DESIGN AND IMPLEMENTATION OF A MVC-BASED ARCHITECTURE FOR E-COMMERCE APPLICATIONS

E. Althammer and W. Pree

C. Doppler Lab for Software Research
University of Constance
Campus P.O. Box D188
D-78457 Constance, Germany

althammer@acm.org, pree@acm.org
www.SoftwareResearch.net

Abstract. Though the separation of a model from its visual representation (view) implies well-known benefits, available Java libraries do not sufficiently support this concept. The paper presents a straightforward way to smoothly enhance Java libraries in this direction independently 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 e-business-systems.

We first present a small case study implemented on top of JGadgets which demonstrates the benefits of the MVC architecture. The paper then goes on to sketch the reflection-based design of JGadgets itself.

Keywords: software components, reuse, MVC architecture, reflection, automated configuration, Java

1 Introduction

Many commercial applications, in particular e-commerce applications, have a client-server architecture which follows roughly the schematic representation in fig. 1: a client accesses and manipulates data in a data repository which is stored on the server. The client might be conceptually split into model and view components, where the model corresponds to the particular client side business logic and the view to the visual representation of the model. The server consists of the application server which contains the server side model and the data repository which contains the data base. The paper focuses on the model-view aspects of the client and leaves out the server part.

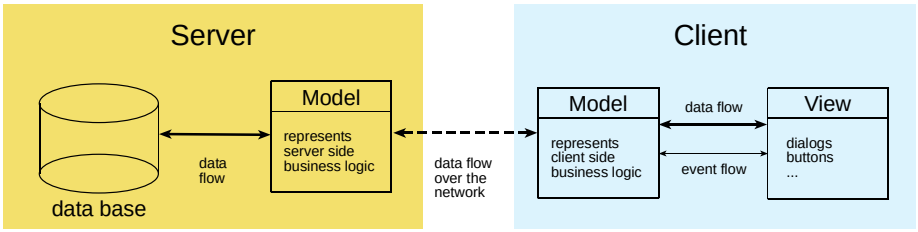

Figure 1 Simplified diagram of a client/server architecture.

Clients in real-world commercial client/server systems usually consist of several dozens if not hundreds of dialogs that form the overall application. The development and en-

hancement of such applications involves, besides careful data modeling, the tedious task of implementing the dialogs so that end users can view, enter and edit data. Since Graphical User Interfaces (GUIs) have become popular, numerous tools support the 'drawing' of these dialogs and automate the development of the client part in various ways.

We found that state-of-the-art Java development environments for Java 2 (formerly called JDK 1.2) [2, 3] only partially automate the development of the client part of such 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 is not pursued.

Model-view separation is crucial when a client server application has to serve different clients with different characteristics, e.g. a desktop client has a large screen and a strong processor whereas the screen of a PDA (personal digital assistant such as a Palm Pilot) or a mobile phone is rather small. Since the application has to be suited to the different clients (depending on the characteristics and capabilities of the device) there is, generally speaking, a separate version of the client application for each device which has to be implemented and maintained.

By separating the views from the model, we can save development and maintenance charges by just having *one* version of the model (since the model is the same for each client type) and an own version of the view for each device. Thinking one step further the views can be automatically generated out of the model descriptions so that the development costs are minimized.

A framework that supports model-view separation should thus foster a single model for different views. For the developer this means that he or she just focuses on working with the models and the framework generates the views out of the model descriptions.

The GUI libraries underlying the views depend on the used platform as well as on the type of the JVM (java virtual machine). E.g., a desktop client (which uses Java 2 standard edition) may rely on Swing or AWT and a PDA client (which uses the Java 2 micro edition) may rely on the KVM GUI library [4]. These libraries are incompatible among each other and cannot be easily exchanged. To allow a switch of platforms, the framework should therefore encapsulate the different GUI libraries. Overall the model-view separation should yield the following benefits for the developers:

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.

Different platforms should be supported. The switch of the platform and thus the employed GUI library should not affect the already developed dialogs or models.

A better reuse of components. Developers should be able to define and reuse model components independent of their visual representation.

The model-view separation should support a developer of client-server 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 of features, such as synchronizing model and views as well as automating the event handling.

