Data Mappings in the Model-View-Controller Pattern[1]

Martin Rammerstorfer and Hanspeter Mössenböck

University of Linz, Institute of Practical Computer Science
{rammerstorfer,moessenboeck}@ssw.uni-linz.ac.at

Abstract. The model-view-controller pattern is used to keep a data model and its views consistent. Usually there is a one-to-one correspondence between the data in the model and its representation in the views, which is sometimes too inflexible. We propose to add so-called data mappings between the model and its views. Data mappings are components that can be plugged together hierarchically. They can perform any transformations on the data as well on notifications, thus allowing a more flexible collaboration between a model and its views. GUI builders can be extended so that a user can add data mappings to a graphical user interface interactively, i.e. without programming.

1 Motivation

Applications with graphical user interfaces (GUIs) usually rely on the *model-view-controller pattern* (MVC pattern) [1] that was first used in the Smalltalk system around 1980. The idea of this pattern is to decompose an applications into three parts:

- The **model**, which stores the data of the application (e.g. a text, a drawing, or a table of numbers for a spreadsheet).
- One or several **views**, which show a representation of the model on the screen (e.g. a text window, a graphics window or a spreadsheet window). All views show the same model and are kept consistent whenever the model changes.
- One or several **controllers**, which handle user input and update the model accordingly. Every view has its own controller, which handle keyboard and mouse input in its own way.

Fig. 1 shows how these parts are connected and how they collaborate.

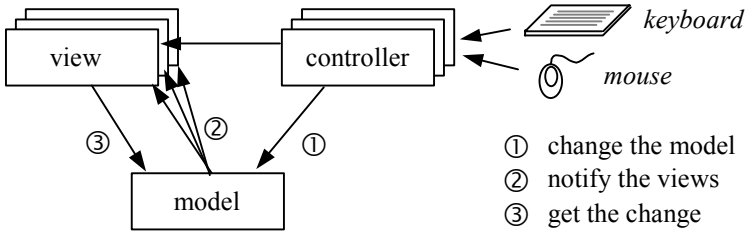

Fig.1 Collaborations in the MVC pattern

[1] This work was supported by the Austrian FWF under the project number P14575-N04.

When the user presses a key the controller intercepts this event and updates the model (1), e.g. by inserting the typed character into a data structure. The model informs all its views about the modification by sending them notification messages (2). Each view in turn accesses the model to get the modification and updates its representation accordingly (3). The controller may also access the view directly, e.g. for scrolling or for setting a selection. All data modifications, however, are first done in the model and then propagated to the views. In this way, the MVC pattern guarantees that the views are consistent all the time.

Views and controllers always occur in pairs. Therefore they are sometimes merged into a single view/controller component that takes on the responsibilities of both the view and the controller.

If a user interface consists of multiple GUI elements such as text fields, checkboxes or radio buttons, the MVC pattern is applied for each of them. That means, for example, that every text field assumes the role of the view/controller component while the corresponding model is stored as a string in memory.

GUI builders and their problems with MVC

Graphical user interfaces are often implemented with a GUI builder, which is a tool that allows the interactive composition of user interfaces by dragging and dropping GUI elements into a window, moving them to the desired position and specifying how they should react to input events.

One problem with current GUI builders is that they don't fully conform to the MVC pattern, i.e.:

- A GUI element (e.g. a text field) often already contains its model value (e.g. a string). There is no clear separation between the model and its view. Both concepts are implemented by the same object. Although some GUI builders (e.g. [8]) generate code such that the model value is kept as a separate variable outside the GUI element, it is usually not possible to connect the GUI element to an arbitrary variable in an already existing model.
- In most cases, GUI builders generate just one view for a given model. For example a string usually cannot be shown and edited in multiple text fields.
- GUI builders require that the structure of a model exactly matches the structure of the view, therefore the design of the GUI often dominates the structure of the model. For example, if the GUI contains a color palette the corresponding model value is usually a RGB value and cannot easily be a string denoting the name of a color. Sometimes, an existing model also dominates the design of the GUI. For example, if a time value is stored in seconds the GUI cannot easily be made such that it shows the value in hours, minutes and seconds.

