

Tabular data representation manipulation

Taras Shymbra

Research Institute for Applied Knowledge Processing

FAW

Helmholtzstr. 16

D-89010 Ulm

Germany

Taras.Shymbra@faw.uni-ulm.de

Abstract

A generic open-source framework for tabular data manipulation(TDM) or to be more specific for tabular data *representation* manipulation is described in this article. The TDM is implemented in Java programming language.

Under the data *representation* we do not mean the data display on user interface screens(*views*) but the *representation* of this data by its *model* or in the TDM terms a *manipulatable model*. A *manipulatable model* does not work with an application data directly but simply redirects queries about it from a *view* to a real data *model* through an internal array of integers which hold its current state(e.g. the order and the number of the rows). The state of a *manipulatable model* is modified from outside by tabular *data manipulators*(e.g. so it can appear to be sorted or filtered). To get additional parameters required by a manipulation algorithm and to support the decisions making concerning the data elements being considered during a manipulation, *data manipulators* use *manipulation criterions*.

The TDM framework provides common implementations of a tabular data row filter and a tabular data sorter. They can be used for Java Swing's `TableModel` filtering and sorting and hence applications that use Java Swing's `JTable` can benefit from using the TDM.

The main advantages of the TDM framework are its simplicity and flexibility.

Keywords

Java, Swing, Data Filtering and Sorting, Model View Controller, iData, Software Development, Software Architecture, Design Patterns

Introduction

Applications that must display and manipulate of large amounts of often complex tabular data(e.g. enterprise applications) have usually a lot of user interface screens to handle this data. The data has to be presented lots of different ways(tables and lists) for different purposes. Advanced user interfaces should allow an user to sort through large volumes of information or filter out those data elements which does not satisfy his or her information requirement. In other words they should allow data *manipulations*.

Applications of this kind most likely use sophisticated UI frameworks like Java Swing to manage its graphic interfaces. An influential role for most such frameworks plays the Model View Controller(MVC) architectural pattern. The Java Swing is also based on this pattern.

The MVC pattern is about the separation of data displaying from data model what is one of the most fundamental heuristics of good software design[FOWMVC]. However, when there is a need for data *manipulations* the following question arises: where does *manipulation* behavior belong. Let's consider data particular kind of data *manipulation* - data sorting. If one want to see many different views of a single table model and one might want each view to show the data in a different order, clearly the view has to be responsible for the sorting. On the other hand the model may have special knowledge of the data which enables it to sort much more efficiently. Anyway, sorting is a data issue not a *view* issue, it belongs to the model. So, if sorting belongs to the model, all of the data model implementations will need to code a sorting algorithm. Since a reasonable job can be done generically, why not do it once in the view? Now, let's consider another kind of data *manipulation* - data filtering: showing rows of data only if they meet certain criteria. That can be done generically too so it should go in the view as well.

To solve this dilemma the TDM takes radically another approach. It encapsulates manipulation behavior and manipulation algorithms within separate so-called data *manipulators* that operate on *manipulatable models*. That means that in the TDM the *manipulation* behavior belongs to the model side not to the view one.

We will further discuss relationships between the TDM artifacts along with their integration into traditional MVC architecture on the example of Java Swing.

Data Manipulation Layer

Let's consider how the *manipulation* behavior can be implemented on the model side in the MVC architecture.

One of the biggest mistakes that is being often done when designing and using models is data copying. When a need to reinterpret (e.g. sort) the data arises, the application creates a new *model* and then *copies* the data into it from the original one. The *view* must be then switched to the new *model* object. Another design implies operating directly on the original *model*. To illustrate the problem in gory detail suppose one want to clear any manipulation effects or to apply filter on the original data. Once it is manipulated, its initial state changes. To fix this problem the data copying (for backup) is required and this introduces unnecessary model state management burden.

Data copying is often unnecessary and extremely expensive. This is one of the key reasons why many Swing applications display poor performance. What most programmers do not think about is that models are merely interfaces; they *do not* need to actually store any data [VADD]. Keeping this in mind and to avoid the outlined above problems the TDM uses technique that does neither store nor modify the original data in an underlying real data *model* - an existent data *model* implementation that actually contains the domain data. The TDM adds just another level of indirection between the real data *model* and the view introducing the data manipulation layer (DML) which is integrated in the MVC architecture as shown in figure 1.

