Kiwiplan Framework and Sorting Filter

Prabhjot Singh Jassal
Department of Computer Science
University of Auckland

July 20, 2010

1 Framework

Kiwiplan framework uses many design patterns to minimize the code duplication. Design patterns are particularly useful when designing large size applications. Designing large size application requires considering issues that may not be visible until later in the implementation. Using design patterns helps preventing many problems. One of the pattern being used in the Kiwiplan framework is Model-View-Controller (MVC) pattern. This pattern separates the logic in 3 different parts, where *view* keeps track of how to display the information, *controller* keeps track of the mouse and keyboard inputs from the user, and *model* manages the data and behaviour of the application domain. This pattern is particularly useful for large size applications.

One of the biggest advantage of using this pattern is that, it allows us to represent data in various forms independent of how the data is being stored. The *view* component takes care of this and also allow the user to act on that data. No real processing is being done on this component. On the other hand, most of the processing is being done on the *model* component. This componet stores all the business rules. The *model* component helps in reducing code duplication because the data returned by the model is neutral to how it will be displayed on GUI. This allows us to reuse the same model for different views. Lastly, the *controller* componenet sits in the middle of other two components. No real processing is being done on this component *controller* interprets mouse click or keyboard input from the user and calls necessary methods of *view* and *model* component to get the desired result. For example, if the user clicks on some button on the interface, then instead of doing the actual processing, it simply decides which *model* component needs to be called and how the resulted data will be formated.

Another pattern being used in the framework is the *Singleton* pattern. This pattern ensures that a class has only one instance and provides a global access to it. This pattern is useful when most of the time is being used in creating a different instances of the same class. Object creation takes time and slows down the performance of the whole application. Instead of creating a seperate object everytime, it is better to use *mutator* methods to set the values of the variable. This saves time by not creating unwanted objects and hence improves the performance of the whole application.

Following diagram shows the basic structure of the framework

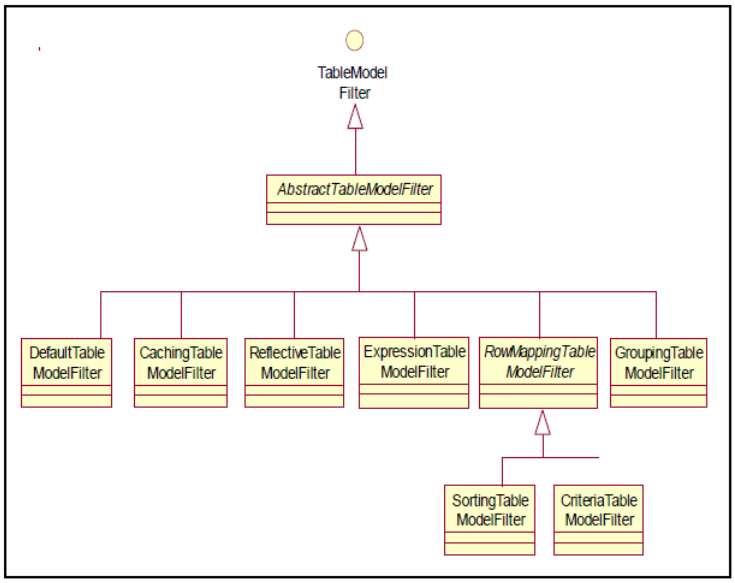

Figure 1: UML Class Diagram of TableModelFilter class hierarchy

The two immediate filter that have been investigated are *SortingTableModelFilter* and *CriteriaTableModelFilter*. Both of these filters have been briefly discussed in the next section.

2 Sorting Filter

Whenever the user clicks on the column header, the column items gets sorted. The first time user clicks on the header, the items will become sorted in an ascending order. Second click on the column header results the items to be sorted in descending order. Every time user clicks on the column header will result the items to be alternate between ascending and descending order.

