
}
```

Listing 10: Matrix multiplication

5.2 Solving a system of linear equations

In this example, we use the `inf.jlinalg` package to solve the system of linear equations

$$\begin{bmatrix} n+1 & 1 & 1 & & 1 \\ 1 & 1 & 0 & \cdots & 0 \\ 1 & 0 & 1 & & 0 \\ & \vdots & & \ddots & \vdots \\ 1 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 2n \\ 2 \\ 2 \\ \vdots \\ 2 \end{bmatrix}$$

where n is the number of equations. Obviously, the solution is $\mathbf{x} = [1 \quad 1 \cdots 1]^T$. Listing 11 shows the Java program that sets up and solves the above system of linear equations.

```
import inf.jlinalg.*;
import inf.text.ArrayFormat;

public class SolverExample {

   public static void main(String[] args) throws SolveFailedException {
      int neq = 4;
      ILSESolver solver = new GeneralMatrixLSESolver();
      QuadraticMatrixInfo aInfo = solver.getAInfo();
      IMatrix a = solver.getA();
      double[] b = new double[neq];

      aInfo.setSize(neq);
      solver.initialize();
      a.set(0, 0, neq + 1);
      b[0] = 2 * neq;
      for (int i = 1; i < b.length; i++) {
         a.set(0, i, 1);
         a.set(i, 0, 1);
         a.set(i, i, 1);
         b[i] = 2;
      }

      System.out.println("Solving A x = b");
      System.out.println("Matrix A");
      System.out.println(MatrixFormat.format(a));
      System.out.println("Vector b");
      System.out.println(ArrayFormat.format(b));
      // after calling solve, b contains the solution
      solver.solve(b);
      System.out.println("Solution x");
      System.out.println(ArrayFormat.format(b));
   }
}
```

Listing 11: Using the equation solver

6 Plotting data

There exist many Java packages that can be used to generate plots of data. The ptolemy project contains the Ptplot tool (`http://ptolemy.eecs.berkeley.edu/java/ptplot`) which is easy to use but slightly limited in functionality. A more powerful but very complex alternative is the JFreechart package available at `http://www.jfree.org/jfreechart`.

Use Ptplot for simple tasks which have to be completed quickly and switch to JFreechart for more complex problems where you need full control over every little detail. In the following, basic principles for both packages are provided; for more information, please consider the original documentation.

6.1 Using Ptplot

The Ptplot software design is very simple and basically comprises a single class **`Plot`** which is derived from **`JPanel`** (see Figure 5).

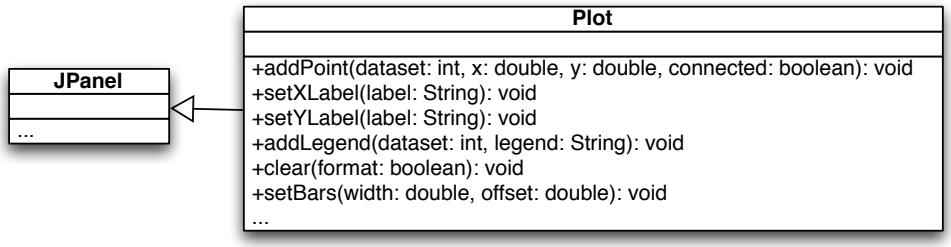


Figure 5: UML class-diagram of the Ptplot **`Plot`** class

Basic plotting The listing of the **`PtplotDemo`** (see Listing 12) demonstrates basic usage of the Ptplot **`Plot`** class. The actual plot is created in the `createPlot` method: In a loop the dataset points for the functions are added to the plot using the `addPoint` method. The corresponding plot is shown in Figure 6(a).

Adding labels and a legend In order to add some labels to the plot, the following statements can be added to the `createPlot` method:

```
plot.setXLabel("x");
plot.setYLabel("y");
plot.addLegend(0, "sin(x)");
plot.addLegend(1, "cos(x)");
```

Note that the first parameter of the `addLegend` method corresponds to the first parameter of the `addPoint` method; both specify the affected dataset (see Figure 6(b)).

Clearing a plot In order to clear a plot, for instance in order to start from the beginning, the `clear` method has to be used. Specify **`true`** as parameter, if you want the labels of the plot to be cleared and **`false`** otherwise.

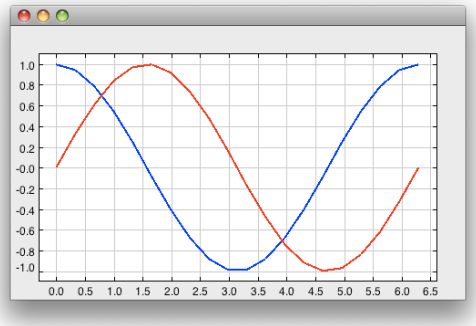
```
public class PtplotDemo extends JFrame {

   public static void main(String[] args) {
      new PtplotDemo().setVisible(true);
   }

   public PtplotDemo() {
      Plot plot = createPlot();
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      add(plot, BorderLayout.CENTER);
      pack();
      setVisible(true);
   }

   private Plot createPlot() {
      Plot plot = new Plot();
      int n = 20;
      double dx = 2 * Math.PI / (n - 1);

      for (int i = 0; i < n; i++) {
         double x = i * dx;
         plot.addPoint(0, x, Math.sin(x), true);
         plot.addPoint(1, x, Math.cos(x), true);
      }

      plot.clear(false);
      return plot;
   }
}
```

Listing 12: Basic plotting with Ptplot

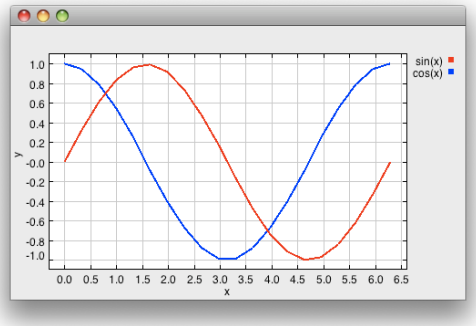
Plotting with bars In order to create a plot with bars, two changes to the original code are required:

