Proceedings of the IASTED International Conference
Software Engineering and Applications
October 6-8, 1999 – Scottsdale, Arizona – USA

AN ARCHITECTURE FOR A STRICT MODEL-VIEW SEPARATION IN JAVA

EGBERT ALTHAMMER AND WOLFGANG PREE
C. Doppler Lab for Software Research
University of Constance
Campus P.O. Box D188
D-78457 Constance, Germany
E-mail: althammer@acm.org, pree@acm.org

Abstract. Though the separation of a model from its visual representation (view) implies well-known benefits, available Java libraries as Swing so far do not sufficiently support this concept. The paper presents a straight-forward way of how to smoothly enhance Java libraries in this direction, independent of the particular graphic user interface (GUI) library: The lean framework JGadgets, which was inspired by the Oberon Gadgets system [1], allows developers to focus on model programming only. This significantly reduces the development costs, in particular in the realm of quite simple, form-based GUIs which are common-place in commercial client-/server-systems.

We first present a small case study implemented on top of JGadgets which demonstrates the benefits of a strict model/view separation. The paper goes on to sketch the reflection-based design of JGadgets itself.

Keywords: software components, reuse, model/view architecture, reflection, automated configuration, Java

1 MOTIVATION OF JGADGETS

Many commercial applications have a client/server architecture which follows roughly the schematic representation in Figure 1: A client accesses and manipulates data in a data repository called server. The client might be conceptually split into model and view component, where the model corresponds to the particular business logic and the view to the visual representation of the model. The model-view concept and the associated benefits are well-known already from Smalltalk (see [2, 3]). As the structure of the server is not relevant in the context of the paper, we won't discuss this aspect.

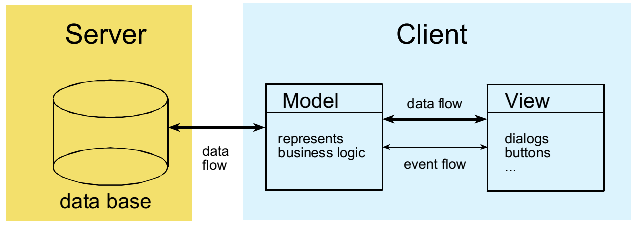

FIGURE 1 SIMPLIFIED ARCHITECTURE OF A CLIENT/SERVER ARCHITECTURE.

Real-world commercial client/server systems usually consist of several dozens if not hundreds of dialogs that form the overall application. The development and enhancement of such applications involves, besides careful data modeling, the tedious task of implementing the dialogs so that end users can enter and edit data. Since Graphical User Interfaces (GUIs) have become popular, numerous tools support the 'drawing' of these dialogs and automate in various ways the development of the client part.

We found that state-of-the-art Java development environments for Java 2 (formerly called JDK 1.2) only partially automate the development of the client part of commercial applications. Though these tools let a developer draw the GUI and generate some event handling code, a clear separation of a model and its view are not pursued. This is partly due to the design of the Java 2 libraries, in particular the Swing GUI library [4]. Swing provides a model-view separation only for its more complex components such as lists and tables, but not for the more simple components.

Typical client/server applications rely on a quite small and well-known set of GUI components: edit fields, labels, action buttons, radio buttons, check boxes, lists/grid controls, and tab controls to name the most common ones. Thus we found that existing Java GUI libraries should be slightly enhanced to support a clear separation of model and views for all components. This should automate the development of the client side. Developers should just focus on working with the models. The view should be generated out of the model descriptions. Overall the model-view separation should yield the following benefits:

- *A simplified development.* Developers should be able to almost ignore the GUI representation. Event handling should also be simplified compared to bare-bone Swing programming.
- *The possibility to switch between different GUI libraries.* The switching, for example between Swing and the Windows Foundation Classes, should not affect the already developed dialogs.

The enhancement of existing GUI libraries should be done in a way that it supports a developer as sketched above. At the same time it should impose as little overhead as possible. The enhancements following these considerations were implemented as a small framework called JGadgets. Analogous to the Oberon Gadgets system, the Java-based model-view framework JGadgets should basically take care that value changes either in the model or in the view should automatically be updated in the other part without any effort by the developer.