JGadgets was developed in a cooperation between the RACON Linz Software GmbH (short RACON), a software company of the Austrian Raiffeisen banking group

and the Software Research Lab at the University of Constance. RACON applies Java technology together with JGadgets for implementing the client part of various systems, in particular, Internet banking applications for desktop PCs and mobile phones.

JGadgets is derived from the original MVC (model-view-controller) architecture which is described in [5, 6] but differs in several points. The following subsection evaluates the most important characteristics of the traditional MVC architecture and contrasts it with the JGadgets architecture.

1.1 JGadgets versus the Traditional MVC Architecture

MVC assumes that a model does not know which views are plugged to it. As a consequence, it allows multiple views on the same model and a decoupled change of the views. JGadgets makes a different approach: the model knows about the existence of its view(s) and about how the views are organized (i.e. the view hierarchy). The motivation behind this is that models are normally not designed without having in mind how the views are going to look like. On the other hand, the decoupling is even stronger than in MVC. Models and views do not have to know the other's interface. The connection is done dynamically based on a naming convention. This approach allows a more flexible connection of model and view.

The MVC architecture provides a strict separation of view and controller. This, however, might introduce an extra complexity to the system. Controllers cannot be fully decoupled from the views because they have to understand the events triggered by the view. Thus, each view needs its special controller. The idea is to give up the strict MVC separation and to integrate view and controller into a single component. Modern GUI libraries such as Swing which are based on an MVC architecture [7] do this. For instance, the Swing list has two classes, JList which corresponds to view and controller and ListModel which corresponds to the model. Swing provides a model-view separation for most of its components. JGadgets does not combine view and controller but provides a generic implementation of the controller as part of the framework which automates the basic MVC operations, such as the event propagation, the connection, synchronization and disconnection of models and views.

The MVC architecture in general reduces performance. Each update of the model causes an update of the views and a user event causes the controller to contact the model. This is critical when model and view are distributed over a network, e.g. as in Web-based client server applications. Thus model-view interaction has to be brought to a minimum, respectively a part of the model has to be kept on the client side, as illustrated in fig. 1.

The MVC architecture is based on the existence of only a single model and does not specify how different models should be organized within an application. JGadgets extends the MVC paradigm by introducing a controller hierarchy that administers the models and the interactions between them.

MVC fosters a transparent reuse of model components independently of their visual representation: a typical model element is a list box with the associated buttons for adding, modifying and deleting list items as well as an editor dialog for editing them [8]. The logic of the list box (list handling, buttons) is quite simple and thus can be easily reused. The view of the list box, however, can vary significantly: the list view could be a single- or multi-column list or a combo box; the buttons could be right aligned or bottom aligned. A button could be left out or exist twice, and so on. By separating model

and view a developer can reuse the (black box) model component but is not restricted to use a certain view template. Analogously, view and controller components can be defined. Due to the model hierarchy of JGadgets reusing of model components gets even simpler.

In summary, the important aspects of JGadgets are the automated linking mechanism between models and views, the generic controller and the scalability of these components.

The next section presents the features and usages of JGadgets from a developer's perspective. A discussion of the core design aspects of the framework is presented in Section 3. We assume that the reader is familiar with the core concepts of object-oriented frameworks as described in [9, 10, 11].

2 Reduced Development Effort Through Model-View Separation—A Case Study

The sample dialog (see fig. 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 the 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.

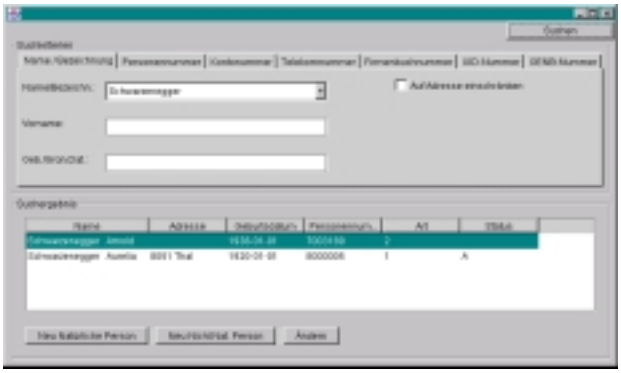

Figure 2 Searching for customers with the last name Schwarzenegger.