These problems arise because there is an unneccessarily close coupling between the model and the view, which does not allow multiple views or transformations between the values that are stored in the model and those that are shown in the view.

What we need is a more flexible framework that decouples a model from its views. It should not only allow a model to have several views but also to arbitrate between nonconforming models and views by using a sequence of data transformations. This led to the idea of so-called *data mappings* that can be inserted between a model and its views. Data mappings are described in the next section.

2 Data Mappings

A *data mapping* is a layer between the model and its views. It forwards the `get` and `set` requests from the view/controller component to the model as well as the change notifications from the model to the view/controller (Fig. 2).

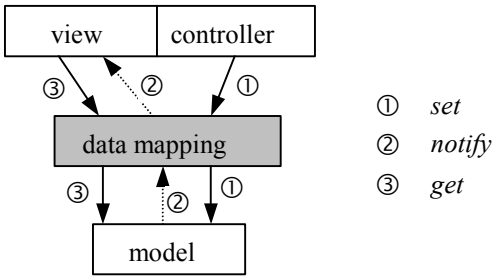

Fig. 2 A data mapping layer between model and view

Instead of just forwarding the requests, the mapping may also do some transformations between the data that is stored in the model and the data that is seen by the view. For example, an integer in the model may be transformed to a string in the view, or a string in the view may be transformed to a boolean in the model. By exchanging the data mapping one can adapt a given view to different models and a given model to different views. This includes the possibility to add multiple views to a model and to keep them consistent.

The data mapping layer may either consist of a single mapping or it may be composed from several mappings that together perform an arbitrarily complex transformation (similar to the way how Unix pipes work). The mapping layer may even be plugged together from prefabricated components using a tool that is similar to a GUI builder. This task can be done interactively, i.e. without programming, which saves considerable implementation time.

Data Mapping Elements

A data mapping layer consists of a number of *mapping elements* that can be plugged together to accomplish a certain transformation. All elements are derived from a common base class `BaseMap`, which has the following (simplified) interface:

```
public abstract class BaseMap implements ChangeListener {
  public BaseMap father;
  public abstract void   set(Object model, Object data);
  public abstract Object get(Object model);
  public void notify(Object model) { father.notify(); }
}
```

The class `BaseMap` has three methods: `set` transfers data from the controller to the model, i.e. it changes the model; `get` reads data from the model and returns it to the view; `notify` finally forwards change notifications from the model to the view, i.e. it invalidates the view and causes a `get` operation.

`BaseMap` implements the `ChangeListener` interface, which allows descendants of this class to be registered as change listeners at variables in the model. If such a variable is changed, the `notify` method of the registered listener is invoked and the notification goes all the way up to the view.

```
public interface ChangeListener {
  public void notify(Object model);
}
```

Subclasses of `BaseMap` implement various kinds of data mappings such as conversions, access operations, notifications and calculations. We will look at some of these mappings in the following subsections.

Model References

The most basic data mapping is the *model reference*. It is used to connect a mapping to a variable of the model and to perform `get` and `set` operations on it. The reference is registered as a listener at the model variable and is notified whenever this variable is changed by the underlying program. Here is a simplified implementation:

```
public class RefMap extends BaseMap {
  private String name;  // the referenced variable in the model

  public RefMap(String name, Object model) {
    this.name = name;
    use reflection to add this as a listener for name in model
  }

  public void set(Object model, Object data) {
    use reflection to set name in model to data
  }

  public Object get(Object model) {
    use reflection to get data from name in model
  }
}
```

If a data mapping consists just of a model reference it performs no transformation but only forwards `get` and `set` requests as well as notifications between the model and its view/controller.