JGadgets was developed in a cooperation between the RACON Software GmbH, a software company of the Austrian Raiffeisen banking group and the Software & Web Engineering Group at the University of Constance. RACON applies Java technology together with JGadgets for implementing the client part of various systems, in particular, banking applications.

The next section presents the features and usage of JGadgets from a developer's perspective. A discussion of the core design aspects of the framework rounds out the paper. We assume that the reader is familiar with concepts of object-oriented frameworks as described in [5, 6, 7].

2 REDUCED DEVELOPMENT EFFORT THROUGH MODEL-VIEW SEPARATION—A CASE STUDY

The sample dialog (see Figure 2), which shows the authentic German labeling, allows end users to retrieve information about a bank customer. The tab control supports the selection of various search criteria such as name, personal identification number, account number, and telephone number. In case of a name-based search, the end user enters the last name (text field labeled Name/Bezeichnung), and/or first name (text field labeled Vorname) and/or the date of birth or date when the company started its operation (text field labeled Geb./Grün. Dat.). After pressing the Search (Suchen) button, the list in the lower half of the dialog displays the search results, in this search example, customers with the last name Schwarzenegger.

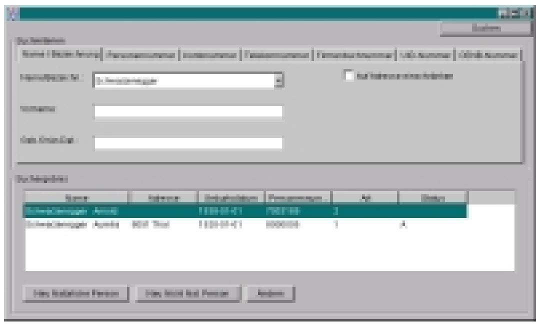

FIGURE 2 SEARCHING FOR CUSTOMERS WITH THE LAST NAME SCHWARZENEGGER.

In order to display or modify the detailed information associated with the customer selected in the list, the end user presses the Modify (Ändern) button. This opens another dialog where the corresponding data can be edited.

MODEL PROGRAMMING IN JGADGETS

A developer who defines a model basically defines what we call *attributes and services.* Attributes store data of a specific type. These types correspond in general to basic Java types, such as integer, float, double and String, but can also be more complex structures such as a lists or edit fields with special formatting such as date or currency fields. Services correspond to GUI elements which trigger action events, such as buttons and menu items. Services and attributes become instance variables of the class that represents the model.

An attribute in the model represents a data container for data (taken from the data base, for instance). It usually has a visual representation in the view. One goal of a model/view separation is that model and view should only be semantically linked. JGadgets requires the developer to adhere to a simple naming convention: An element of the model and an element of the view are associated with each other if they have the same name. The programmer has only to ensure that the names of the instance variables are the same and JGadgets takes care of the linking (see Figure 3).

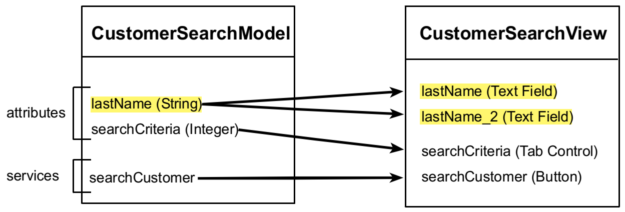

FIGURE 3 NAMING CONVENTION FOR LINKING MODEL AND VIEW.

For example, the attribute with the name *lastName* (of type String) corresponds to the GUI field (text field) with the name *lastName.* If there is more than one visual representation for a model item, a *_2 (_3)* is attached to the name of the view elements (*lastName_2*, …) to distinguish them. This mechanism permits an unambiguous connection of model and view elements.

JGadgets provides the class JGAttribute (note that classes that belong to JGadgets all start with JG…) as the general abstraction for attributes. JGadgets automatically ensures that a change of an attribute's state is reflected in its corresponding view and vice versa.

Besides of the functionality of a data container, an attribute further contains a fixed set of GUI status information. This status information somehow introduces a view flavor into the model. For example, lastName contains the data (String) and general information about the text field, whether it is visible, enabled and whether the focus has been set. Thus, the fact that model elements contain view information seems to undermine the strict model/view separation. We argue that this information belongs to the model, but affects, of course, the user interface. If a service is disabled, neither model nor view (button cannot be clicked) are able to change it directly.