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 (see fig. 3).

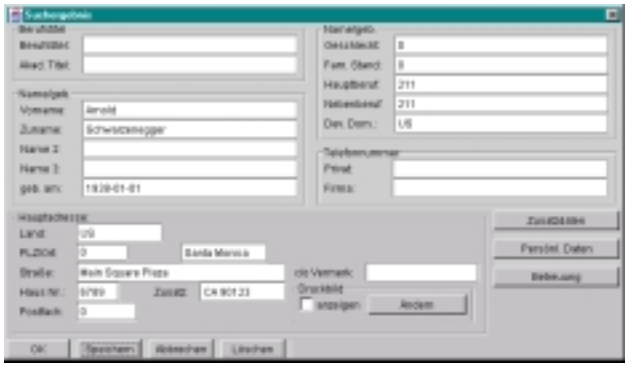

Figure 3 Editing customer data.

The views illustrated in fig. 2 and 3 are rather complex and are often subject to change. If the application is ported to a device with a small display (e.g. a Palm Pilot) the views have to be separated into several single views. A model-view separation makes this process significantly easier.

The next subsection describes how the developer composes an application with JGadgets by means of the case study of fig. 2 and 3.

2.1 Overview

The developer only implements the model part of the application. The view is generated out of the model descriptions and is edited with a GUI editor. The controller is part of the framework. Hence, the major part of the following subsections is dedicated to model programming. The view part describes the view generation and the controller part describes the controller API used by the model programmer.

2.2 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 which corresponds to any Object type in Java. This could range from basic Java types, such as Integer, Float, Double and String, to more complex structures such as lists or formatted fields (date fields, currency fields, etc.).

Services are model entities which do not contain data but describe self-contained operations of the business logic. Input and output parameters of services are stored in attributes. E.g., a service *searchCustomer* performs a search for customer data on a database using search criteria. The search criteria are specified in a set of String attributes and the result (the customer data which matches the criteria) is stored in a list attribute. Attributes and services become instance variables of the class that represents the model.

2.3 Attributes and Services

JGadgets requires the developer to store data exclusively in attributes in order to keep it synchronized to the views. A model developer has to understand the API of attributes and services because they represent the skeleton of each model. That is the reason why we start with the properties and methods of attributes and services. The other features of JGadgets, such as event handling, rely on these concepts.

2.3.1 Implementation of Attributes

JGadgets provides the class JGAttribute as the general abstraction for attributes (note that classes that belong to JGadgets all start with JG…). JGAttribute is implemented as a Java bean and has a set of properties which are listed in table 1.

Table 1

Properties of an attribute: bound properties are marked with *, read only properties with **

Property	Type	Description
Data Field		
data*	Object	holds actual data
type	Class	Allowed data type
dataOld**	Object	keeps a backup of the data
MVC Field		
controller**	JGController	Reference to the controller
views**	Vector	Reference to the view elements
Accessibility Field		
enabled* visible* focus*	Boolean	Accessibility flags

Note that some properties are bound which means that JGadgets automatically notifies and updates the corresponding view elements when that property has changed. Bound properties are marked with a *. Properties marked with ** are read only. They are automatically set by JGadgets.

The properties can be grouped into three categories, the data field, the MVC field and the accessibility field.

The ***data field*** manages the data contained in the attribute. The *data* property holds the actual data. Note that primitive types are not allowed in JGadgets. Instead, wrapper classes such as Integer are used. The *type* property (of type Class) specifies which data type can be stored in the attribute. Whenever an attempt is made to change the data property, a type check is performed: only data which is an instance of the specified type (thus also sub-types) or a string that contains the string representation of a compatible data type is accepted; incompatible data is refused. E.g., an integer attribute only accepts integers or strings that contain an integer value. The property *dataOld* keeps a backup of the data.

