CSCI E-10b – Term Project Write-Up: Knitting Pattern Editor

Johanna Bodnyk | May 9, 2014

Summary and Functionality

My term project is a program to render and edit knitting patterns. There are two types of knitting patterns: written instructions consisting of standardized abbreviations that describe the stitches to be knit, row by row; and charts that use standardized symbols, arranged in a grid, to indicate the arrangement of stitches.

The program I have written:

- reads a set of written knitting instructions from text file, parses the abbreviations, and renders the corresponding knitting chart;
- allows a user to edit the instructions in a GUI and re-render the corresponding chart (or write instructions from scratch in the GUI and render the chart);
- allow a user to save the edited (or newly created) instructions, as a new file, or overwriting the original file.
- provides error messaging if a pattern cannot be parsed, and a help pane describing the pattern syntax rules and permissible abbreviations.

Rationale

As explained in this article, knitting charts can be more helpful than, or a useful accompaniment to, written instructions. The rationale for this program is to allow someone using a pattern that only has written instructions to generate a chart, and to allow pattern authors to generate both instructions and a chart at the same time.

Instructions

To run the Knitting Pattern Editor program:

1. Type "java KnittingEditor" at the command line.
2. Draft a knitting pattern directly in the Knitting Instructions pane (click ***Help*** to view a list of recognized abbrevations and syntax rules).

   OR

   Click ***Import*** to import a pattern from a text file (e.g. samplepattern.txt)

3. After drafting pattern instructions or editing imported instructions, click *Render* to view the corresponding knitting chart in the Knitting Chart pane.

Classes and Files

The program consists of the following classes and files:

KnittingEditor.java

This class extends JFrame to set up the program's GUI. It also includes helper methods called by the action listeners attached to the GUI's buttons—to import a file, save or save as, render a pattern into a knitting chart (by creating an Instructions object), or display a help window.

Instructions.java

This class is responsible for parsing the user-supplied knitting pattern text instructions into a 2D array of Stitch enum values, to be used by the main KnittingEditor program to render the corresponding knitting chart.

Stitch.java

This enum class defines the stitch abbrevations allowed by the program, and maps them to their corresponding chart symbols and prose stitch names.

samplepattern.txt

This text file contains a sample set of knitting pattern instructions containing the allowed stitch types and syntax variations, for testing purposes.

Program Description

Program Flow

1. **Getting User Input (Knitting Pattern)**
   When the program starts, a welcome dialog explains that a knitting pattern may be entered in one of two ways: by clicking the ***Import*** button to open a text file from the user's computer or by typing the pattern directly into the Instructions Pane.
   a. **Import Button**: When the user clicks the ***Import*** button, its action listener first checks to see whether the Instructions Pane currently contains any text. If so, a JOptionPane is launched asking the user whether they want save their work first, or proceed without saving. If the user chooses to save first, the ***saveAs()*** method is called, followed by ***importFile().*** If not, or if the Instructions Pane is empty, ***importFile()*** is called directly.
   b. **importFile():** This method launches a JFileChooser open dialog allowing the user to navigate to the file they want to open. If the file cannot be opened, or if it contains no readable text, an error message is displayed in a JOptionPane message dialog. If it contains readable text, its contents are read line by line and saved in a String array (one line per array item). The method then uses that array as an argument in calling both the ***updateInstructionPane()*** method to display the imported text in the instructions pane, where it may be edited by the user, and the ***renderChart()*** method to attempt to convert the text pattern to a chart.
2. **Rendering the Knitting Chart**
   The ***renderChart()*** method may be called two ways—from the ***importFile()*** method as described above, or when the user clicks the ***Render*** button. The user may click the ***Render*** button multiple times in the course of editing a pattern to see what the resulting chart looks like.
   a. **Render Button:** When the user clicks the ***Render*** button, its action listener starts by calling ***parseInstructionsPane(),*** which returns an array containing the lines of text currently in the Instructions Pane. This

may be text the user has typed from scratch, or text from an imported file that the user has subsequently edited in the GUI. The method then uses that array as an argument in calling the ***renderChart()*** method to attempt to convert the text to a chart.

b. **renderChart():** This method begins by instantiating a new object of the Instructions class, using the user-supplied knitting pattern text as an argument. As described in more detail below, the Instructions class's constructor parses the text pattern and generates a 2D array of corresponding stitch symbols (represented by Stitch enum values), which is retrieved by calling the Instructions object's ***getStitches()*** method. After clearing the current contents of the Chart Pane, the method sets a new GridLayout sized to match the 2D array, then loops[1] through the array to add JLabels containing the stitch symbols to the grid. This forms the knitting chart. Each chart square also has a tooltip containing the prose name of the stitch, for reference.

3. **Saving a Pattern**

   The text currently displayed in the Knitting Instructions pane may be saved by clicking either the ***Save*** button or the ***Save as*** button.

   a. **Save Button:** If the pattern currently displayed in the GUI is the result of a file import, or if the pattern was previously saved, the program tracks that previously used File object in an instance variable. When the user clicks the ***Save*** button, its action listener first checks to see whether that instance variable is set. If so, it calls the ***save()*** method using that instance variable as an argument. If not, it calls the ***saveAs()*** method to get a File object to save to.

   b. **Save as Button:** When the user clicks the ***Save*** button, its action listener calls the ***saveAs()*** method.

   c. **saveAs()**: This method is called when the user clicks the ***Save as*** button, by the ***Save*** button's listener if no File object to save to exists, and by the

[1] While knitting patterns are read like English prose (top to bottom, left to right), knitting charts are read from bottom to top, and rows are read in alternate directions – first right to left, then left to right. The outer loop therefore counts down through the dimension of the array, while the inner loop counts up or down depending on whether the current outer element's index is even or odd.