To handle the business logic associated with each attribute, a method can be defined which is invoked when the data of the attribute has been changed. The name of the method is composed by the name of the attribute and the String *Changed* as, for instance, lastNameChanged(). The association is based again on a naming convention.

A service (class JGService) represents a GUI element that does not contain data but just triggers events. Buttons and menu items would be typical examples. To each service there is a corresponding method implemented in the model, which is invoked every time the GUI element is activated. It has the name exec concatenated with the name of that service, for instance, execSearchCustomer().

The event handling in most of the dialogs should be accomplished by these simplified mechanisms. For more sophisticated situations the Java/Swing event handling can be used if necessary.

There are some GUI groups that occur very often, such as list boxes with associated buttons for adding, modifying and deleting list items [8]. They can be described in separate models which can be integrated into other models as submodels. This leads to a treelike model hierarchy.

Models and views can be changed dynamically, i.e., attributes and services can be added and removed at runtime. In this case, a dynamic list administrates the attributes and services. This issue is beyond the scope of this paper and is not described here.

Example 1 sketches the implementation of the CustomerSearchModel (model) which corresponds to the model underlying the dialog in Figure 2. A Model has to implement the empty JGadgets interface JGModel, in order that the JGadgets framework recognizes it as a model. The constructor initializes the attributes searchCriteria and lastName as attributes of type Integer (the JGadgets type corresponding to type int in Java) and String, respectively. Note that the type is passed as a string parameter. The field searchCriteria corresponds to the tab control of the dialog. Each different selection state is reflected through a different value of searchCriteria. It is set to 0 which means that the leftmost panel is selected (labeled "Name/Bezeichnung" in Figure 2). The field lastName together with the method lastNameChanged() correspond to the text field which displays the text "Schwarzenegger" in Figure 2. The service searchCustomer and the related method execSearchCustomer() correspond to the button labeled "Suchen". The field searchResult is of type ListBoxModel, which represents a list box. It is an example of a submodel. The implementation of ListBoxModel is not described here.

```
import JGadgets.*;
public class CustomerSearchModel extends Object
implements JGModel {

  public JGAttribute searchCriteria;
      // corresponds to tab control,
  public JGAttribute lastName;
      // text field,
  …
  public JGService searchCustomer;
      // button 'Suchen',
  …
  public ListBoxModel searchResult;
      // list box
  …

  public CustomerSearchModel () {
    searchCriteria = new
      JGAttribute("Integer");
    searchCriteria.setValue(0);
      // set selection state of the tab-
      // control to the leftmost value

    lastName = new JGAttribute("String");
    …
    searchCustomer = new JGService();
      // the GUI element which repre-
      // sents the service defined
      // in CustomerSearchView
    …
    searchResult = new ListBoxModel();
      // list box with associated buttons
      // (implemented as submodel)
    …
  }

  public void lastNameChanged() {
    // check entered text
  }
  public void execSearchCustomer() {
    // search data on server using the search
    // criteria and display them in the list box
  }
  …
}
```

EXAMPLE 1 A SAMPLE MODEL CLASS.

VIEW GENERATION

JGadgets provides a tool that generates the view out of the model class. The view conceptually contains the GUI elements but not the container of the view. This is the task of the controller.
Analogous to a model class, a view class implements the empty interface JGView. JGadgets introduces GUI elements, such as JGTextField, JGButton, which correspond to the GUI elements of Java GUI libraries