Method Mappings

In addition to visualizing a model on the screen, a graphical user interface also has the task to respond to user input. For example, when a button is clicked a certain action should be performed. This behavior can be modeled by so-called *method mappings*, which encapsulate Java methods that are called during a `set` operation.

Method mappings and model references are those model-view connections that are usually also established by a traditional GUI builder. In contrast to the code generated by GUI builders, however, a data mapping layer consists only of data structures without generated code. It can be composed from prefabricated mapping elements without having to be coded and compiled. The layer can even be stored (e.g. in XML) and replaced at run time.

Decorator Mappings

Data mappings can be plugged together according to the *Decorator* design pattern [2]. This results in a chain of mapping elements, each doing a certain transformation on the data before forwarding them to the next mapping. We call this a *decorator mapping*.

A simple example of a decorator mapping is the *not mapping* (Fig. 3) that implements the boolean `not` operation. It inverts a boolean value during its `get` and `set` operations, i.e. a `set(true)` is forwarded as a `false` value and a `set(false)` is forwarded as a `true` value. If this mapping is applied the view shows the inverted value of the model.

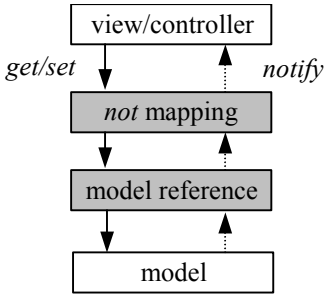

Fig. 3 A data mapping layer consisting of a *not* mapping and a model reference

Other examples of decorator mappings are:

- A mapping that inverts the sign of a number.
- A mapping that computes the absolute value of a number.
- A mapping that converts between numeric values in the model and string values in the view (e.g. `"red"`, `"blue"` and `"green"`)
- A mapping that implements a conditional `set` operation, i.e. it forwards a `set(true)` as a `set(fixedValue)` and ignores a `set(false)`.

Note that some decorator mappings are symmetric (e.g. the *not* mapping or the sign inversion), while others are not symmetric (e.g. the conditional setting of a value) or not reversible (e.g. the computation of the absolute value).

Composite Mappings

Data mappings cannot only be plugged together sequentially (giving a chain of decorators) but also hierarchically (giving a tree of mappings). We call this a *composite mapping*. It is an application of the *Composite* design pattern [2] and allows a view value to be composed from several model values.

For example a date like `"2/15/2003"` can be composed from three numbers in the model representing the month, the day and the year, respectively. A composite mapping connects to multiple child mappings. It forwards a `set(value)` by splitting the value into parts and calling `set(part)` for each child. A `get` operation first gets the parts from the children and computes the combined value that it returns to its father. Notifications from the children are simply forwarded to the father (Fig. 4).

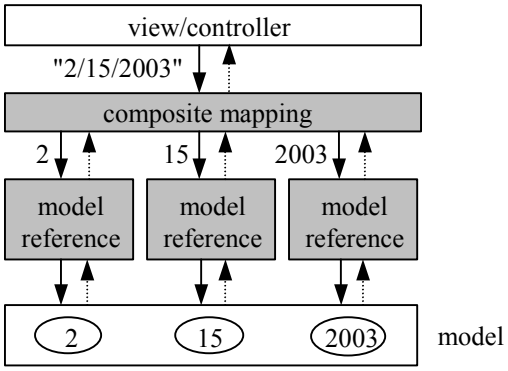

Fig. 4 A Composite mapping computing a date string from numeric values

All composite mappings are derived from a common base class `CompositeMap` with the following interface:

```
public abstract class CompositeMap extends BaseMap {
  public BaseMap[] components;
}
```

Fig. 5 shows a class diagram combining the different kinds of mappings discussed so far.