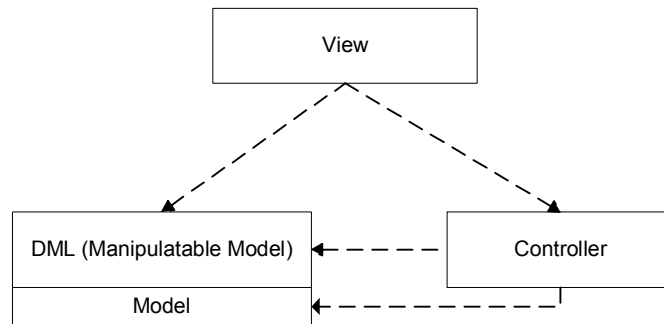


Figure 1: The DML place in the MVC architecture.

The DML is represented by the bundle of classes that implement the *manipulation* behavior. The core TDM classes are shown in figure 2.

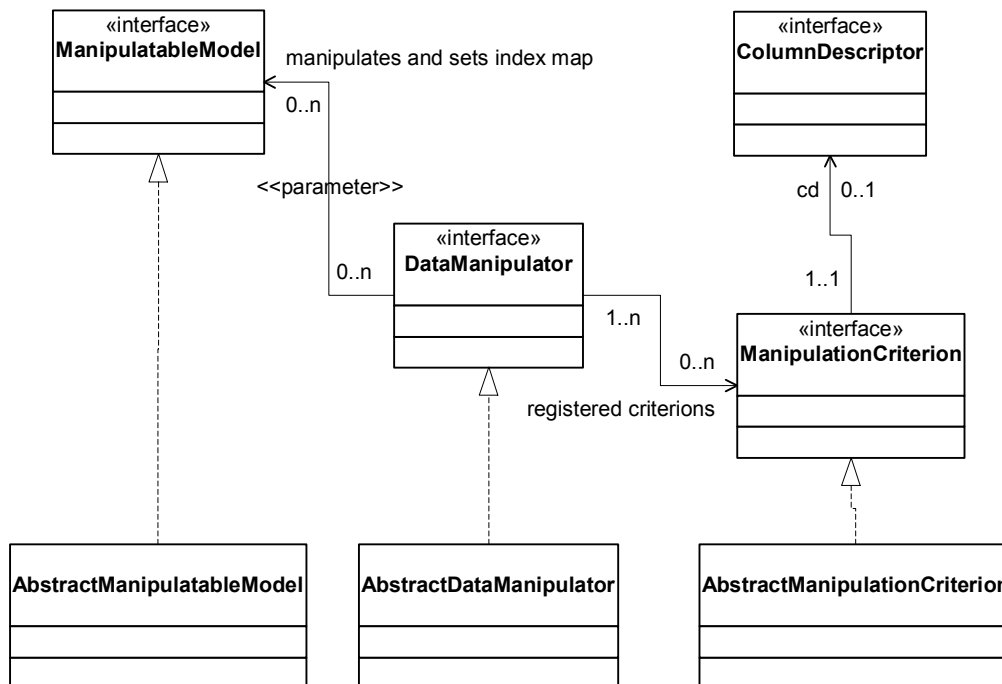


Figure 2. The TDM Overview Class Diagram.

The TDM abstractions' description along with interfaces and classes that represent them proceeds below.

Manipulated Model

A *manipulatable model* is the TDM's key abstraction that is represented by the `ManipulatableModel` interface. First of all, it is just a way of data representation for data *manipulators* (for more details see section „Data manipulators“) as well as an artifact on which data *manipulators* (they do not know anything about the MVC) actually operate. The `ManipulatableModel` has a number of operations that provide the following services for them data *manipulators*:

- allows changing its state through setting an *index map* (i.e. applying a manipulation)
- provides access to the domain data directly
- provides access to the domain data indirectly through the *index map*
- provides access to the *index map* itself.

Those services are quite enough for data *manipulators* to implement any data *manipulation* algorithm.