Import button's listener if the user indicates they want to save their current work before importing. The method launches a JFileChooser save dialog allowing the user choose the new or existing file name and location they want to save to. The method then calls the ***save()*** method using the resulting File object as an argument.

d. **save():** This method retrieves the contents of the Instructions Pane and writes it to the File object that was passed to it (which is either the File object previously imported from or saved to, or a new file chosen by the user through the ***saveAs()*** method).

4. **Displaying Help**
    a. **Help Button:** The Help Pane is a separate JFrame which is hidden by default when the program starts. When the ***Help*** button is clicked, its action listener sets the JFrame's visibility to true. Closing the panel makes it invisible again.

Pattern Parsing

The syntax rules and allowed abbrevations, as explained in the Help Pane, are as follows:

- All rows must contain the same number of stitches
- Each abbreviation should be followed by a comma and one space

  E.g. `k, p, ktog`
- Multiple knit or purl stitches may be denoted by K or P followed by a number

  E.g. `k6, p2`
- Repeated sequences may be enclosed by parentheses, brackets, or asterisks, followed by an indication of the number of times the sequence is to be repeated. The preceding abbreviation must be followed by a comma.

  E.g. `k2, (yo, k2tog) 3 times, k2`

- The following (case insensitive) abbreviations are recognized:

STITCH NAME	ABBREVIATION	CHART SYMBOL
knit	k	[blank]
purl	p	*
yarn over	yo	O
knit 2 together	k2tog	/
slip, knit, pass	skp	\\

A sample pattern, therefore, may look like:

```
k3, (k2tog, yo) 2 times, k4
k2, p7, k2
k3, k2tog, yo, k1, yo, skp, k3
k2, p7, k2
k3, (skp, yo) 2 times, k4
k2, p7, k2
```

The Instructions class's constructor calls ***parseTextInstructions(),*** which itself calls a number of helper methods in sequence in order to parse the knitting pattern text supplied by the user into an array of Stitch enum values to that can be rendered into a knitting chart:

1. **processRepeats():** Beginning with the array of Strings representing the lines (pattern rows) from the user-supplied text, this method looks for syntax indicating repeated sequences of stitches (denoted within parentheses, brackets, or asterisks) and replaces them with a "written out" version. For example, after processing the sample pattern above, the method would return a String array consisting of the following lines:

```
k3, k2tog, yo, k2tog, yo, k4
k2, p7, k2
k3, k2tog, yo, k1, yo, skp, k3
k2, p7, k2
k3, skp, yo, skp, yo, k4
k2, p7, k2
```

2. Next, the ***parseTextInstructions()*** method creates a 2D ArrayList from the String array by splitting each String in the String array on the delimiter of a comma followed by a space. It then calls the ***processMultiples()*** method using the 2D ArrayList as an argument.
3. **processMultiples():** This method loops through the 2D array list and checks each item to see if it is an abbreviation denoting multiple knit or purl stitches (eg. K6 or p2), and if so replaces it with the corresponding number of individual stitch abbrevations. This replacement and insertion modifies the number of elements in the row, which is why an ArrayList is necessary as opposed to a simple array. After processing the above pattern, the method would return a 2D array corresponding to the following lines:

```
k, k, k, k2tog, yo, k2tog, yo, k, k, k, k
k, k, p, p, p, p, p, p, p, k, k
k, k, k, k2tog, yo, k1, yo, skp, k, k, k,
k, k, p, p, p, p, p, p, p, k, k
k, k, k, skp, yo, skp, yo, k, k, k, k
k, k, p, p, p, p, p, p, p, k, k
```

4. The 2D ArrayList is then returned to the constructor. At this point, a valid pattern should have an equal number of elements (stitches) in each row, so the constructor calls the ***checkRowLengths()*** method to confirm this.
5. **convertToStitches():** If the row lengths are equal, the constructor calls this method, which creates a 2D array of the Stitch enum type matching the size of the text pattern 2D ArrayList. It then loops through both arrays, and for each element, tests all the possible values of the Stitch enum type to see if any of the value names match the stitch abbreviation at that point in the array. If so, a Stitch of that type is saved into the new array. This is the array that is returned to the KnittingPattern program when an Instructions object's ***getStitches()*** method is called.

Parsing Errors

All Exceptions generated during the process of parsing a text knitting pattern or rendering its chart are allowed to bubble up to the method that called the ***renderChart()*** method—the action listener for the ***Render*** button or the ***importFile()*** method, each of which enclose the call to ***renderChart()*** in a try/catch block. The rationale for this is that given the complexity of the knitting pattern syntax, identifying exactly what kind of mistake in the pattern is causing the exception may be quite difficult. Uneven line lengths, for instance, may actually be the result of a missed comma, especially if there is also an abbreviation that cannot be matched to an enum value (because it is actually two stitches with the comma between them missing). Developing error reporting sophisticated enough to helpfully report potential syntax errors (akin to the error messages generated by the Java compiler) would be an interesting challenge, but not one I felt I had time to take on in this project. Therefore it seemed most elegant to catch all exceptions resulting from syntax issues or unknown abbrevations in one place, and report them to the user with a generic error message suggesting referring to the Help pane and checking the syntax before clicking ***Render*** to try again.

Reaction

I chose this project because the task of developing code to parse a kind of language seemed like it could have many real-world applications. I found that even the relatively simple syntax of knitting patterns was quite complex to parse, and needed to be simplified even further to be manageable for this project (in actual knitting patterns there are many more stitch types and syntax rules, line lengths can be uneven, etc.), but I still think it was a useful exercise. I was also pleased to discover the power of the enum type—I had not realized before that enum values could have fields, which was very helpful for this project.