Note that the only difference to “normal” Java programming is that the single instance variables are wrapped into the JGAttribute objects. JGAttribute itself is never sub-classed, the variation of the attribute is implemented in the *type* property which can be *any* valid java class. Thus all existing Java classes can be used. The data is accessed with the JGAttribute’s method *getData()*. Since this method returns an object of type Object, the result has to be casted to the correct type. Data changes must be implemented with the data’s setter method *setData()* to start the JGadgets update mechanisms. For complex data changes it is recommended to make a local copy of the data, perform the changes on the copy and invoke *setData()* only once.

The ***MVC field*** is responsible for maintaining the relationships between models, views and controllers. Each attribute has a reference to the controller object (*controller* property) and to the corresponding view gadgets (*views* property). These properties are set by JGadgets automatically and are read-only.

The ***accessibility field*** describes the accessibility status of the encapsulated data of an attribute. Information such as *enabled*, *visible* and *focus* somehow introduces a view flavor into the model. 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 business logic and thus to the model. The business logic specifies the actual state of an attribute, for example if it is accessible. This is expressed by the attribute's flag *enabled*. JGadgets takes care that the status is correctly displayed by the view. This forces the developer to describe the business logic more carefully. A second reason for putting these properties into the model is to keep multiple views synchronized.

Typical GUI representations of attributes would be text fields and combo boxes (String, numeric attributes), lists/tables (list/table structures), labels (Strings), check boxes and sliders (Boolean, Integer).

2.3.2 Implementation of Services

JGadgets uses the class JGService as an abstraction for services. Services have the same properties as attributes but lack the data field (see table 1). This is reasonable because services do not contain data. The JGService class is also not sub-classed.

A service is activated in the model with the service's method *perform()* which is implemented in the JGadgets framework. It does not contain the implementation of the service but is only the starting mechanism. It checks whether the service is active (e.g., the flag *enabled* must be set *true*) and triggers a JGadgets action event where the event is encapsulated in an object of type ActionEvent. The implementation of the service is contained in the corresponding event handling method (see subsection 2.5). The reason for this is that services can also be activated in the view, e.g. when the user clicks on the corresponding view element (button or menu item). Since the activation in the view is event based, this mechanism allows a uniform handling of services.

2.4 Naming Convention to Link Models and Views

JGadgets uses a naming convention to automatically link the models to the views:

Model and view elements are associated with each other if they have the same name.

The name refers to the name of the instance variable of an attribute or service in the model class and a GUI element inside the view class. The programmer has only to ensure that the names of the instance variables are the same and JGadgets takes care of the linking (see fig. 4).

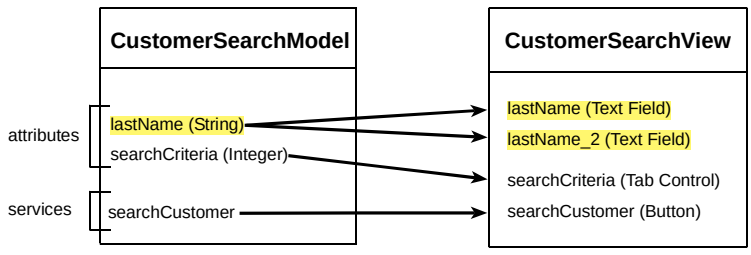

Figure 4 Naming convention for linking model and view.

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 names of the view elements

(*lastName_2, …*) to distinguish them. This mechanism permits an unambiguous connection of model and view elements. The link between model and view is done once when models and views are instantiated.

The advantages of the naming convention are evident. The developer does not need to explicitly set links between a model and its views. This is done by JGadgets at runtime. Hence, the correctness of the binding cannot be statically checked by the Java compiler. For this reason JGadgets provides a verification tool which does exactly this. If the view is generated, the corresponding names for the view elements are automatically set.

The naming convention makes two assumptions about the model. First there must be a model element for each view element whose data content exactly corresponds to the data the view should display. Thus the model programmer has to have in mind which elements could be displayed in the view. Thus models may have to define attributes which depend on others just to satisfy all view elements.

In simple cases – if the view field is not editable – JGadgets could be extended to support so called *view combination fields* through an extended naming convention: Consider the case that a text element of the view should display the full name of a person. The model contains, however, only the two string attributes, *firstName* and *lastName*. Thus the view element would display a string containing the content of the two attributes plus a white space in between. Instead of defining a dependent model attribute which contains both names, the view name could be like *firstName_SPACE_lastName*. In this case JGadgets automatically links the two model elements to one view element and updates it whenever *firstName* or *lastName* is changed.