Since a *manipulatable model* must not physically change a domain data, it needs additional means to store its state after a manipulation. For this purpose the `AbstractManipulatableModel` class (basic implementation of `ManipulatableModel` interface) has an internal array of integers which holds the information about the order and the number of the rows – an *index map*. It simply contains the real data rows' indexes in such order as they should appear in a *view*. The size of the index map denotes the number of rows after a manipulation. In figure 3 an example of an *index map* is represented after applying sorting in the descending order and then filtering out elements (cars) those names begin with „F“.

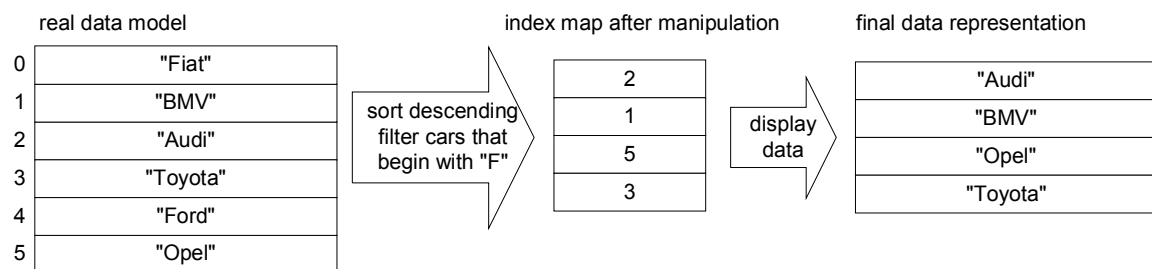


Figure 3: An Example of an *Index Map*

As can be understand, `ManipulatableModel`'s operations that provides access to the domain data indirectly through the *index map* first use it to get the corresponding manipulated data element's index for a given real data element index (i.e. index of the element in the real data model). An example of the *index map* usage is shown below in listing 1.

Listing 1. Implementation of `ManipulatableModel`'s `getMappedElementAt` method

```
public Object getMappedElementAt(int index, int columnIndex)
{
    if (columnIndex < 0 || columnIndex > getColumnCount())
    {
        throw new IndexOutOfBoundsException("Column number "+columnIndex+" doesn't exist in the model");
    }

    int newRowIndex = index;

    //get the manipulated index if available
    if (null != indexMap)
    {
        newRowIndex = indexMap[index];
    }

    //get the element from the domain data using the newly calculated index
    return getElementAt(newRowIndex, columnIndex);
}
```

The additional level of indirection that is represented by the DML is implemented on the low level in the `AbstractManipulatableModel` by means of *index map*.

To become a full-fledged participant of the MVC triad, `ManipulatableModel` must be connected with an application specific real data model implementation what involves some integration efforts. This is accomplished by connecting the `ManipulatableModel` interface to the Java Swing's `TableModel` interface as shown in the next section.

Manipulated Model Adaptation to the Java Swing

To use the TDM in applications whose GUI are built with the Java Swing component kit, a special class `ManipulatableTableModel` have been implemented. It is elaborated as an *adapter* according to the Adapter Design Pattern[GOF]. So, it converts the `ManipulatableModel` interface into the `TableModel` interface that the Java Swing's `JTable` expects and lets the both classes to work together that could not otherwise because of incompatible interfaces. In terms of the Adapter DP participants, `TableModel` is the *target*, `JTable` is the *client*, the `ManipulatableModel` is the *adaptee* and the `ManipulatableTableModel` is the *adapter*(figure 5).