such as Swing. A view only contains JG-components. As JG-components are JavaBeans any Beans editor can be used for editing the visual representation of the dialogs. Example 2 shows source code fragments of the generated class CustomerSearchView. Note that the instance variables searchCriteria, lastName and searchCustomer follow the naming convention, i.e., they have the same names as the corresponding items in the model. Instance variable names with an underscore refer to items of submodels. Here searchResult_listBox refers to the list box which is an attribute (with the name listBox) of the submodel searchResult. The instance variable lastNameLabel is an example of an extra item that has no counterpart in the model. It represents the label of the text field lastName.

```
import JGadgets.*;

public class CustomerSearchView extends Object
implements JGView {

    public JGTabControl searchCriteria;
      // same name as attribute in model
    public JGLabel lastNameLabel;
    public JGComboBox lastName;
      // same name as attribute in model
    …
    public JGButton searchCustomer;
      // same name as service in model
    public JGList searchResult_listBox;
      // refers to submodel searchResult
    …

    public CustomerSearchView() {
        searchCriteria = new
          JGTabControl("Name/Bezeichnung$…")
        lastNameLabel= new JGLabel("Vorname:",
          size and position);
        lastName = new JGComboBox(size and
          position);
        …
        searchCustomer = new JGButton("Suchen",
          size and position);
        searchResult_listBox = new JGList(size
          and position);
        …
    }
```

EXAMPLE 2 A SAMPLE VIEW CLASS.

CONTROLLER

Model and view are both created by a special entity called controller (class JGController). A controller is the manager for models and views and defines the container element wherein the view is displayed. The container can be any type of window such as a dialog, a frame or even a simple panel. Example 3 shows the statement that creates a model and a view and associates it with a controller. A new instance of JGController is created and model and view are passed as String parameters (their class name). The type of container is also passed as a String parameter.

```
JGController customerSearchCtrl = new
   JGController("CustomerSearchModel",
   "CustomerSearchView", "JGFrame",
   <actualController>);
```

EXAMPLE 3 CREATION OF MODEL AND VIEW OF EXAMPLE 1 AND 2.

The last parameter refers to the controller hierarchy. Figure 4 schematically illustrates this aspect. M, V, C are the abbreviations of model, view and controller. The class names next to the circles refer to our case study. The controller *CustomerSearchCtrl* creates the model *CustomerSearchModel* and the view *CustomerSearchView* (Figure 2). When the end user presses the Modify (Ändern) button in the dialog in Figure 2, a subcontroller of CustomerSearchCtrl, *CustomerEditCtrl*, is instantiated which creates the model/view pair *CustomerEditModel* and *CustomerEditView*, which are not described in the paper. In this example it can be seen that the controller hierarchy corresponds to the window hierarchy of the user interface, which means that the controller is responsible for window handling.

Another important aspect of the controller is the role as an information broker between distant model components. If a model wants to send information regarding events to another model the events pass through the controller hierarchy.

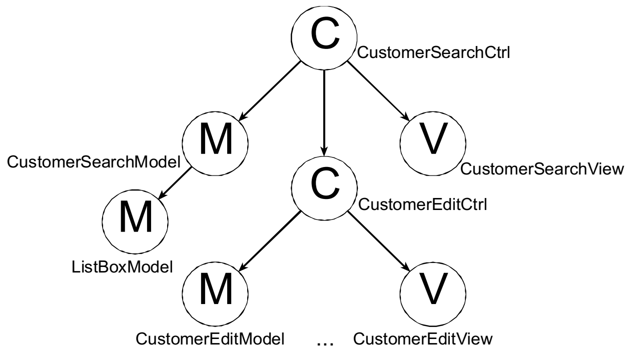

FIGURE 4 CONTROLLER HIERARCHY.

3 CORE DESIGN ASPECTS OF JGADGETS

This section first outlines the static structure of JGadgets, i.e., its classes and interfaces. Based on this description, the design of the connection mechanism between models and views as well as the updating and event handling are discussed.

STRUCTURE OF JGADGETS

The framework JGadgets consists of only a few classes and interfaces (see Figure 5): The class JGController, the two empty interfaces JGModel and JGView, the classes JGAttribute (and subclasses for more complex structures such as lists or tree structures), JGService and the GUI classes, such as JGButton, JGTextField or JGList. Note that the core classes of JGadgets do not extend other Java classes. They are derived directly from the root class Object (see Figure 6).