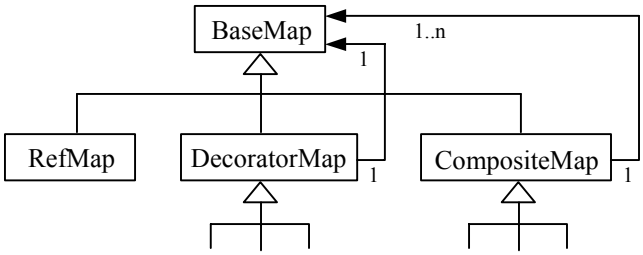

Fig. 5 Class diagram of mapping classes

Further examples of composite mappings include:

- A mapping that computes the sum of several model values.
- A mapping that copies a value from one mapping to another. This can be used to keep several model values synchronized. A `set` operation on the composite mapping `gets` a value from one of its components and `sets` it in the other components.
- A mapping that multiplexes its `set` and `get` operations to one of several other mappings depending on a special control value in the model.

3 An Example

We will now look at an example that shows how the date composition from Fig. 4 can be implemented in Java using the classes described in Section 2. We assume that we have a model that is stored in an object of the following class `Date`.

```
public class Date {
  private int year, month, day; // the variables of the model
  private ChangeListener[] yListeners, mListeners, dListeners;

  public Date(int month, int day, int year) {...}

  public void addChangeListener(String name, BaseMap map) {
    register map as a listener for the field name
  }

  public setYear(int y) {
    year = y;
    foreach (ChangeListener x in yListeners) x.notify(this);
  }
  ...
}
```

The model reference mappings are already available as `RefMap` objects so we don't have to implement them any more. The composite mapping, however, that combines the three parts of a date has to be implemented as a class `DateMap`, say, that is derived from `CompositeMap` and does the conversion.

```
public class DateMap extends CompositeMap {
  public DateMap(BaseMap[] components) { super(components); }

  public void set(Object model, Object data) {
    split data into year, month and day;
    components[0].set(model, month);
    components[1].set(model, day);
    components[2].set(model, year);
  }

  public Object get(Object model) {
    return
      components[0].get(model).toString() + "/" +
      components[1].get(model).toString() + "/" +
      components[2].get(model).toString();
  }
}
```

Now we just have to instantiate and compose the mappings in the following way:

```
Date date = new Date(2, 15, 2003);  // the model
TextField f = new TextField();      // the view
f.setMapping(
  new DateMap(                      // the mapping
    new BaseMap[] {
      new RefMap("month", date),
      new RefMap("day", date),
      new RefMap("year", date)
    }
  )
);
```

In this example the composition is done in Java, but it is easy to imagine that it is done interactively by a builder tool once the mapping components are available. In fact, we are currently implementing such a builder tool that allows us to create and compose arbitrary data mappings from a growing catalog of available mappings.

The implementation of our data mapping framework heavily relies on *reflection*, i.e. on the use of metainformation to access and modify data at run time [3]. Reflection is used, for example, to connect model references to arbitrary variables in the model that can be specified by names or even by path expressions such as

```
x.y[3].z
```

We also use reflection in the implementation of the method mapping that identifies a method by its name. When a method mapping object is created, the method name is resolved using reflection and invoked whenever the `set` operation of this mapping is called.

4 Advanced Uses of Data Mappings

So far we have used data mappings only for simple transformations including the combination of several model values into a single view value. However, data mappings can also be used to accomplish more complex tasks, especially for optimizing the performance of the MVC pattern. Examples of such mappings include *caching*, *polling*, *lazy update*, *concurrency*, or *distribution*, which will be described in the following subsections.

Caching

A *cache mapping* is a decorator that remembers the result of the last `get` request. The next `get` is then answered from the cache. A change notification resets the cache and a `set` operation overwrites the cache contents.

The user may add a cache mapping to the mapping layer if the data retrieval from the model is too inefficient, e.g. if the calculation of the value is time consuming or if the data must be retrieved over a network first. With a builder tool, adding a cache mapping to an existing data mapping is as easy as adding scrollbars to a GUI element.