For attributes which contain a complex data structure (e.g. customer data) and whose data elements should be described in several view elements JGadgets provides an elegant solution with the use of sub models. If the attribute is formulated as a sub model (where the single data elements are defined as attributes and services, see subsection 2.6), JGadgets parses it recursively, extracts the attributes and services and connects them accordingly to the view.

The second assumption is that attributes or services with the same name mean the same thing and are therefore connected to the same view fields. The developer has to carefully select the names of attributes and services to avoid the connection of wrong elements. The standard name for different kinds of attributes and services should be part of the documentation of the application. It will also happen that more than one attribute of the same kind is used in the application. This would result in a wrong association. JGadgets provides a solution for sub models but does not come up with a general solution.

2.5 JGadgets Events

2.5.1 Listening to JGadgets Events

JGadgets provides a simplified event handling for standard events, for example when a bound property of an attribute is changed or a service is started. These so-called JGadgets events differ from the standard Java events in the sense that the event listener needs not to register with the event source but is automatically bound if it implements a method, the so-called event handling method, whose name matches the following naming convention:

An event handling method which handles events triggered by a certain attribute or service is composed by the name of the attribute or service and a string which characterizes the event.

E.g., the event that describes the change of the data property of an attribute is called *dataChanged*. That means that the event handling method of the data-changed-event of the attribute *lastName* has the name *lastName_dataChanged(..)*. Note that a "_" is put between the names to clearly separate attribute and event names. Other event handling methods are defined by combining a property with -Changed, such as *focusChanged*. The activation of a service is handled inside the method <name>+_+<actionPerformed>, hence *searchCustomer_actionPerformed(..)* (see fig. 5). The event handling methods can optionally have one parameter which contains the Java event object. In the case of the *dataChanged* event the parameter would be an object of type *java.beans.PropertyChangeEvent* and in the case of the *actionPerformed* event an object of type *java.awt.event.ActionEvent*.

The event handling methods are defined in the model where the event is handled. This can either be the model where the event source is defined but also other models. JGadgets automatically notifies them and invokes the event handling method. Note that the event handling methods are *only* invoked by the framework.

2.5.2 Triggering JGadgets Events

JGadgets automatically generates a JGadgets event when a user makes an action with the mouse or keyboard, for instance pressing a button or entering text into a text field. Further a JGadgets event is triggered when the model programmer invokes the *perform()* method of a service.

If the model programmer wants to generate JGadgets events manually and/or needs other events than those specified above, he or she generates them with the method *fireEvent()* of JGController. This method will be explained in subsection 2.8 under point 3.

2.5.3 Other Types of Events

Data Events: JGadgets supports events generated by the data property of an attribute other than the standard *dataChanged* event. For instance, the data property of a list attribute *listBox* generates an event when list items have been selected. These events are implemented as regular Java events and the controller works as event adapter. Thus it has to register with the model. (For this reason JGController implements all the necessary listener interfaces.) The model implements the event handling method which is composed by *data* and the event type such as *selectionChanged*, thus *listBox_dataSelectionChanged(..)*. Thus, in the case of a list attribute, the normal Swing list model can be used.

General Model and Controller Events: General model events do not refer to a specific attribute or service but to general changes of the model. They are handled with *this_<eventName>()*, for example *this_dataChanged(..)* (= the data of any attribute in this model has changed), *this_focusChanged(..)* (= any model element has changed the focus). Controller events are events which are handled by the controller. These are, for instance, standard events referring to the window hierarchy, such as the *windowClosing* event. When this event is triggered, the controller automatically closes the associated view(s) and sub-view(s) and disconnects the view elements.