```
  private Plot createPlot() {
    Plot plot = new Plot();
    int n = 20;
    double dx = 2 * Math.PI / (n - 1);
      for (int i = 0; i < n; i++) {
      double x = i * dx;
      plot.addPoint(0, x, Math.sin(x), false);
      plot.addPoint(1, x, Math.cos(x), false);
    }
    plot.setBars(dx / 2, dx / 2);
    return plot;
  }
```

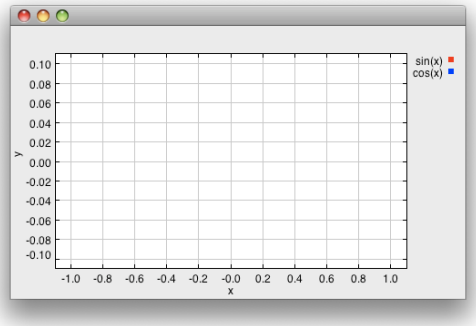
Firstly, we specify `false` as last parameter to the `addPoint` method in order to indicate, that no line should be drawn. Secondly, the `setBars` method (after the loop) is used to indicate, that bars should be plotted. The first parameter specifies the bar width and the second parameter the offset of the bars. Experiment with the second parameter in order to understand how it works.



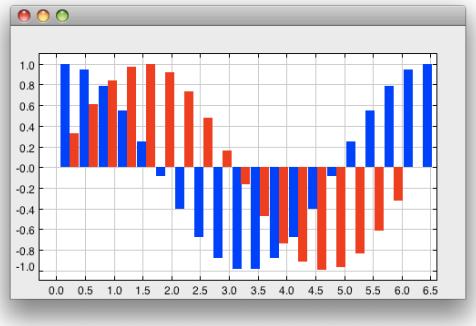
(a) Basic plot



(b) Plot with labels



(c) After invoking `plot.clear(false);`



(d) A plot with bars

Figure 6: Plot examples generated with Ptplot

6.2 JFreechart

6.2.1 JFreechart software design

JFreechart comprises more than 500 classes and is rather complex. Therefore, a bit of background information makes it easier to make use of the package. The JFreechart software design makes a clear distinction between data and representation. Basically, there are three layers: Data, plot generation and plot rendering. Consider the UML-class diagram in Figure 7: Data is represented by object implementing the **`Dataset`** interface. Very often, you want to create standard plots in a xy coordinate system. Use a class implementing the **`XYDataset`** interface such as **`DynamicDataset`** for this purpose. Plots are generated by **`JFreeChart`** objects that basically consist of a **`Plot`** object plus some additional information like a plot title and so on. The **`Plot`** object is responsible for rendering the data. Many subclasses

exist; very often the **`XYPlot`** class will be most suitable. Finally, a plot is drawn on the screen by a **`ChartPanel`** object which is a specialized **`JPanel`**.

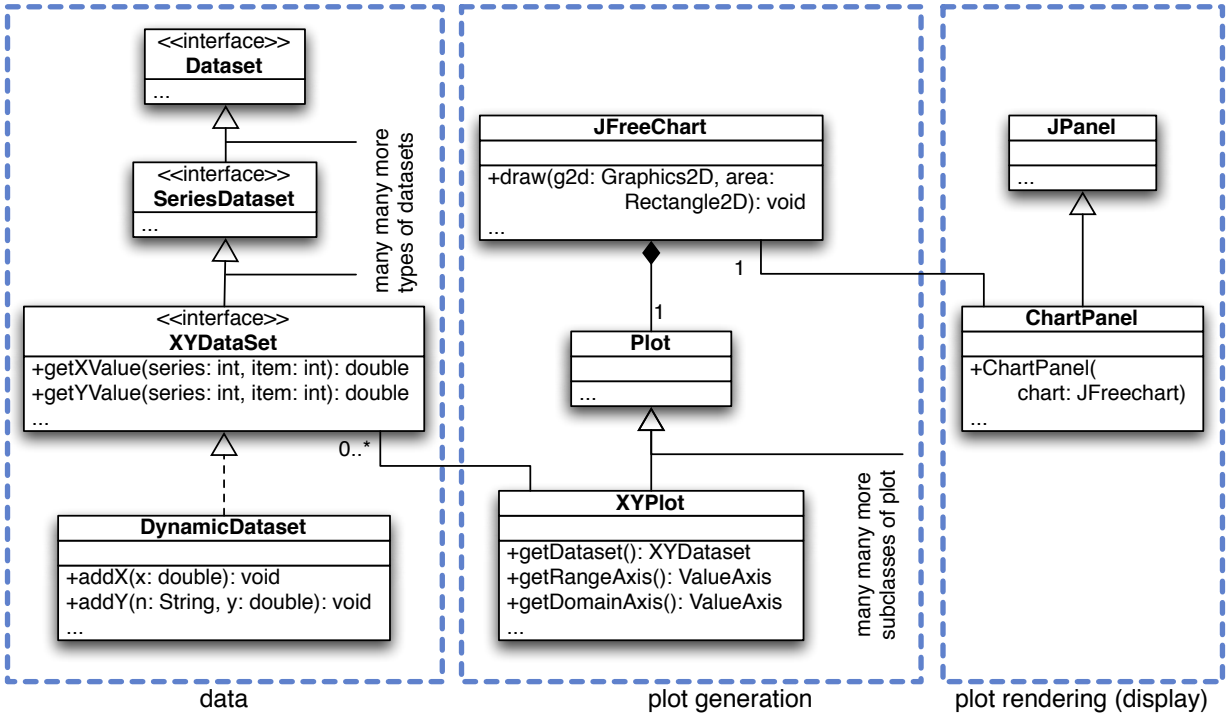


Figure 7: Some JFreechart classes

Following the JFreechart design principles, creating a plot with JFreechart involves three steps:

1. Construct a **`Dataset`** object.
2. Obtain a **`JFreechart`** object using the **`ChartFactory`** class.
3. Create a **`ChartPanel`** and add it to your frame.

6.2.2 A plot panel

This example introduces the **`PlotPanel`** class that can be directly used in your application to easily plot a **`XYDataset`**. The constructor of **`PlotPanel`** takes the plot title, the axis labels and the dataset as arguments. The code of the **`PlotPanel`** is in Listing 13. A simple application using the plot panel is given in Listing 14.

```
public class PlotPanel extends JPanel {

   public PlotPanel(String title, String xlabel, String ylabel,
         XYDataset dataset) {
      JFreeChart chart = ChartFactory.createXYLineChart(title, xlabel,
            ylabel, dataset, PlotOrientation.VERTICAL, true, false, false);
      ChartPanel chartPanel = new ChartPanel(chart);

      setLayout(new GridLayout(1, 1));
      add(chartPanel);
   }
}
```

Listing 13: A plot panel

```
import iceb.plotutils.DynamicDataset;

import javax.swing.JFrame;

import org.jfree.data.xy.XYDataset;

public class PlotPanelExample extends JFrame {

   public static void main(String[] args) {
      new PlotPanelExample().setVisible(true);
   }

   public PlotPanelExample() {
      XYDataset dataset = createDataset();
      PlotPanel plotPanel = new PlotPanel("some plots", "x", "y", dataset);

      add(plotPanel);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      pack();
   }

   private XYDataset createDataset() {
      DynamicDataset dataset = new DynamicDataset();
      int n = 500;
      double dx = 2 * Math.PI / (n - 1);

      for (int i = 0; i < n; i++) {
         double x = i * dx;

         dataset.addX(x);
         dataset.addY("sin(x)", Math.sin(x));
         dataset.addY("cos(x)", Math.cos(x));
         dataset.addY("cos(2*x*x)", Math.cos(2 * x * x));
      }
      return dataset;
   }
}
```

Listing 14: Using the plot panel

7 How to define a function

Mathematical functions $f : \mathbb{R}^n \to \mathbb{R}$ such as $f(x, y) = x^2 + y^2$ can easily be defined with the **`UserFunction`** class. Example code for the function

$$\begin{aligned} f_1(x, y) &= x^2 + y^2 \\ f_2(x_1, x_2, x_3) &= x_1 + x_2 + x_3 \\ f_3(x) &= \pi + x; \end{aligned}$$

is given in Listing 15. Note that because the function expressions supplied to the constructors may contain errors, we have to catch a possible exception.