Polling

Polling allows one to retrieve data from models that do not offer change notifications. This may be useful for prototyped models where no notification mechanism is implemented yet or for debug views, which show internal data that is not intended to be displayed in the production view.

A *poll mapping* never forwards notifications from the model but uses a timer to generate notifications to the view in regular time intervals. The view in response to the notification issues a `get` operation on the mapping in order to retrieve the data.

Lazy Update

Some models change so rapidly that the high number of change notifications can become a burden to the system. This situation can be handled with a *lazy update mapping* that delays change notifications from the model until a certain time has elapsed and then forwards a single notification message to the view. This obviously reduces the total number of notification messages in the system. In contrast to the poll mapping, the lazy update mapping does not send notifications to the view if no notifications arrive from the model.

Concurrency

Modern GUI applications are event-driven, i.e. they wait for input events (keyboard input, mouse clicks, etc.) and then react to them. In order not to block the GUI for too long, every event usually causes a new thread to be started. The main thread just dispatches the events one after the other while the actual processing is done by separate threads in the background.

This behavior can be modeled with a so-called *concurrency mapping*. A concurrency mapping decorates some other mapping and starts a new thread whenever it performs its `get` or `set` operations. Change notifications, however, are immediately forwarded to the event-dispatching thread and are handled without delay, even if the last `get` or `set` operation is not yet finished (Fig. 6).

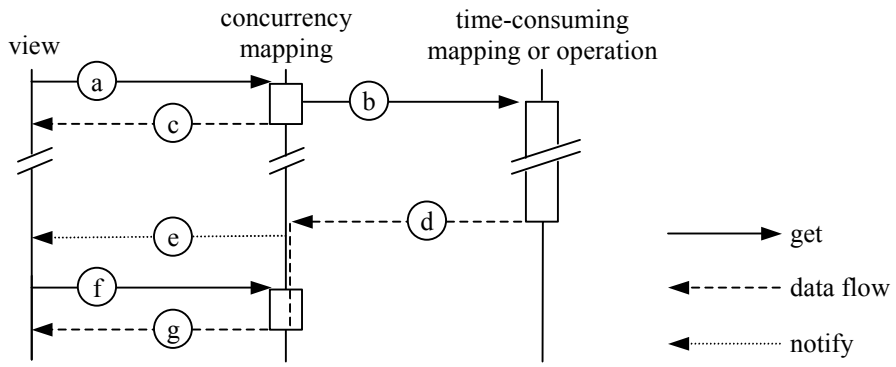

Fig. 6 Trace diagram of a concurrent get operation

While the `set` operation just starts a new thread to update the model and then returns, the `get` operation is a bit more complicated (see Fig.6), because the caller expects a result, but the event-dispatching thread cannot wait. After being invoked (a), the `get` operation starts a new working thread that performs the actual `get` on the model (b) and returns a default value immediately (c), so the calling thread will not be blocked. After the working thread has finished (d), the result remains in the concurrency mapping and a change notification (e) requests a `get` operation (f) to fetch the result (g).

Distribution

In some applications the model and its views reside on different computers. A *distribution mapping* can be used to hide the transfer of data and notifications over a network ([7]). Fig. 7 shows such a situation. The mapping has a server-side part and a client-side part both consisting of several decorator mappings that serialize, compress and encode the data.

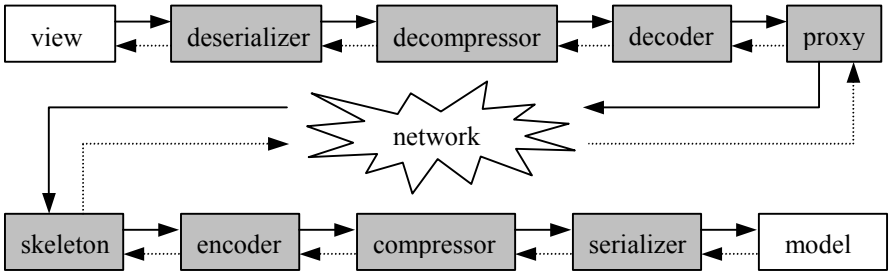

Fig. 7 A data mapping connecting a model on the server with a view on the client

5 Related Work

The MVC paradigm is also known as the *Observer* design pattern [2] where several observers can be attached to a model and are notified whenever the model is changed. The Observer pattern is more general than the MVC pattern because an observer can be any object, not just a graphical view. Similar to the MVC pattern, however, the Observer pattern does not define any mapping of the data or the notifications that flow between the model and its observers.