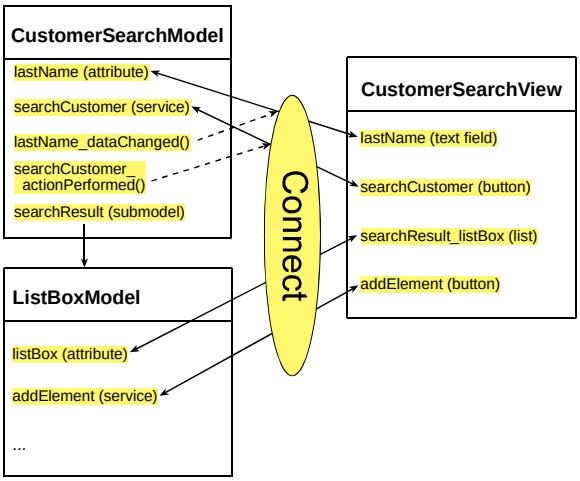

Figure 5 A sample connection process

2.6 Further Issues of Model Programming

2.6.1 Reuse of Model Components – Sub Models

JGadgets fosters the reuse of model components independently of their visual representation. A typical model building block would be a list box with associated services for adding, modifying and deleting list items. Such building blocks are described in separate models and are integrated into other models as sub models. Sub models are implemented exactly like normal models, hence any model can be used as a sub model inside a parent model. Sub models become instance variables of the parent model and their attributes and services are connected as the attributes and services of the parent model. The name of the view element is either equal to the name of the attribute or – to avoid possible interferences – is composed by the name of the sub model and the attribute name: <*nameOfSubModel*>_<*nameOfAttribute*>. E.g., the sub model named *searchResult* contains a service *addItem*. Thus the name of the corresponding view element is *addItem* or *searchResult_addItem*.

Fig. 5 illustrates the connection of the list box as a sub model. The model of the list box is implemented in the class *ListBoxModel*. It is integrated into the model CustomerSearchModel where it gets the name *searchResult*. The view CustomerSearchView contains the visual representation of the list box model.

2.6.2 Dynamic Models

Normal models are implemented statically, thus attributes and services are instance variables of the model class. JGadgets also offers dynamic models which allow adding and removing attributes and services at runtime. These models administer the elements in dynamic lists and implement the interface *JGDynamicModel* which contains methods for adding and removing model elements. Dynamic models are used, for instance to read tables of a relational data base. The columns correspond to the attributes that are dynamically added. When a model changes, it has to be reconnected to the view. This is automatically handled by JGadgets.

2.6.3 Model Implementation Example

Example 1 sketches the implementation of *CustomerSearchModel*, which corresponds to the model underlying the dialog in fig. 2. A Model has to implement the empty JGadgets interface *JGModel*, in order to be recognized as a model by the JGadgets framework. The constructor initializes the attributes *searchCriteria* and *lastName* as attributes of type Integer and String, respectively. Note that the type is passed as a Class parameter. The field *searchCriteria* is used to control the tabbed pane of the dialog. Each different selection state is reflected through a different integer value of *searchCriteria*. If it is set to 0, it means that the leftmost panel is selected (labeled "Name/Bezeichnung" in fig. 2). The field *lastName* together with the method *lastName_dataChanged(PropertyChangeEvent e)* contains the last name of a customer. The service *searchCustomer* (and method *searchCustomer_actionPerformed(ActionEvent e)*) performs a data base query on the base of the attributes such as *lastName* stores the result in *customerData* and puts it into the attribute *searchResult* which is of type *ListBoxModel* and represents a list box. It is an example of a sub model (implementation, see Example 2). It contains items of type *CustomerData*.

```
package jgadgets.*;

import jgadgets.listBox.*;
import java.awt.event.*;
import java.beans.*;

public class CustomerSearchModel implements JGModel {

  public JGAttribute searchCriteria;        // manages the tab control of the view
  public JGAttribute lastName;              // stores the last name of a customer,
  …
  public JGService searchCustomer;          // searches a customer from the data base
                                            // and displays the result in the
  …
  public ListBoxModel searchResult;         // list box (implemented as sub model)
  public CustomerData[] customerData;       // stores the customer data which match
                                            // the query (sub model)
  …

  public CustomerSearchModel () {
    searchCriteria = new JGAttribute(Integer.class);       // short constructor
    searchCriteria.setData(0);              // sets selection state of the tab control
                                            // to the leftmost value

    lastName = new JGAttribute(String.class);
    …
    searchCustomer = new JGService();
    …
    searchResult = new ListBoxModel();      // initializes the list box
    searchResult.itemType.setData(CustomerData.class);
                                            // sets the type of the list items
    …
  }

  public void lastName_dataChanged (PropertyChangeEvent e) {
    // checks the entered text
  }
  public void searchCustomer_actionPerformed (ActionEvent e) {
    // perform a search on the data base
    // result is stored in customerData
    …
    searchResult.listBox.setData(customerData);     // fill the list box with the data
  }
  …
}
```

Example 1 A sample model class.