```
import gnu.jel.CompilationException;
import inf.math.UserFunction;

public class UserFunctionExample {
   public static void main(String[] args) {
      try {
         UserFunction f1 = new UserFunction("x*x+y*y", "x", "y");
         UserFunction f2 = new UserFunction("x1 + x2 + x3", "x1", "x2", "x3");
         UserFunction f3 = new UserFunction("PI + x", "x");

         System.out.println("f1(4, 5) = " + f1.valueAt(4, 5));
         System.out.println("f2(1, 2, 3) = " + f2.valueAt(1, 2, 3));
         System.out.println("f3(0) = " + f3.valueAt(0));
      } catch (CompilationException e) {
         e.printStackTrace();
      }
   }
}
```

Listing 15: Defining functions

8 How to create two-dimensional drawings

When it comes to two-dimensional drawing in a Java application, the easiest solution is to create a subclass of **`JPanel`** and to override the `paintComponent` method. The `paintComponent` method is invoked by the Java graphics system whenever the component needs to be drawn on the screen. Components can be drawn either by using the **`Graphics`** class or the **`Graphics2D`** class which extends **`Graphics`** and adds many features useful for creating fancy figures.

8.1 Using the `Graphics` class

The `paintComponent` method has an argument of type **`Graphics`**. The **`Graphics`** class has a set of methods that are used to generate graphical entities. Examples of such methods are `drawLine`, `drawOval` and so on; consider the Javadoc for more information. All these drawing-methods take coordinates as input. These coordinates are specified in pixels whereas the origin of the coordinate system is in the upper left corner of the panel (see Figure 8).

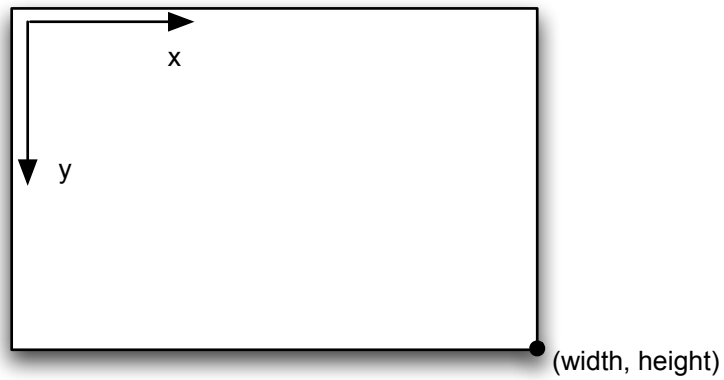


Figure 8: Coordinate system of a Swing component

In addition to the drawing-methods, there are methods that affect the style of subsequent drawing commands. Using the `setColor` method, the color of objects is changed. The `setFont` method is used to specify the font for rendering text.

Listing 16 shows such a class that extends **`Panel`** and overrides `paintComponent`. Such custom components can now be added to a frame or any other container. Listing 17 demonstrates this for a simple application. The running application in two different window sizes is shown in Figure 9.

```
import java.awt.*;

import javax.swing.JPanel;

public class GraphicsPanel extends JPanel {
   protected void paintComponent(Graphics g) {
      Dimension d = getSize();

      g.setColor(Color.GREEN);
      g.drawLine(0, 0, d.width, d.height);
      g.drawLine(d.width, 0, 0, d.height);
      g.setColor(Color.BLUE);
      g.fillOval(d.width / 2 - 15, d.height / 2 - 75, 30, 150);
      g.setColor(Color.BLACK);
      g.drawString("Hello World", d.width / 4, d.height / 4);
   }
}
```

Listing 16: Graphics panel. The `getSize` method returns the size of the component in pixels

```
import javax.swing.*;

public class GraphicsPanelApplication extends JFrame {
   public static void main(String[] args) {
      new GraphicsPanelApplication().setVisible(true);
   }
   public GraphicsPanelApplication() {
      add(new GraphicsPanel());
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setSize(600, 350);
   }
}
```

Listing 17: Using the **`GraphicsPanel`** class in a Frame

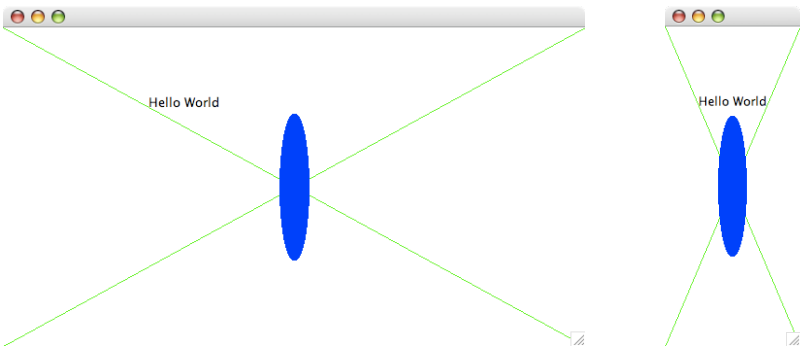


Figure 9: The application on MacOS. The drawing adjusts itself to the size of the panel.

8.2 Using the `Graphics2D` class

The **`Graphics2D`** class adds to the **`Graphics`** class many features. It can render various types of graphics objects like lines, polygons, or circles. In addition, it provides methods to perform affine transformations that can be used to adjust drawing size. In many situations, the **`Graphics2D`** option is the better choice.

8.2.1 Drawing shapes

While the **`Graphics`** class defines many drawing and filling methods each for a specific shape, the **`Graphics2D`** class only uses two such methods. In order to paint objects, the methods

```
public void draw(Shape s)
```

which paints the outline of the specified shape and

```
public void fill(Shape s)
```

which fills the shape with the current color. The argument type **`Shape`** is a Java interface which is implemented by many classes. Examples include **`Line2D`**, **`Polygon`**, **`Rectangle2D`**, **`RoundRectangle`** etc. Consider the example in Listing 18 which demonstrates this approach. This simple example uses one single **`Polygon`** object which is initialized in the constructor. In the `paintComponent` method, the polygon is filled with blue color and gets an outline of three pixels width in red.

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

public class Graphics2DPanel extends JPanel {

   private java.awt.Polygon polygon_ = new java.awt.Polygon();

   public Graphics2DPanel() {
      polygon_.addPoint(0, 100);
      polygon_.addPoint(100, 300);
      polygon_.addPoint(200, 200);
      polygon_.addPoint(300, 300);
      polygon_.addPoint(400, 100);
   }

   protected void paintComponent(Graphics g) {
      Graphics2D g2d = (Graphics2D) g;
      g2d.setColor(Color.BLUE);
      g2d.fill(polygon_);
      g2d.setStroke(new BasicStroke(3));
      g2d.setColor(Color.RED);
      g2d.draw(polygon_);
   }
}
```

Listing 18: Graphics 2D panel

The screen-shot corresponding to listing 18 is shown in Figure 10. If you run the application, you can see that the figure always draws at the same size no matter what size the panel is.

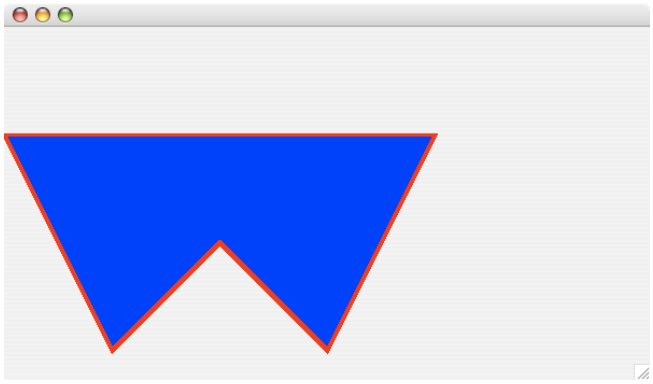


Figure 10: The graphics 2D application

8.2.2 World and view coordinate systems

In mostly every graphical application, you have to deal with two different coordinate systems: Your model is defined in a certain coordinate system where one unit corresponds to some physical size (e.g. one meter). On the other hand, the drawing is performed using screen coordinates which are in pixels. For instance, if your model is of size 3×2 (in meters) and you want to draw it in a panel which is 400×300 pixels, you have to do some scaling. Usually, your model coordinate system is called world coordinate system and the coordinate system of your panel is called view coordinate system. Figure 11 shows these two coordinated systems. Consider point A: Obviously it has different coordinates with respect to both coordinate systems.