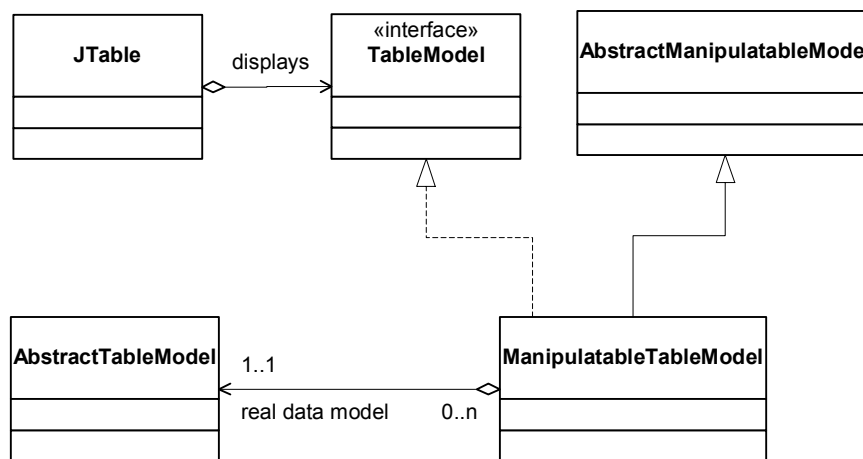


Figure 5: The TDM Adaptation to the Java Swing kit Class Diagram

As can be seen in figure 5, `ManipulatableTableModel` aggregates an `AbstractTableModel` instance which plays the role of a real data *model*. It must be supplied at a `ManipulatableTableModel` instance creation time. Such design is called *composition* since the existing `TableModel` implementation(e.g. `DefaultTableModel`) becomes a component of the new `ManipulatableTableModel` class [BLOCH]. Each `TableModel`'s operation implementation either forwards the method call to the corresponding method on the contained `AbstractTableModel` or invokes appropriate `ManipulatableModel`'s operations when an indirection through usage of the *index map* must take place.

To plug the `ManipulatableTableModel` into the Java Swing's architecture, a client application must perform several additional steps. So, it must not pass an existent `TableModel`'s implementation to a `JTable` as its model but a `ManipulatableTableModel` instance. The TDM contains a mediator class `ManipulatableTable` that takes care about those TDM-to-Java Swing integration issues.

Manipulation Events

The TDM provides notifications mechanism to propagate a *manipulatable model* state changes by publishing *manipulation events* to interested objects (listeners). The basic support for *manipulation events* is implemented in the `AbstractManipulatableModel` class as shown in figure 4. The *manipulation event* is represented by the `ManipulationEvent` class. The `AbstractManipulatableModel` aggregates only one instance of this class and uses it for all *data manipulated* notifications. Such notifications occurs particularly when a manipulatable model's *index map* is set. Classes that want to observe *manipulation events* must implement `ManipulationListener` interface.

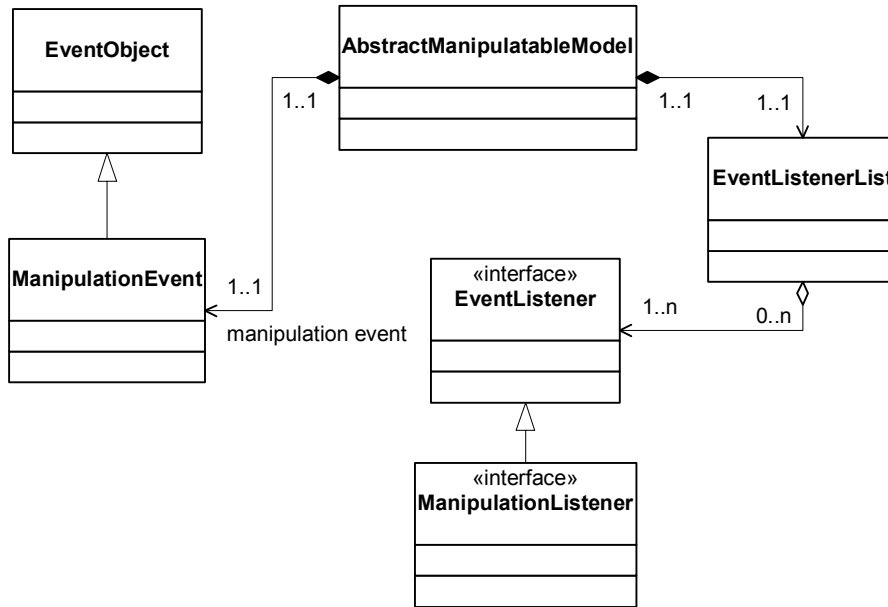


Figure 4: The TDM Manipulation Events Class Diagram

Data Manipulators

We argued that the *manipulation* behavior must be located on the *model* side in the MVC architecture and discussed how the DML carries out this task into practice via *manipulatable model*. Now we face a task of implementing particular data *manipulation* algorithms (e.g. data sorting or data filtering). The simplest way is to encapsulate the *manipulation* machinery in the *manipulatable model*. In this case we will be up to as many particular *ManipulatableModel*'s implementations as there are *manipulation* algorithms. Everything is ok with such an approach until an application requires the data to be *manipulated* only in a single way.