Each row on the interface represents one record. This record represents some *Business* object of some type. A *business* object is a "reification of some abstraction that is important in the problem domain". As mentioned, in the mid-semester report that the sorting is being done on the mapping of the instances of business object rather than on the actual data itself. The following figure make things more clear:

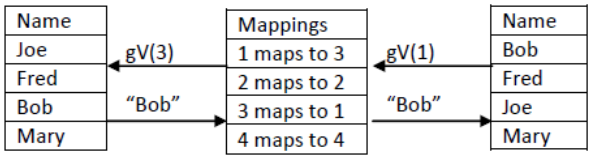

Figure 2: Demonstration of Sorting Mechanism

Thus, to display the first element in the sorted list, *getValueAt(1)* (gV is its abbreviation) will be called. This method then checks in the mapping that which position does 1 maps to. In our example, it maps to 3 and hence, the call *getValueAt(3)* has been made. This returns "Bob" to the filter, which returns it back to the view to display it on GUI.

2.1 Sort Algorithm

Algorithm 1 Shuttle Sort pseudocode

```
shuttleSort(array1, array2, low, high)
if (high − low) == 1 then
  return
else
  middle = (low + high) / 2
  //please note the arrays have been swapped
  shuttleSort(array2, array1, 0, middle)
  shuttleSort(array2, array1, middle, high)
  //merge step
  if max(array1) ≤ min(array2) then
    concatenate array1 and array2
  else
    merge both arrays by comparing one element at a
    time
  end if
end if
```

Shuttle sort has been used to sort the elements. Shuttle sort is based on *Divide and Conquer* strategy and is a variation of standard *Merge sort* algorithm. The time when the framework was getting developed, it was decided to use *Shuttlesort* ahead of *Quicksort*, even though *Quicksort* runs faster. This is because *Shuttlesort* is stable, while *Quicksort* is not[1]. It is easy to understand by the following example, as to why the stable sort is necessary. Let's say that if the list contains these two rows:

Name	Age
Bob	35
Bob	50

Figure 3: Demonstration of Sorting Mechanism

Let's say, the user has already sorted the table according to the "Age" column. Now, the user also wants to sort the table according to the "Name" column.

[1]Quicksort can be made stable by keeping track of the indexes of the items having equal value. However, this project is not just about sorting, that is why, the shuttle sort itself became parallelized.

If the sorting algorithm is not stable, then the ordering of the two rows could be changed. Hence, it is important for an algorithm to be stable. Since, *shuttle sort* is just a variation of *merge sort*, their time-complexities are same $\Theta(n \cdot log(n))$.

The algorithm makes two recursive calls and divides the array in two halves. It is easier to parallelize recursive methods, because each array can be sorted independently of other. The fork/join framework introduced in Java 7 especially makes the parallel programming of "Recursive" algorithms easier To parallelise *shuttle sort* algorithm, the class needs to *extends* a predefined class class *RecursiveAction*. There is another class which can be extended, is *RecursiveTask*. One should extend *RecursiveTask* if the method returns some value. For example, if we would have been interested in parallelising a method which returns the maximum of the given array, then our class should extend *RecursiveTask*. On the other hand, if the method does not return any value, as is the case with our sorting algorithm, then we should extend *RecursiveAction* class. Whether we extend *RecursiveAction* or *RecursiveTask*, the abstract method *compute()* needs to be implemented. Basically, the whole sequential code for the sorting will go in this method with slight modifications. The method *invokeAll* will be used which basically tells thread, that *"get up and work on this partition of the array"*.

Algorithm 2 Shuttle Sort in parallel pseudocode

```
class ShuttleSort extends RecursiveAction
compute()
if (high − low) == 1 then
  return
else
  middle = (low + high) / 2
  //please note the arrays have been swapped
  invokeAll(new ShuttleSort(array2, array1, 0, mid-
  dle), new ShuttleSort(array2, array1, middle, high))
  //merge step
  if max(array1) ≤ min(array2) then
    concatenate array1 and array2
  else
    merge both arrays by comparing one element at a
    time
  end if
end if
```

There is also another way which can be used to sort the elements in parallel. Java 7 has also introduced a new data structure called *ParallelArray*. This data structure has an inbuilt *sort()* method, which does the sorting in parallel. However, to use this class, one needs to have an actual data. In our case, instead of having an actual data, mappings to the data have been used, that's why we stick to the standard *invokeAll()* method.

3 RowVisibility Filter

Imagine the record of all computer science students is displayed on GUI. The staff member might be interested

in displaying only those students whose have achieved an average gpa of 7 and above. Basically, in the background we are checking one by one that whether the student has fullfiled the required *criteria* or not. If yes, then display it on GUI otherwise discard it. Since, the decision of keeping one row is independent of another rows, this can be done in parallel.