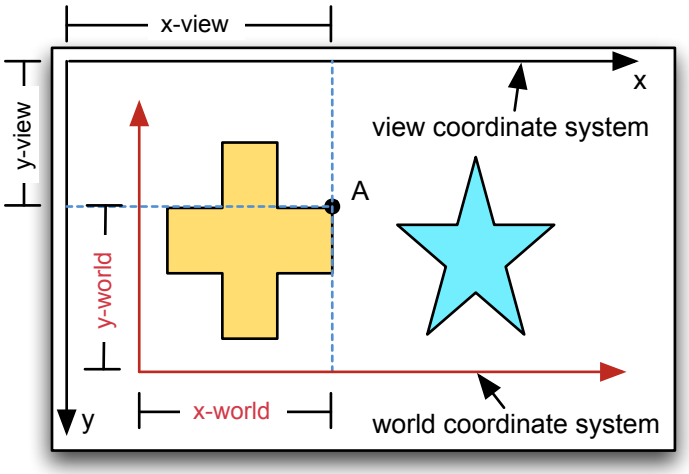


Figure 11: Objects in view coordinate system and in world coordinate system

The **`Graphics2D`** class has methods that allow us to set up a coordinate transformation such that the shapes to be drawn can be specified in world coordinates. Namely, it provides the methods

scale(double sx, double sy)

that adds a scaling factor to the subsequent drawing commands and

translate(double tx, double ty)

that translates the origin. Using the `scale` method first and then the `translate` method, we set up a function that gives the view coordinates in terms of world coordinates according to

$$\begin{aligned} x_v &= s_x(x_w + t_x) \\ y_v &= s_y(y_w + t_y) \end{aligned}$$

where the translation is specified in world coordinates. Knowing this relation, we can set up a coordinate transformation that maps the world window into the center of the so called viewport and adjust the scaling accordingly. Figure 12 illustrates the mapping of the world window into the viewport.

After some thinking (and some experiments) we can figure out that the code in Listing 19 performs the transformation that we want. The method takes as arguments the specification of the world window and the **`Graphics2D`** object on which we want to paint.

If we add the `setUpTransform` to the **`Graphics2DPanel`** class in Listing 18 and introduce the statement

setUpTransform(0, 100, 400, 300, g2d);

to the `paintComponent` method, we get the desired behavior as illustrated in Figure 13. Note that the arguments to the method call are the bounding box of the polygon defined in the constructor. Now, whenever the panel is drawn, first the transformation is set up correctly.

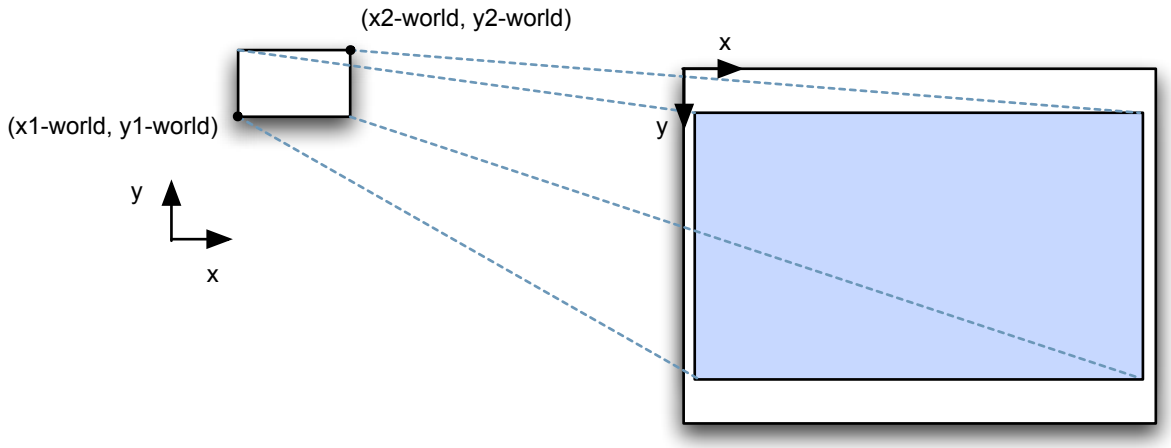


Figure 12: Mapping of world-window into viewport

```
private void setUpTransform(double wx1, double wy1,
                    double wx2, double wy2, Graphics2D g2d) {
  Dimension vSize = getSize();
  int vWidth = vSize.width;
  int vHeight = vSize.height;
  double vCenterX = 0.5 * vWidth;
  double vCenterY = 0.5 * vHeight;
  double wWidth = wx2 - wx1;
  double wHeight = wy2 - wy1;
  double wCenterX = wx1 + 0.5 * wWidth;
  double wCenterY = wy1 + 0.5 * wHeight;
  double scale = 0.9 * Math.min(vWidth / wWidth, vHeight / wHeight);
  double wTranX = vCenterX / scale - wCenterX;
  double wTranY = -vCenterY / scale - wCenterY;

  g2d.scale(scale, -scale);
  g2d.translate(wTranX, wTranY);
}
```

Listing 19: Set up transformation

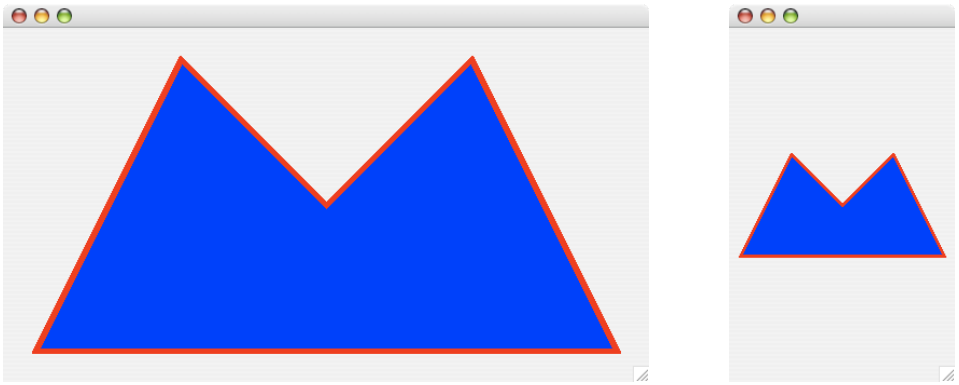


Figure 13: The scaled graphics application

9 Managing interactive applications

Developing non-trivial applications with a graphical user interface (GUI) is difficult. Without using some sort of an architecture which organizes the responsibilities of individual classes, the code of such applications easily becomes very hard to understand and maintain. In this chapter, a simplified version of the well known Model-View-Controller architectural design pattern is presented which can be used as a template for new applications.

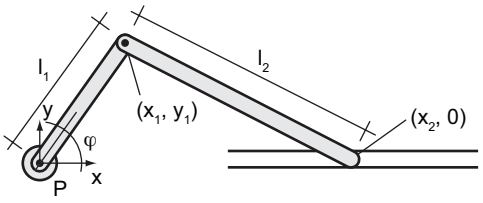


Figure 14: Mechanism

The graphical application to be developed deals with the mechanismas as shown in Figure 14. A mechanism consists of two tubes lying in the xy-plane which are connected by a hinge. The first tube (length l_1) rotates at one end around the z-axis. The other end of the first tube is connected to a second tube (length l_2) which is fixed vertically at its other end.