The list box model (Example 2) is a standard component that can be reused. It contains an attribute *listBox* which stores the list items. The list functionality including the selection of items is defined in an element of type *javax.swing.DefaultListModel*. Note that this is just an example and any other list model class could be used. The *listBox* attribute triggers a data event and thus the controller has to register. This is done in the (optional) method *init()* because at the time the constructor is invoked the controller is not defined (see subsection controller, task 1). The corresponding event handling method is *listBox_dataSelectionChanged(ListDataEvent e)*. The attribute *itemType* which contains a Class object specifies the type of the list items. The list box has services for adding (*addItem* is shown), modifying and removing list items.

```
package jgadgets.listBox;

import jgadgets.*;
import javax.swing.*;

public class ListBoxModel implements JGModel {

  public JGAttribute listBox;              // stores the list items
  public JGAttribute itemType;             // specifies the type of the list items
  public JGModel editDialog;               // model where items can be edited

  public JGService addItem;                // adds a list item
  …
  public ListBoxModel () {
    listBox = new JGAttribute(DefaultListModel.class);      // uses the swing list model
    itemType = new JGAttribute(Class.class);
    addItem = new Service();
    …
  }

  init() {                                 // the controller registers with the list box
    ((DefaultListModel)listBox.getData()).addListDataListener(listBox.getContoller());
  }

  public listBox_dataSelectionChanged(ListDataEvent e) {
    // handles changes of the list selection state
  }

  …                                        // other event handling methods
}
```

Example 2 The list box model.

2.7 View Generation

The view describes the presentation layer and contains a static description of the GUI items. Note that the view does not contain any business logic.

JGadgets encapsulates the existing GUI libraries, such as Swing, AWT, and the GUI components for the KVM library by introducing a new set of GUI classes, which start with JG-, such as JGTextField or JGButton. The JG-GUI-components contain the necessary methods for the automated model-view connection as well as for the JGadgets event handling. The view only contains JG-components and is independent of the underlying GUI library.

For a faster view development JGadgets provides a tool that supports the generation of the view out of the model descriptions: the user chooses the JG-components for the

attributes and services and the tool generates the source code of the view. By means of a GUI editor the user places the GUI widgets at the correct position.

The view elements are placed into a container which can be a dialog, frame, applet or even a simple panel. This is *not* specified by the view itself but by the controller. The reason for this is the possibility to use the same view in different containers.

Views are hierarchically organized in sub views, similar to sub models. Sub views are placed into JGPanels which are part of the parent view. Note that since the container element is not specified in the view, every view can be used as sub view of another view.

Views do not need to show all the GUI elements to the user at the same time. Instead they can be organized dynamically, for example with a tabbed pane (see fig. 2). The control of the dynamic part is typically managed by an extra integer or string attribute which indicates the dynamic state of the view. In our example we used the integer attribute *searchCriteria*.

Example 3 shows source code fragments of class CustomerSearchView which was generated out of class CustomerSearchModel. 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 sub models. Here *searchResult_listBox* refers to the attribute *listBox* of the sub model *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 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();
        searchCriteria.setLabel("Name/Bezeichnung…");

        lastNameLabel= new JGLabel();
        lastNameLabel.setLabel("Vorname:");
        lastNameLabel.setBounds(size and position);

        lastName = new JGTextField();
        lastName.setBounds(size and position);
        …
        searchCustomer = new JGButton();
        searchCustomer.setLabel("Suchen");
        searchCustomer.setBounds(size and position);
        searchResult_listBox = new JGList()
        searchResult_listBox.setBounds(size and position);
        …
    }
```

Example 3 A sample view class.

2.8 Controller

The JGadgets controller (class JGController) is a generic implementation that is independent of the specific view. The following section explains the functionalities of the controller which can be grouped into four tasks.

1. Instantiation and connection of model and view
2. Managing the application hierarchy
3. Generation and propagation of JGadgets events
4. Exchanging information between models

Instantiation and connection of model and view: A model creates a new model/view pair with the static method *connect()* as illustrated in Example 4. The first two parameters of the method represent the model and view which are to be created. These can either be specified with the class (either as String or Class) or with an object. In the first case the corresponding objects are instantiated dynamically.