But more difficult situation arises when the application data must be manipulated in different ways using different *manipulation* algorithms and effects of *manipulations* must be combined. Let's take for example the *ManipulatableTableModel* class described in section „Manipulated Model Adaptation to the Java Swing“. Because it „is-a“ *TableModel* it can be passed to another *ManipulatableTableModel* instance playing a role of a real data model. Therefore it is feasible to chain multiple *manipulatable models* to combine their manipulation effects.

Practically such a design is acceptable and is discussed particularly on Swing model filtering example by Mitch Goldstein[COLD]. However it leads to a couple of problems. First of all, each *manipulatable model* holds its own *index map* and each time a view requests its model (data elements requests occur especially often during view repainting) the request must go through all *manipulatable models* in the chain until it reaches the real data *model*. This brings additional quantum of computation and hence is expensive. Second, the chaining technique puts additional stipulation on the collaboration between *manipulatable models* in the chain. To keep *index maps* of all connected *manipulatable models* consistent, each one must publish its *index map* changes along the chain. To fix this problem an unnecessary bulk of logic must be added to the *ManipulatableModel*'s implementations.

To summarize, the TDM rejects this design both due to performance as well as to complexity reasons. In the TDM the concept of *manipulatable model* is further separated from the concept of *data manipulators* that operate on *manipulatable models* by setting an *index map* as a result of their *manipulation* (see figure 2). A *data manipulator* is represented in the TDM by the *DataManipulator* interface (see figure 2) and concrete *manipulator* implementation specifies a data *manipulation* algorithm which normally should be abstract and application independent to enhance its reuse. Examples of data manipulators are: data sorters, data filters, data reverses etc.

This approach fully eliminates the need to chain *manipulatable models*. There is only one *manipulatable model* that seats between view and real data *model* and hence only one *index map* which is set by a *data manipulator*.

Of course such separation of concerns does not come for free. The *manipulatable models* is passive, it only manages the *index map* related issues. This requires some external stimuli like a *controller* that manages the state of the *manipulatable model* and coordinates its *data manipulators* accordingly. The TDM contains such a