This section is split in two parts: In the first part of this section, the application is created such that mechanisms of different dimensions can be created and rotated stepwise. In the second part, animation is added.

9.1 Part 1: The bare application

The application to be created in Part 1 is shown in Figure 15. On the left hand side, there are graphical components which allow users to create a new mechanism of the specified dimensions or to rotate an existing mechanism. On the right hand side, the current state of the mechanism is displayed graphically as well as numerically. Whenever users click a button, the state of the mechanism is updated.

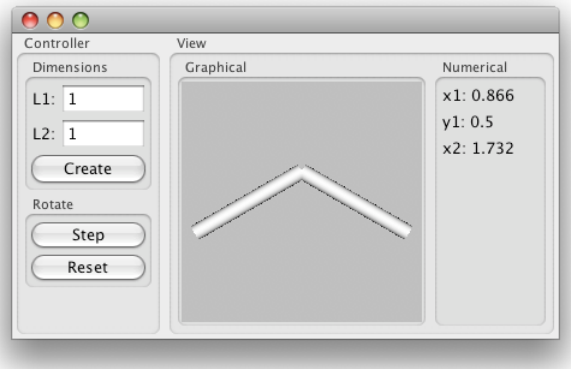


Figure 15: The bare application

Obviously the application consists of a part which controls the mechanism and another part which lets users view properties of the mechanism. This observation leads to the software design shown in Figure 16.

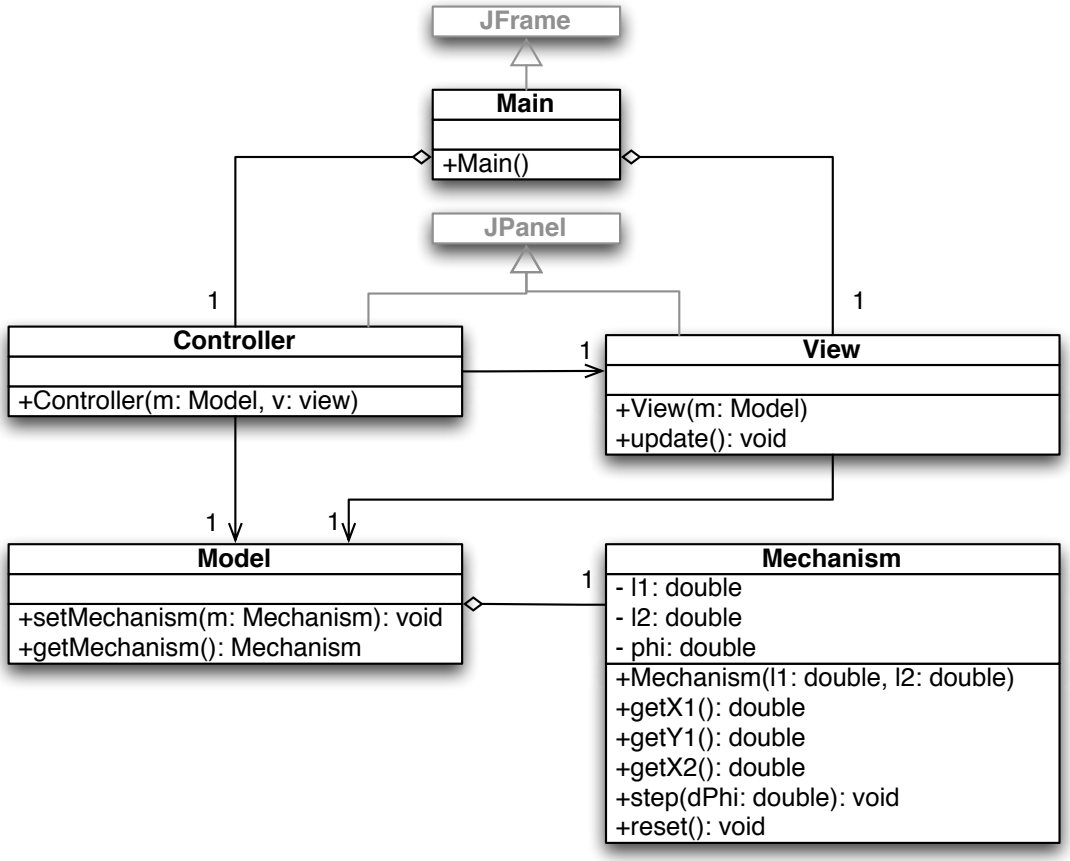


Figure 16: UML class diagram of the mechanism-application

`Mechanism` This class represents the mechanism. It has attributes for the dimensions l_1, l_2 and the current angle φ. Several methods make it possible to get information about the mechanism and to let the mechanism rotate.

`Model` This class is only a container for the current mechanism and has very limited functionality.

`View` This class, which is derived from **`JPanel`**, provides the graphical and textual representation of the current mechanism. It is associated with the **`Model`** class. Most important, it has an `update` method which updates the graphical information displayed according to the mechanism currently associated with the model.

`Controller` This class (again derived from **`JPanel`**) receives input from the user. It has associations to the **`Model`** class and the **`View`** class.

`Main` The application class is derived from **`JFrame`** and consists of a **`Controller`** and a **`View`**.

9.1.1 Basic classes

The basic classes **`Mechanism`** and **`Model`** are straightforward and do not require further explanation.

```
public class Mechanism {

   private double l1_;
   private double l2_;
   private double phi_;

   public Mechanism(double l1, double l2) {
      l1_ = l1;
      l2_ = l2;
      reset();
   }

   public double getX1() {
      return l1_ * Math.cos(phi_);
   }

   public double getY1() {
      return l1_ * Math.sin(phi_);
   }

   public double getX2() {
      return getX1() + Math.sqrt(Math.pow(l2_, 2) - Math.pow(getY1(), 2));
   }

   public void reset() {
      phi_ = Math.toRadians(30);
   }

   public void step(double dphi) {
      phi_ += dphi;
   }
}
```

Listing 20: The **`Mechanism`** class

```
public class Model {

   private Mechanism mechanism_ = new Mechanism(1, 1);

   public Mechanism getMechanism() {
      return mechanism_;
   }

   public void setMechanism(Mechanism m) {
      mechanism_ = m;
   }
}
```

Listing 21: The **`Model`** class

9.1.2 The application framework

The purpose of this step is to establish the application framework. The application after completion of this development step is shown in Figure 17. The application frame contains two areas, one for the controller and one for the view.