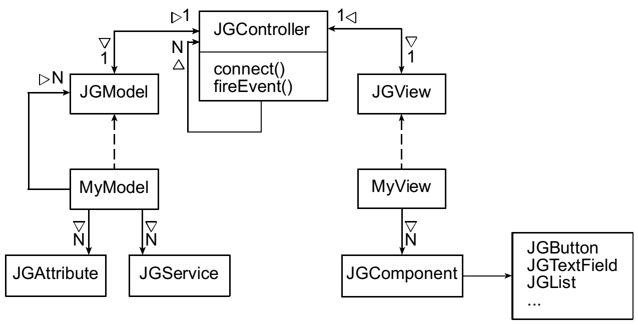

FIGURE 5 SIMPLIFIED CLASS DIAGRAM OF JGADGETS.

JGController is the core part of JGadgets. It has basically two methods: connect(), which does the reflection-based linking of model and view, and fireEvent() which manages the event handling. JGController has a reference to a model and a view (and vice versa) and to (possible) subcontrollers and a parent controller (controller hierarchy). Details of the implementations of the classes of JGadgets are not within the scope of this paper and are not described here.

MyModel and MyView classes are sample model and view classes which have to implement the empty interfaces JGModel and JGView, respectively. This is expressed by dotted arrows in Figure 5. MyModel contains instance variables of JGAttribute, JGService, other models (submodels) and a reference to the parent model (model hierarchy) and MyView contains instance variables of the JG-specific GUI classes.

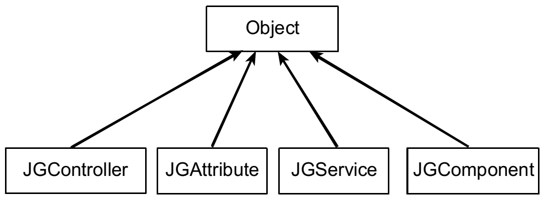

FIGURE 6 CORE JGADGETS CLASSES ARE DERIVED FROM THE JAVA OBJECT CLASS.

JGadgets introduces additional classes for all GUI elements which are required in the dialogs, for example, text fields, labels, buttons, grid controls and numeric spinners. The GUI classes all extend JGComponent and are independent of a special GUI library (as AWT or Swing). The JGadgets components are separated from the Java GUI components through the Adapter pattern [5]. Currently JGadgets supports AWT and Swing GUI components. The design allows a straightforward migration of JGadgets to future Java GUI libraries. As mentioned in the previous section, it is sufficient to adapt the JGadgets GUI components. Model and view remain unchanged.

AUTOMATED CONNECTION OF MODEL AND VIEW

The framework JGadgets defines two abstract components which we call hot spots [6]. These are the (empty) interfaces JGModel and JGView which have to be implemented by the adaptation classes. Since there is no restriction on the programming level for the model and view classes (every class can implement an empty interface), they act together and communicate at a higher, semantic level. In our case this is achieved by means of the simple naming convention explained in Section 2.

In order to connect model and view, the method connect() of class JGController iterates over the attributes and services of the model (and submodels) and the GUI elements of the view and puts them into hash tables. Then model and view elements are checked for name equivalence and corresponding elements get a mutual reference. The model is further sought for attribute or service methods (execServiceName() or attributeNameChanged()).

The method connect() is called when a model/view pair has just been created and also when an attribute or service was added or removed from the model in case of dynamic models. In the latter case only the elements that have been changed are reconnected, in order to save time.

The connection process is illustrated in Figure 7 and refers to our case study. The view has to be connected to the model CustomerSearchModel as well as to the submodel ListBoxModel.

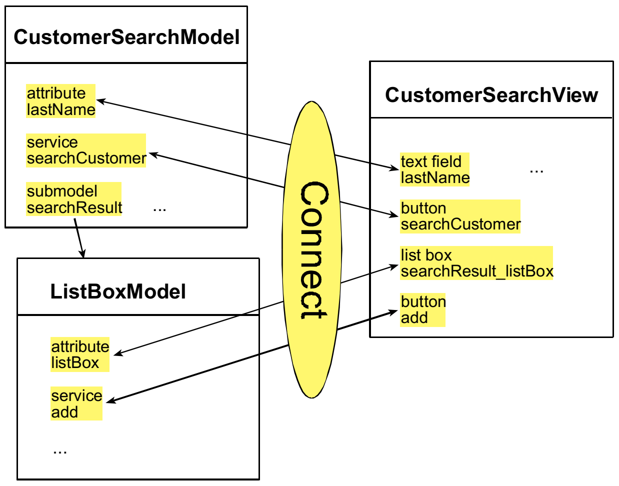

FIGURE 7 A SAMPLE CONNECTION PROCESS.