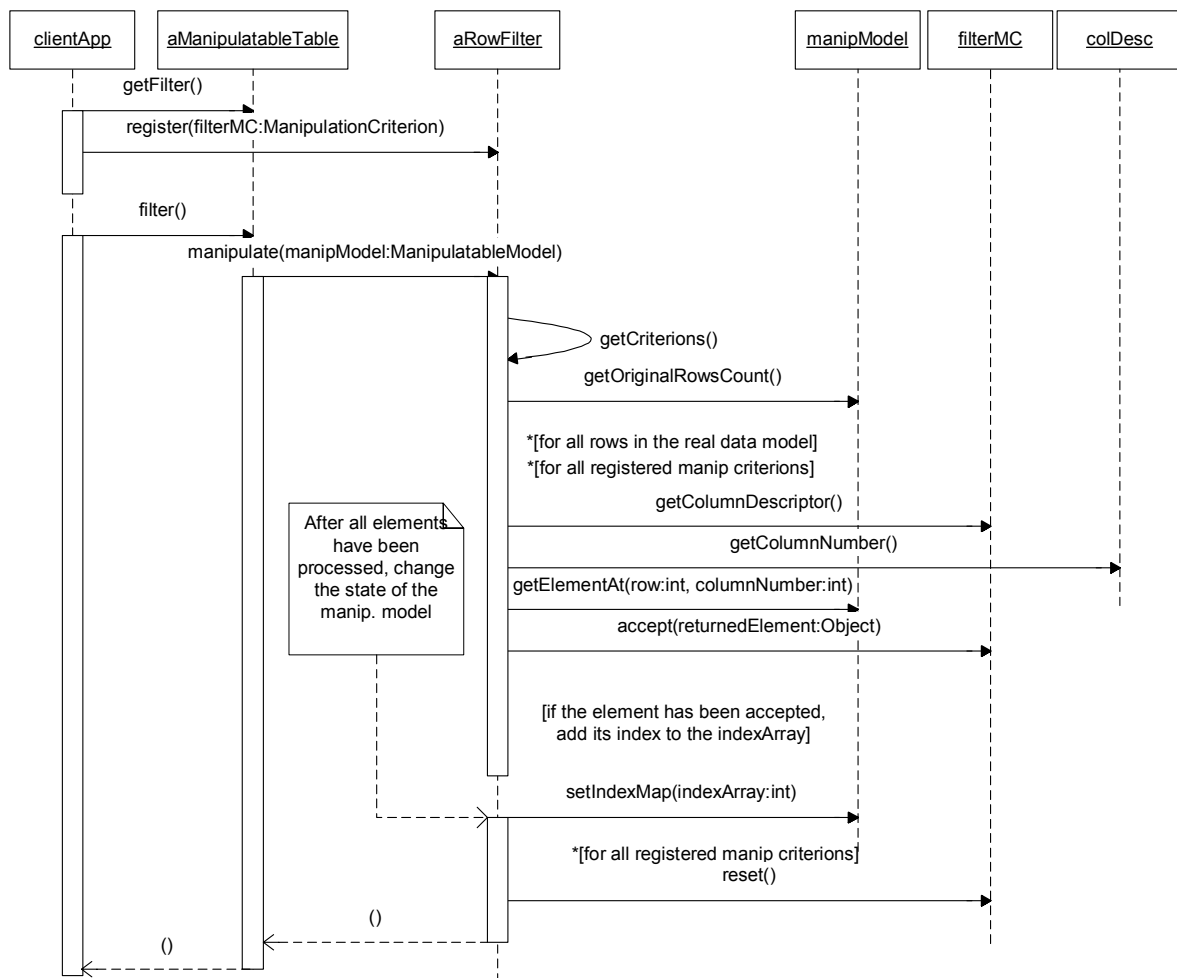


Figure 7: The ManipulatableTable Filtering Sequence Diagram

Another very important *data manipulator* implementation that the TDM contains - is a data sorter represented by the `FullSorter` class. It can sort a real data *model* by specified column and demands that each data element must implement `Comparable` interface. The `FullSorter` obtains the sorting order as well as the column number to sort from specified *manipulation criterion* and to be more specific from special tailored to sorting needs `TableSorterManipulationCriterion` class instance(see figure 6).

Both `FullSorter` and `RowFilter` perform their manipulations with respect to some *manipulation criterions*. The notion of *manipulation criterion* is discussed in details in the next section.

Manipulation Criterions

Some *data manipulator's* implementations those manipulation algorithms require additional, application specific parameters have to be obtained from a client application. For this purpose, the TDM has the `ManipulationCriterion` interface that models a *manipulation criterion* - an universal intermediary between freestanding concrete *data manipulator* and the application needs. Both application specific information required by *data manipulators* and application specific logic are represented by a concrete *manipulation criterion* implementations. Normally each *manipulation criterion* is associated with some aspect(property) of a domain data object. In tabular data representations each data object's aspect is normally represented in a separated table column and hence each `ManipulationCriterion` instance is it connected with a table's column though a *column descriptor*. The `ManipulationCriterion` instances does not represent column related information(like column number or column name etc.) directly but provide it by associated with it `ColumnDescriptor` object(see figure 2). The `ColumnDescriptor` can instruct *data manipulators* which

column to use to fetch the cell object that must be passed to the owner ManipulationCriterion for checking(required by filters) or which column to use to perform sorting.

As was shown above, the TDM classes FullSorter and RowFilter use *manipulation criterions*(see section „Data Manipulators“) with different purposes. A *manipulation criterion* can only them participate in the manipulation process when it is registered to a *data manipulator*(see figure 2). There can be more than one *manipulation criterion* registered for a given *data manipulators*. The TDM uses small optimization and namely it supports the *manipulation criterions* sorting according to their *weights*. The semantics of *weight* is a matter of contact between custom *manipulation criterions* and *data manipulators* that use them. For example, a manipulation criterions' weight can notice the "computational weight" of heavily used method's implementation(e.g. the RowFilter calls accept method of a ManipulationCriterion for each real model data element it considers). The higher is the weight's value, the more time is required to execute this method and hence the "heavier" (from a performance standpoint) is the manipulation criterion. So, *data manipulators* which must rely on more than one manipulation criterion can sort them and consider first an "easier" one. If the manipulator is satisfied with the result that the later returns, it can skip considering the "heavier" *manipulation criterions* avoiding unnecessary CPU heat.