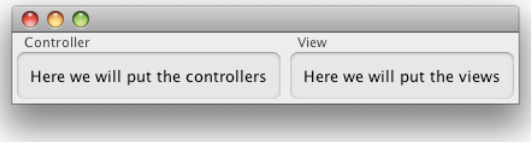


Figure 17: The application

The corresponding **`View`** and **`Controller`** classes are given in the listings below: In to add a border to the panels, the **`BorderFactory`** class is used. A **`JLabel`** is placed inside the panels as a placeholder for the actual graphical components which will be added later.

```
public class View extends JPanel {

   public View(Model m) {
      setBorder(BorderFactory.createTitledBorder("View"));
      add(new JLabel("Here we will put the views"));
   }
}
```

Listing 22: The empty **`View`** class

```
public class Controller extends JPanel {

   public Controller(Model model, View view) {
      setBorder(BorderFactory.createTitledBorder("Controller"));
      add(new JLabel("Here we will put the controllers"));
   }
}
```

Listing 23: The empty **`Controller`** class

In the constructor of the **main** class, the **Model**, **View** and **Controller** objects are created. Then the view and the controller are added to the frame (note that the **JFrame** class uses a **BorderLayout** by default). Finally, the default close operation is specified and the components inside the frame are arranged using the `pack` method.

```
public class Main extends JFrame {

   public static void main(String[] args) {
      new Main().setVisible(true);
   }

   public Main() {
      Model m = new Model();
      View v = new View(m);
      Controller c = new Controller(m, v);

      add(c, BorderLayout.WEST);
      add(v, BorderLayout.CENTER);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      pack();
   }
}
```

Listing 24: The **Main** class

9.2 The view

Now, the view on the mechanism is added to the application. Figure 18 shows the application after completion of this development step. The view is composed of two individual views: A graphical view and a numerical view which renders the state variables of the mechanism numerically.

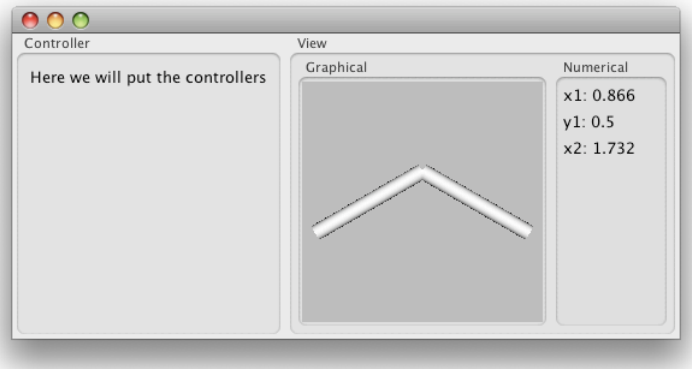


Figure 18: The application

Figure 19 shows the detailed UML class diagram of the view classes. Corresponding to the application in Figure 18, a **View** object consists of two individual views: The **GraphicalV** and the **NumericalV**. Note that attributes and associations are not displayed in the diagram.

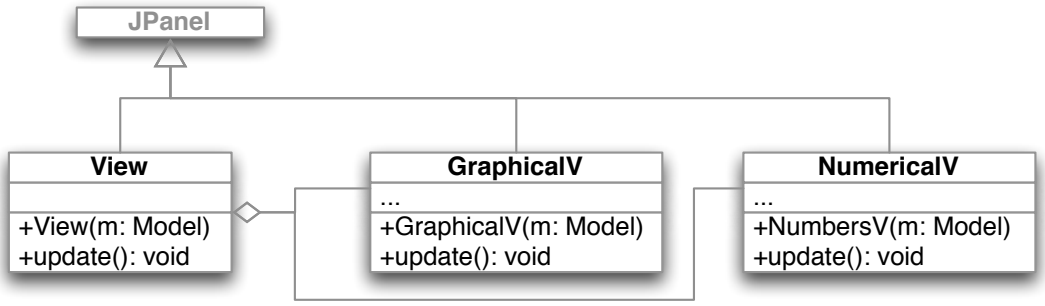


Figure 19: Detailed UML class diagram of the view classes

Two cylinders are used to visualize the mechanism in the **`GraphicalV`** class. The cylinders are displayed in the **`ViewerPanel`**. In the constructor, the association to the **`Model`** is established and the **`ViewerPanel`** is added. The purpose of the `update` method is to let the **`ViewerPanel`** display the current state of the mechanism. For that, first the values x_1, y_1, x_2 are obtained from the mechanism, then the cylinders are modified according to these values and finally, the **`ViewerPanel`** is rendered such that the new mechanism state is displayed.

```
public class GraphicalV extends JPanel {

   private Model model_;
   private Cylinder c1_ = new Cylinder(0, 0, 0, 0, 0, 0);
   private Cylinder c2_ = new Cylinder(0, 0, 0, 0, 0, 0);
   private ViewerPanel viewerPanel_ = new ViewerPanel();

   public GraphicalV(Model m) {
      model_ = m;
      setBorder(BorderFactory.createTitledBorder("Graphical"));
      setLayout(new GridLayout(1, 1));
      add(viewerPanel_);
   }

   public void update() {
      Mechanism m = model_.getMechanism();
      double x1 = m.getX1();
      double y1 = m.getY1();
      double x2 = m.getX2();

      c1_.setRadius(0.065);
      c1_.setPoint2(x1, y1, 0);
      c2_.setRadius(0.065);
      c2_.setPoint1(x1, y1, 0);
      c2_.setPoint2(x2, 0, 0);
      viewerPanel_.render();
   }
}
```

Listing 25: The graphical view

In the numerical view, the the values x_1, y_1, x_2 are be displayed numerically in three rows in the upper part of the **NumericalV** panel. Three **JLabel** objects are used for this purpose (the attributes `x1L_`, `y1L_` and `x2L_` point to these objects). In order to achieve the desired graphical arrangement of the labels, an additional **JPanel** with a **Gridlayout** is created in the constructor of the **NumericalV** class (the local variable `p` points to this panel). The labels are added to this panel which itself is then added to the north-compartment of the **NumericalV** panel which has a **BorderLayout**. In order to give a fixed width to the **NumericalV** panel, the preferred size is specified using the `setPreferredSize` method. Since only the width is interesting at this point, the value 0 is specified for the height. Finally, the connection to the **Model** is established.

The content of the labels is updated in the `update` method corresponding to the current values of the mechanism. A **NumberFormat** object is used in order to convert the numbers nicely into strings.

```
public class NumericalV extends JPanel {

   private Model model_;
   private JLabel x1L_ = new JLabel();
   private JLabel y1L_ = new JLabel();
   private JLabel x2L_ = new JLabel();

   public NumericalV(Model m) {
      JPanel p = new JPanel(new GridLayout(3, 1));

      p.add(x1L_);
      p.add(y1L_);
      p.add(x2L_);

      setBorder(BorderFactory.createTitledBorder("Numerical"));
      setLayout(new BorderLayout());
      add(p, BorderLayout.NORTH);

      setPreferredSize(new Dimension(100, 0));
      model_ = m;
   }

   public void update() {
      NumberFormat nf = NumberFormat.getNumberInstance();
      Mechanism m = model_.getMechanism();
      double x1 = m.getX1();
      double y1 = m.getY1();
      double x2 = m.getX2();

      x1L_.setText("x1: " + nf.format(x1));
      y1L_.setText("y1: " + nf.format(y1));
      x2L_.setText("x2: " + nf.format(x2));
   }
}
```

Listing 26: The numerical view

In order to incorporate the new view classes, the **View** class has to be extended. Two attributes of type **GraphicalV** and **NumericalV** are introduced to implement the associations shown in the class diagram in Figure 19. In the constructor, the corresponding objects are created and added The last statement in the constructor is a call to the new `update`-method in which the `update`-methods of the associated **GraphicalV** and **NumericalV** objects are invoked.

```
public class View extends JPanel {

   private GraphicalV graphicalV_;
   private NumericalV numbersV_;

   public View(Model m) {
      graphicalV_ = new GraphicalV(m);
      numbersV_ = new NumericalV(m);
      setLayout(new BorderLayout());
      setBorder(BorderFactory.createTitledBorder("View"));
      add(graphicalV_, BorderLayout.CENTER);
      add(numbersV_, BorderLayout.EAST);
      update();
   }

   public void update() {
      graphicalV_.update();
      numbersV_.update();
   }
}
```

Listing 27: The extended **View** class

9.3 The controller

Finally, the controller is added such that after completion of this development step, the applications looks as shown in Figure 15. There, it can be seen that the controller consists of two parts: One for the dimensions of the mechanism and one to rotate the mechanism (see Figure 20).

