Refactoring Data Transforms in MaTSE for Flexibility

Paul Craig*, Jessie Kennedy†

Center for Information and Software Systems, Edinburgh Napier University

Abstract

In early prototypes of the Microarray Time-series Explorer (MaTSE) data transforms were tightly coupled with visualisation components. While this allowed us to achieve our initial objective of developing the application to the level where we were able to demonstrate the basic visualisation technique with a specific data-set, refactoring to a more flexible code structure was required in order to apply a larger number of transforms and accommodate a wider variety of data-sets. This paper reports on our planning and execution of this refactoring exercise.

Keywords: Refactoring, Information Visualisation.

1 Introduction

Refactoring can be described as the process of changing a software systems internal structure in order to improve the developers' prospects of successfully implementing further functionality [1]. As initial software requirements evolve, code structures designed to support those requirements become outdated and less able to support new functionality. Refactoring deals with this problem by updating structures as they begin to lose their effectiveness. Small refactorings take less than a day and can include tasks such as the renaming of methods and variables. These are applied continuously throughout development and are partially automated within most modern IDEs. Substantial changes in software requirements tend to elicit what are known as 'large refactorings'. These involve significant changes to an applications core architecture, take more than a day and can require a significant amount of planning in order to be successful [2]. The integration of new data-transforms into the Microarray Time-series Explorer (MaTSE) software is a case where the introduction of new software requirements called for large refactoring.

2 Original Code Structure

The version of MaTSE released prior to our large refactoring exercise [3] (Figure 1) includes two coordinated views of the data. These are a line chart and a scatter-plot. Individual genes are represented as poly-lines in the line chart and points in the scatter-plot. The views are coordinated in two ways. Firstly selections made in either view, to highlight or select genes, are displayed in both views. Secondly, a slider attached to the line-chart selects a time-slice through the data and changes the position of genes in the scatter-plot.

This version of the software was limited to working with a specific type of microarray data. It was assumed that the data would contain only one biological sample and the user would not require to compare samples in their analysis. We also assumed that two particular data transforms would always be appropriate. These were the $\log_2$ scaling of all data points and the scaling of data to values at the first time point. This meant that in our original code structure all data transforms could be hard-coded into a single class. This class was named *TSLinechartScatterplot* and acted as a wrapper for MaTSE's line-chart and scatter-plot components passing the indices of brushed and selected genes as well as handling data transforms.

This basic code structure allowed us to achieve our early objectives and code an application that could be used to demonstrate the basic IV technique with a small number of data-sets and the implementation of a limited number of fixed data transforms.

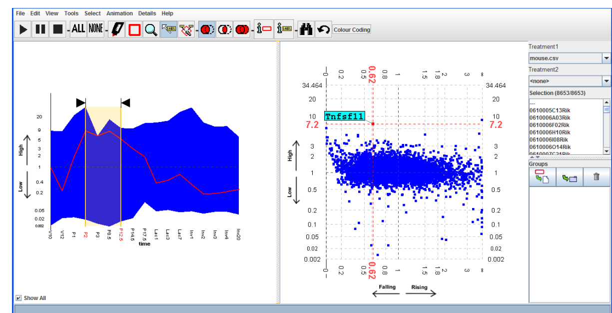

Figure 1. A screenshot of the MaTSE interface as developed before refactoring.

3 Additional Requirements and New Code Structure

As we began to consider an additional set of requirements for MaTSE, weaknesses in its early code structure became apparent. The requirements that necessitated a rethink of our code structure relating to data transforms are summarized as follows:

1. Should work with a variety of microarray data with different characteristics: different transforms are required to facilitate the analysis of these data. For example data may require to be scaled using a $\log_2$, $\log_{10}$ or cube root transform. Some data need to be rescaled to a certain time-point or allow time-points to be reordered. Replicate values may need to be combined in the data.
2. Should work with multi sample-data and allow different samples to be compared: This would be likely to involve different configurations of views relating to different portions of the data.
3. Should allow the user to store and retrieve selections used to find patterns in the data: Here there was a need to store not only the results of queries but also the parameters used to compose those queries. As queries are executed using views of transformed data there is a need to store details of transforms along with other selection criteria.

It is easy to see how our existing code structure would buckle under the weight of these new requirements. The first thing to be noticed was that the *TSLinechartScatterplot* was too closely linked to the existing layout. Since this class handled data transforms and there would be a bigger role for transforms in future versions of the software it made sense to remove this class, build a separate structure for data transforms and link line-charts and scatter-plots directly to models for selection and data

*e-mail: p.craig@napier.ac.uk

†e-mail: j.kennedy@napier.ac.uk

transformation. Since we wanted our visual components to work with raw data and different transformed versions of the data, we decided to assign raw data and transforms a common interface. This interface was to be called *Data*. Figure 3 shows the new code structure constructed to support data transforms.