AUTOMATED UPDATING OF A CORRESPONDING MODEL/VIEW ELEMENT PAIR

If the status of a model or view element has been changed, the counterpart has to be updated. This is easily achieved based on the mutual references between model and view elements, which are created in the realm of the connection process.

The update mechanism is done by a simple assignment:

```
myModelElement.viewElement.set<Property>
    (myModelElement.get<Property>)
```

The important aspect of the update is that in case that the view element has been changed by the user, the model has a veto right if the entered value has not been correct for whatever reason. The model then restores the previous state.

A numeric text field (class JGTextField) in the view with the name myNumericField corresponds to the Integer attribute myNumericField in the model. If the number is changed in the model (by invoking the method myTextField.setValue(12345)), the change is propagated to the view where the new number is displayed in the corresponding GUI field. The view informs the model and the model invokes the method myTextFiebldChanged(). If, on the other hand, a number is entered into the text field, an attempt is made to update the model element. If the value entered is a valid number, the associated method myTextFieldChanged() is invoked, otherwise (if a letter is typed into the numeric field, for instance) the model refuses the update and restores the previous state.

AUTOMATED EVENT HANDLING

JGadgets provides an event handling mechanism which relies on the Java event listener model [9] and which is managed by the method fireEvent() of JGController. Whenever a model element is changed, it triggers an event, which other models can listen to. An event moves up and down the controller hierarchy so that submodels and parent models can handle it (see Figure 4). In order to do this the method fireEvent() is implemented recursively and is executed for all the parent controller and subcontrollers to find other models which listen to the event.

A model which listens to events has to implement the according interfaces. JGadgets introduces a series of new events, such as ModelChangedEvent (an attributed has been added or removed from a model), AttributeChangedEvent (any attribute has been changed) and ViewActivatedEvent (the view window has got the focus).

4 SUMMARY AND RELATED WORK

There are numerous other systems that provide a model/view separation and that automate the synchronization of these two components as well as the event handling in the GUI. For example, the Microsoft Foundation Classes provide a mechanism for dynamic data exchange (DDE). [10] describes a system that applies reflection to some extent in the realm of synchronizing model and view components. The Oberon Gadgets system was already mentioned as another example originating from the academic community.

The goal of JGadgets is to augment Java libraries in a small and convenient manner. Overall, we feel that the design and implementation of the framework architecture at hand is reasonably small: The bytecode size of JGadgets, including the JG-GUI components is around 40 KB. The linking based on naming conventions ensures that developers benefit from JGadgets without being bothered by the framework.

REFERENCES

[1] Gutknecht J (1994) Oberon System 3: Vision of a Future Software Technology. Software—Concepts & Tools 15, 1

[2] Goldberg A, Robson D (1985) Smalltalk-80 / The Language and its Implementation. Addison-Wesley

[3] Krasner G and Pope S (1988). A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1(3):26-49, August/September 1988

[4] Eckstein R, Loy M, Wood D (1998). Java Swing. O'Reilly

[5] Gamma E, Helm R, Johnson R and Vlissides J (1995). Design Patterns—Elements of Reusable Object-Oriented Software. Reading, Massachusetts: Addison-Wesley

[6] Pree W (1996). Framework Patterns. New York City: SIGS Books (German translation, 1997: Komponentenbasierte Softwareentwicklung mit Frameworks. Heidelberg: dpunkt)

[7] Fayad M, Johnson R and Schmidt D (1999) Object-Oriented Application Frameworks. Wiley.

[8] Pree W, Althammer E and Sikora H (1998) Self-Configuring Components for Client-/Server Applications. IEEE Workshop on Large Components, DEXA'98, Vienna, Austria, August 1998

[9] Campione M, Walrath K (1999) The Java Tutorial Second Edition. Object-Oriented Programming for the Internet. Addison Wesley

[10] Hewitt C (1998) Developing Business Object-based Applications in JBuilder (http://www.oop.com /white_papers/java/business_objects.htm)