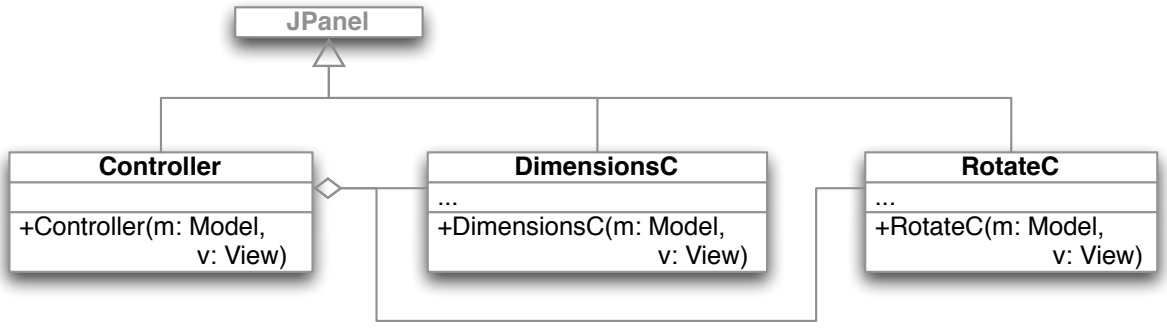


Figure 20: Detailed UML class diagram of the controller classes

Two **JTextField** objects are used in the **DimensionsC** class to take input from the user. The class implements the **ActionListener** interface and is registered as

a listener at the apply Button. The `addTextField` method is a utility method which just adds a text field along with a label.

Inside the `actionPerformed` method, the numerical values are extracted from the text fields and checked for consistency. If the values are valid, a new **`Mechanism`** is created and passed to the associated **`Model`** object. The the **`View`** is updated such that the new mechanism becomes visible. If the values are not valid, a dialog is shown up and no new mechanism is constructed.

```
public class DimensionsC extends JPanel implements ActionListener {

   private Model model_;
   private View view_;

   private JTextField l1TF_ = new JTextField("1", 5);
   private JTextField l2TF_ = new JTextField("1", 5);
   private JButton createB_ = new JButton("Create");

   public DimensionsC(Model model, View view) {
      model_ = model;
      view_ = view;
      createB_.addActionListener(this);

      setLayout(new GridLayout(3, 1));
      setBorder(BorderFactory.createTitledBorder("Dimensions"));

      addTextField("L1:", l1TF_);
      addTextField("L2:", l2TF_);
      add(createB_);
   }

   public void actionPerformed(ActionEvent e) {
      double l1 = Double.parseDouble(l1TF_.getText());
      double l2 = Double.parseDouble(l2TF_.getText());

      if (l1 <= l2) {
         model_.setMechanism(new Mechanism(l1, l2));
         view_.update();
      } else {
         JOptionPane.showMessageDialog(this, "l1 > l2", "Illegal Input",
               JOptionPane.ERROR_MESSAGE);
      }
   }

   private void addTextField(String l, JTextField tf) {
      JPanel p = new JPanel(new BorderLayout());

      p.add(new JLabel(l), BorderLayout.WEST);
      p.add(tf, BorderLayout.CENTER);
      add(p);
   }
}
```

Listing 28: The dimensions controller

In the **RotateC** class, two buttons are used to control the state of the mechanism: One to reset the mechanism in its initial position and on to advance the rotation. Therefore, in the `actionPerformed` method, a distinction regarding the button which generated the event has to be made: In one case, the rotation of the mechanism is advanced, in the other case, the mechanism is reset. Finally, the view is updated such that the new state becomes visible.

```
public class RotateC extends JPanel implements ActionListener {

   private View view_;
   private Model model_;
   private JButton stepB_ = new JButton(”Step”);
   private JButton resetB_ = new JButton(”Reset”);

   public RotateC(Model model, View view) {
      model_ = model;
      view_ = view;
      stepB_.addActionListener(this);
      resetB_.addActionListener(this);
      setLayout(new GridLayout(2, 1));
      setBorder(BorderFactory.createTitledBorder(”Rotate”));
      add(stepB_);
      add(resetB_);
   }

   public void actionPerformed(ActionEvent e) {
      if (e.getSource() == stepB_) {
         model_.getMechanism().step(2 * Math.PI / 30);
      } else if (e.getSource() == resetB_) {
         model_.getMechanism().reset();
      }
      view_.update();
   }
}
```

Listing 29: The rotate controller

Finally, the **Controller** class has to be extended in order to incorporate the two new controllers. In order to achieve the desired component arrangement, an additional **JPanel** has to be used.

```
public class Controller extends JPanel {

   public Controller(Model model, View view) {
      JPanel p = new JPanel(new BorderLayout());
      p.add(new DimensionsC(model, view), BorderLayout.NORTH);
      p.add(new RotateC(model, view), BorderLayout.CENTER);
      setBorder(BorderFactory.createTitledBorder(”Controller”));
      setLayout(new BorderLayout());
      add(p, BorderLayout.NORTH);
   }
}
```

Listing 30: The extended **Controller** class

10 Part 2: Adding animation

Figure 21 shows the application after the animation feature has been added. In the controller, there are new buttons to start, stop and reset the animation and a new view shows the time history of the mechanism state variables.