The same pattern also underlies the *Java event model* [4], where observers are called *listeners*. A listener is a class that implements a certain listener interface, which defines one or several call-back methods for notification. Listeners can be registered at other objects that assume the role of the model. When a certain event happens in the model the notification method of all registered listeners is called so that the listeners can react to the event. A special implementation of the Java event model can be found in the *JavaBeans* standard [6]. Again, however, the Java event model does not have the concept of mappings between an object and its listeners. Of course, a mapping transformation can be implemented by hand (e.g. by installing a special listener with glue code between the model and the actual listener) but this requires manual coding and cannot be automated easily.

GUI builders (e.g. Microsoft's Visual Studio [10], Sun ONE Studio [9] or IBM's Visual Age for Java [8]) allow a user to interactively compose a graphical user interface from prefabricated GUI elements by dragging and dropping them into a window and by modifying their properties. In most cases, however, the GUI elements do not implement the MVC pattern properly because the model data is stored directly in the GUI element, which plays also the role of the view. Thus, the model and the view are not separated and it is not possible to attach several views to a model.

Some GUI builders (e.g. [8]) allow a GUI element to be attached to a separate variable which stores the model. However, this variable is generated by the builder and cannot be chosen to be a field of an already existing model, say. Other GUI builders allow glue code to be inserted at special places in the user interface, but this glue code has to be written by the user manually.

None of the GUI builders known to us has the concept of data mappings that can be plugged together from prefabricated mapping elements without coding. In this respect, our approach is novel and unique.

6 Summary

In this paper we have introduced the concept of a data mapping which is a layer between the model and its views in the MVC pattern. A data mapping is responsible for conciliating a model and its views if their structure is not identical. It can be hierarchically composed from simpler mappings using the Decorator and the Composite design patterns. We have shown how to use data mappings to accomplish data transformations as well as caching, polling, lazy update and concurrent model access.

Most current GUI builders do not allow transformations between a model and its views. Those which allow such transformations require the user to implement glue code by hand. We propose to extend GUI builders so that users can compose a data mapping from prefabricated mapping elements interactively, i.e. without coding. This is in accordance with component-based programming [5] and consistent with the way how a GUI is composed from GUI elements in a builder environment.

We are currently working on such an extended GUI builder which separates a model from its views and allows the user to add a wide selection of data mappings between these two components in a plug and play manner.

References

1. Krasner G.E., Pope S.T.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming (JOOP), August/September (1988) 26-49
2. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns—Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
3. Flanagan D.: Java in a Nutshell, 4th edition, O'Reilly, 2002
4. Campione M., Walrath K., Huml A.: The Java Tutorial—A Short Course on the Basics, 3rd edition, Addison-Wesley, 2000
5. Szyperski C.A., Gruntz D., Murer S.: Component Software, 2nd editions, Addison-Wesley, 2002
6. O'Neil J., Schildt H.: JavaBeans Programming from the Ground Up, McGraw-Hill, 1998
7. Hof M.: Composable Message Semantics in Object-oriented Programming Languages. Dissertation, University of Linz, 2000
8. Visual Age for Java: `http://www-3.ibm.com/software/ad/vajava/`
9. Sun ONE Studio 4 (formerly Forte): `http://wwws.sun.com/software/sundev/jde/`
10. Visual Studio .NET: `http://msdn.microsoft.com/vstudio/`