The basic support of *manipulation criterion'* *weights* is implemented in the AbstractManipulationCriterion class that implements Comparable interface indicating that its instances have a natural ordering.

On the data manipulators side the TDM's class AbstractDataManipulator is responsible for their sorting before they will be used by custom subclasses.

Another useful facility a ManipulationCriterion provides for client applications, is the change tracking of its instances. The TDM's ManipulatableTable uses this benefit to avoid redundant table sorting until the column number or the sorting order actually change.

Using the TDM with the IData toolkit

As the TDM is about the data manipulation, the Swing iData toolkit represented by Jonathan Simon[IDATA] is about maintaining that data within advanced Swing components (i.e. JTable and JTree). The iData technique establishes a generic architecture for integrating intelligent data with Swing components while preserving the Model-View-Controller architecture. The TDM can be just nice integrated with the iData and this will keep complicated Swing development even simpler. The iData toolkit really shines for tabular data sorting since in applications that use the iData each table cell is represented by an ImmutableIData instance(it represents a concrete aspect of the underlying domain data object). Custom iData objects can implement the Comparable interface enabling type safe sorting and can therefore be sorted by the TDM's FullSorter. So data filtering: RowFilter algorithm is independent from the data objects on which it operates and relies concerning the data issues on external *manipulation criterions*. With the usage of the iData particular *manipulation criterions* will work only with concrete iData types. Such a decomposition promotes loose coupling between data manipulators and the data objects as well as high cohesion between TDM's and iData entities. In general, it is strongly recommended to use the TDM with the iData.

Conclusion

In general, the main idea taken by the TDM is the separation of data storing, data representation and data manipulation concepts. The TDM closely addresses data representation and data manipulation issues while leaving data storing issues aside. The data representation by a *manipulatable model* can differ from the data representation by the underlying real data model.

The ManipulatableModel instance is a „passive“ entity and in contrast to „active“ so-called *self managing models* that encapsulates manipulation algorithm must be controlled from outside by external artifacts(*data manipulators*) that drive its state.

The TDM's manipulation technique demands that *data manipulators* first process all *manipulatable model* elements before setting it into the correct state while *self managing models* can re-interpret data „on-fly“(by demand) using the lazy evaluation approach. Problems can show up in situations when a real data *model* contains a large data set but a client application should show at a time only a small its subset(for example, this can be caused by a small GUI component viewport size). So, for example, the TDM's RowFilter will process all real data *model* elements which in most situations might by excessive. This can be considered as one of the TDM's weaknesses. On the other hand, as was shown in this paper, the „active“ approach faces some difficulties with *models* chaining when manipulation effects must be combined.

Two simple data manipulators implementations – RowFilter and FullSorter along with managing code(in ManipulatableTable class) are provided to facilitate client applications in simple tabular data sorting and filtering. Those implementations can easily be replaced with other ones that use advanced or

proprietary algorithms and exploit special knowledge of the data, either to improve performance or customize the behavior for a particular application. The only what is expected from client applications are custom *manipulation criterions*.

One of the biggest benefits of the TDM that it does not fall into data copying or modifications of a real data *model*. It leaves an original data structure untouched. The *index map* mechanism is build using an array of ints(in Java it is one of the most effective data structures) and therefore as an additional level of indirection does not have much performance impact in practice.

The TDM can be ideally used for developing interactive user interfaces where users explicitly initiates any manipulation for example sorting or filtering. It is small, simple, easy to understand, flexible and fast framework.

References

[IDATA] J. Simon, *Intelligent data keeps Swing simple*, available at <http://www-106.ibm.com/developerworks/java/library/j-idata/>

[FOWMVC] M. Fowler, *Model View Controller*, available at <http://www.martinfowler.com/isa/mvc.html>

[VADD] S. Stanchfield, *Model View Controller*, available at <http://www7.software.ibm.com/vad.nsf/Data/Document2329>

[GOF] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley, 1995.

[BLOCH] Joshua Bloch, *Effective Java Programming Language Guide*. Reading, MA: Addison Wesley, 2001.

[COLD] M. Goldstein, *Swing model filtering. Using filter objects to reinterpret data and state models*, available at <http://www-106.ibm.com/developerworks/java/library/j-filters/>