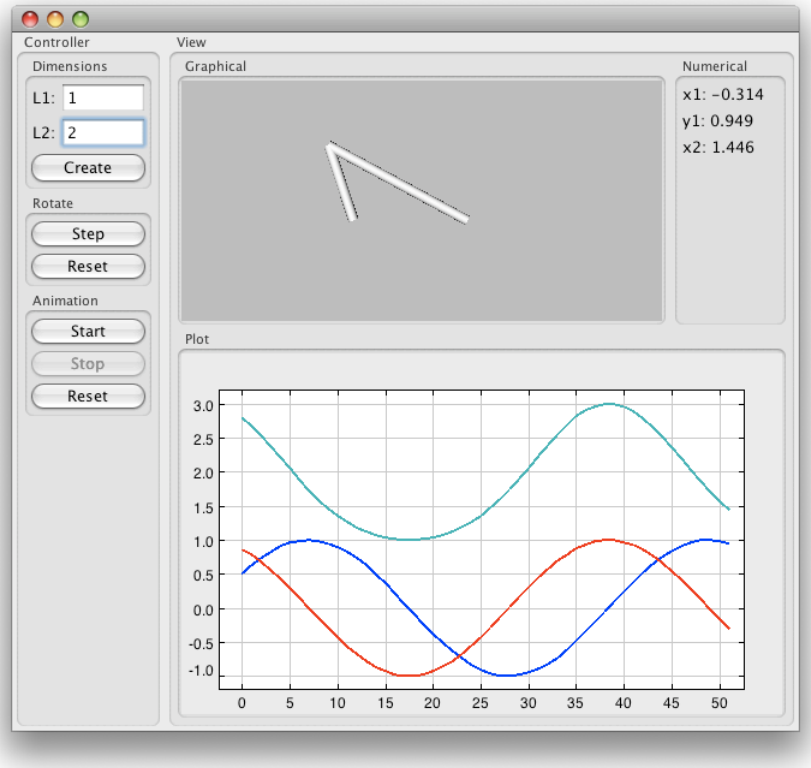


Figure 21: The application with animation

10.1 One more view

One more specific view has to be added to the **`View`** class in order to visualize the time history of the state variables. The **`Plot`** class from the `ptolemy.plot` package is used for this purpose. Since the plot view accumulates information while the animation is running, a `reset` method has to be introduced which is used to clear the plot when the user clicks the reset button.

```
public class PlotV extends JPanel {

   private int step_ = 0;
   private Model model_ = new Model();
   private Plot plot_ = new Plot();

   public PlotV(Model model) {
      model_ = model;

      setBorder(BorderFactory.createTitledBorder("Plot"));
      setLayout(new GridLayout(1, 1));
      add(plot_);
   }

   public void update() {
      Mechanism m = model_.getMechanism();
      double x1 = m.getX1();
      double y1 = m.getY1();
      double y2 = m.getX2();

      plot_.addPoint(0, step_, x1, true);
      plot_.addPoint(1, step_, y1, true);
      plot_.addPoint(2, step_, y2, true);
      plot_.repaint();
      step_++;
   }

   public void reset() {
      step_ = 0;
      plot_.clear(true);
   }
}
```

Listing 31: The plot view

In the updated view class, the **PlotV** is incorporated and a new `reset` method is introduced. Inside the `reset` method, the plot view is reset and then the whole view is updated.

```
public class View extends JPanel {

   private GraphicalV graphicalV_;
   private NumericalV numbersV_;
   private PlotV plotV_;

   public View(Model m) {
      graphicalV_ = new GraphicalV(m);
      numbersV_ = new NumericalV(m);
      plotV_ = new PlotV(m);
      setLayout(new BorderLayout());
      setBorder(BorderFactory.createTitledBorder("View"));
      add(graphicalV_, BorderLayout.CENTER);
      add(numbersV_, BorderLayout.EAST);
      add(plotV_, BorderLayout.SOUTH);
      update();
   }

   public void update() {
      graphicalV_.update();
      numbersV_.update();
      plotV_.update();
   }

   public void reset() {
      plotV_.reset();
      update();
   }
}
```

Listing 32: The updated view class

10.2 The animation thread

Whenever an application performs a long running task such as showing an animation, there is an issue which requires special care: Threads. Explaining threads in detail is impossible at this place but some background information should be provided.

As a basis for the discussion of threads, we consider the desired behavior of the application. It is as follows:

- At the beginning, the user clicks the start button and the animation begins.
- After some time, the user stops the animation by clicking the stop button and the animation stops.

Without using threads, the application will not behave like that, because after clicking the start button, the program is busy showing the animation and can not receive further input from the user. Therefore, the animation has to be started in its own

thread. A thread can be understood as an independent sequence of parallel task execution. Similar to people who can do a phone call and write email at the same time, the computer can carry out different tasks in different threads.

The first step in programming with threads is to subclass **`Thread`**, in this case the class is called **`AnimationThread`**. The class has associations to a **`Model`** and a **`View`** and a boolean attribute `stopped_` which is used to indicate that the animation should be stopped. Inside the `run` method, which is executed when the thread is started, there is a loop which is repeated as long as the `stopped_` attribute is not set to `true` using the `stopThread()` method. Inside the loop, the mechanism is rotated a little bit, the view is updated and then, the thread is halted for 30 milliseconds (here we have to catch a possible exception but this is just a technical detail). The thread is halted because otherwise the animation would run too fast.

The **`AnimationThread`** class can be used as it is for your own application. The only thing which has to be replaced is the statement `m.step(30)` which is specific to the mechanism application.

```
public class AnimationThread extends Thread {

   private Model model_;
   private View view_;

   private boolean stopped_ = false;

   public AnimationThread(Model model, View view) {
      model_ = model;
      view_ = view;
   }

   public void run() {
      Mechanism m = model_.getMechanism();

      while (!stopped_) {
         m.step(0.15);
         view_.update();

         try {
            sleep(30);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

   public void stopThread() {
      stopped_ = true;
   }
}
```

Listing 33: The animation thread

10.3 One more controller

The **AnimationC** controller is associated with a **Model**, a **View**, an **AnimationThread** and three individual buttons. When the start button is clicked, a new **AnimationThread** is created and started. When the stop button is clicked, the animation thread is stopped. When the reset button is clicked, the mechanism and the view are reset. The `setRunning` method updates the state of the buttons. For instance, the start button should not be clickable when an animation is already running.

```
public class AnimationC extends JPanel implements ActionListener {

   private Model model_;
   private View view_;
   private AnimationThread at_;
   private JButton startB_ = new JButton("Start");
   private JButton stopB_ = new JButton("Stop");
   private JButton resetB_ = new JButton("Reset");

   public AnimationC(Model m, View v) {
      model_ = m;
      view_ = v;
      setLayout(new GridLayout(3, 1));
      setBorder(BorderFactory.createTitledBorder("Animation"));
      add(startB_);
      add(stopB_);
      add(resetB_);
      setRunning(false);
      startB_.addActionListener(this);
      stopB_.addActionListener(this);
      resetB_.addActionListener(this);
   }

   public void actionPerformed(ActionEvent e) {
      if (e.getSource() == startB_) {
         at_ = new AnimationThread(model_, view_);
         at_.start();
         setRunning(true);
      } else if (e.getSource() == stopB_) {
         at_.stopThread();
         setRunning(false);
      } else if (e.getSource() == resetB_) {
         model_.getMechanism().reset();
         view_.reset();
      }
   }

   private void setRunning(boolean b) {
      startB_.setEnabled(!b);
      stopB_.setEnabled(b);
      resetB_.setEnabled(!b);
   }
}
```

Listing 34: The animation controller

The new animation controller is added to the controller using one more call to `add` in the constructor.

```
public class Controller extends JPanel {

   public Controller(Model model, View view) {
      JPanel p = new JPanel(new BorderLayout());
      p.add(new DimensionsC(model, view), BorderLayout.NORTH);
      p.add(new RotateC(model, view), BorderLayout.CENTER);
      p.add(new AnimationC(model, view), BorderLayout.SOUTH);
      setBorder(BorderFactory.createTitledBorder("Controller"));
      setLayout(new BorderLayout());
      add(p, BorderLayout.NORTH);
   }
}
```

Listing 35: The updated controller class

11 Using the code from this section

The main purpose of this section was to introduce a reusable design pattern for an architecture which is suitable for moderately complex applications. If you want to use this pattern for your own program, the following steps might serve you as a guideline:

1. Implement and test the class representing the type of objects your application is concerned with. In this example, this was the mechanism class.
2. Create the model class.
3. Establish the application framework. You can copy the classes from Section 9.1.2 since they do not contain any domain specific code.
4. Implement your own specialized view classes and add them to the view.
5. Develop your own specialized controller classes and add them to the controller.

Programming style You might have noticed that in this application, a relatively large number of classes have been introduced which might seem overly complicated. And of course, it would have been possible to implement the whole application in a single class. However, there are good reasons to use some more small and simple classes instead of on big and complicated class:

- Every class has a single and well defined purpose.
- The individual classes are short and easy to understand.
- Applications are easy to extend and additional functionality can be added with little hassle (as the animation feature).