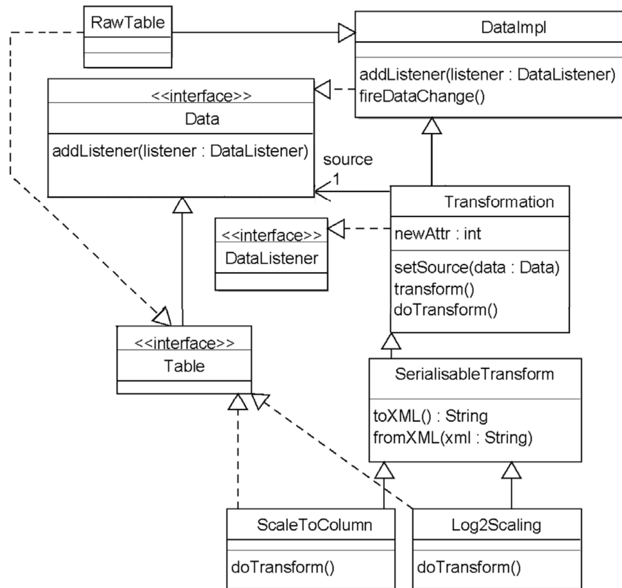

Figure 2. Updated MaTSE code structure for data transforms where transforms and raw data share a common interface.

Our new code structure was designed to perform a number of functions. Firstly it was to allow alternative transforms to be chained together in a flexible manor. Each time a data transform is initialized its 'source' object can be set to an instantiation of any class that implements *Data*. This can be either raw data or another transform. The *setSource* method in *Transform* automatically sets the new transform to listen to its *source* object. If there is any change to the *source* object, its *transform* method fires a change event to all its listeners. This causes changes at any point in the transform pipeline to be propagated all the way along the pipeline until they effect the appropriate changes in visualisation components. When visualisation components are written to implement the *DataListener* interface they can be registered as listeners in the last transformed version of the data and act as end points to their respective data transformation pipelines.

Another important feature of this structure is that separate transform objects can share the same *source* object. This allows for forks in a pipeline so views can share a subset of transforms. This feature is used in MaTSE when different views need to share the same value-rescaling but required separate column-filter transforms to display different samples.

Specific data transforms can either extend *Transform* or *SerialisableTransform*. When *SerialisableTransform* is used, transforms must implement the abstract *toXML* and *fromXML* methods supporting serialization. Figure 2 shows two specific transforms that extend *SerialisableTransform* and extend the *Table* interface. While *Table* is the only specialization of the *Data* interface shown in Figure 2, it's possible to extend *Data* with other specializations to represent different data types. Other data types used in the current version of MaTSE are *Interpolation* and *Pairs*. The *Interpolation* data type returns continuous values for elements and is the result of transforms such as cubic-spline and linear interpolation. The *Pairs* data type holds two values for each element and is the result of transforms used to calculate scatter-plot coordinates from interpolations.

4 Refactoring Process

The key to a large refactoring exercise is to break the process down into small increments [2]. It's also important to realize how easy it is to choose erroneous refactoring routes. To avoid irrevocable damage to code it is advisable to use software versioning software which can reverse any changes made during refactoring. In our project we decided to use SVN. Stages of refactoring were planned so as to provide opportunities to test different portions of the new code structure and minimize the amount of disruption to our software's functionality.

The first step of our refactoring exercise was to remove *TSLinechartScatterplot* and create the section of our new code structure relating to raw data. This included the introduction of our *Data* interface, *Table* interface and *RawTable* class, and allowed our structure to be tested with a visualisation of the raw data. The next step was to introduce the *DataListener* interface and adapt our visual components to implement this interface so their contents would be redrawn when the data was changed. Our third step was to code the *Transformation* class, *Log2Scaling* transform and *ScaleToColumn* transform. This allowed us to test multiple transforms joined in a pipeline with basic transforms that converted tabular data. The fourth step in refactoring was to introduce additional data types and code transforms for interpolation and the calculation of scatter-plot coordinates. This allowed us to test the data transform code structure for different data types and modularize interpolation and layout operations in our scatter-plot. Our final task was to introduce the *SerialisableTransform* abstract class and make transforms we wanted to serialize subclasses of this class. This included the implementation of *toXML* and *fromXML* methods.

5 Conclusion

This paper introduces a new code structure for multiple piped data transforms in an information visualisation application and describes the refactoring process involved in the conversion of MaTSE to this new structure. Our primary motivation for this refactoring was to allow data transformation structures to be more flexible during further development of the software. This has proven to be the case and the code structure has since been successfully used to support a variety of data-transforms and data-transform configurations. These new transforms include those that deal with replicate values, filtering data columns, rescaling and comparing samples. Transforms are also combined in different ways with views of different samples sharing the same rescaling transforms and different column filters. The modularization of data-transforms in our new data structure also has the advantage of allowing the substitution of transforms without destructive changes to the code. This has proven useful for exercises such as the testing of different scatter-plot layouts, different interpolation methods and alternate implementations of performance critical transforms.

References

[1] Fowler, M. (1999). Refactoring: Improving the Design of Existing Code (1st ed.): Addison-Wesley.

[2] Roock, S., & Lippert, M. (2005). Refactoring in Large Software Projects (Paperback ed.): John Wiley & Sons.

[3] Craig, P., Kennedy, J. and Cumming, A. (2005). Animated Interval Scatter-plot Views for the Exploratory Analysis of Large Scale Microarray Time-course Data. Information Visualisation, 4(3), 149-163.