```
JGController customerSearchCtrl = JGController.connect("CustomerSearchModel",
                                                       CustomerSearchView.class,
                                                       new JGFrame(),
                                                       <parentController>);
```

Example 4 Instantiation and connection of the model/view pair of example 1 and 3.

Since this statement is invoked by the model, it assumes that the model knows about the existence of the views which are connected to it (see second parameter in Example 4; it is possible to connect more than one view by invoking *connect()* more than once). The model needs, however, not to know about the implementation of the views. JGadgets connects them dynamically on the basis of the naming convention. This is the strength of JGadgets.

The third parameter of the method indicates the view container object which is one of the JGadgets container widgets. If JGFrame or JGDialog is specified, a new window for the view is opened. If the parameter is of type JGPanel, the new view is placed as sub view inside the view of the actual model.

With the creation of a model/view pair also a new controller maybe instantiated, depending on the needs of the business logic. This is specified with the fourth parameter. If it is not null, it defines the position of the new controller within the existing controller hierarchy by specifying its parent controller (see also task 2 and fig. 6).

The result of the connection process is that each model element has a reference to its corresponding view elements and vice versa. Both model and view elements get a reference to the controller.

Closing a model works the same way using the method *disconnect()*. This method disconnects the elements of the specified model and view and removes the model, the view, the controller and all the child elements - if there are any - from the application hierarchy.

Managing the application hierarchy: The architecture of the overall application is defined by the relationship of the controllers instances and the attached models and views. Fig. 6 schematically illustrates this aspect. M, V, C are the abbreviations of model, view and controller and the class names next to the circles refer to our case study. The controller *CustomerSearchCtrl* holds the model *CustomerSearchModel* and the view *Cus-*

tomerSearchView. When the end user presses the Modify (Ändern) button in the dialog in fig. 2, a new dialog is opened. For this purpose a new model/view pair (*CustomerEditModel* and *CustomerEditView*) is created together with a new controller *CustomerEditCtrl* which is a child of the first controller.

The controller hierarchy (especially the graphic version as in fig. 6) gives us a good insight into the current state of the application. If we only take into account the models and controllers, we see the relevant business logic and the relationships between the models. If we leave out the models (left hand), we get the window hierarchy.

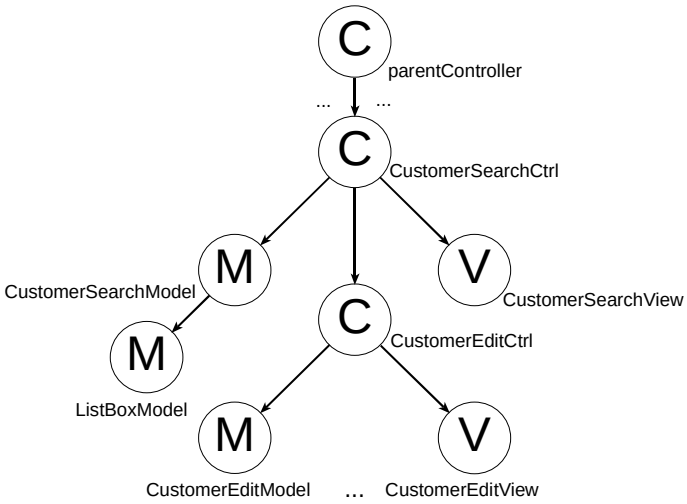

Figure 6 Controller hierarchy.

Generation and propagation of JGadgets events: The controller provides a method *fireEvent()* to generate JGadgets events (see Example 5). This method has two parameters: the first is the event type encapsulated in an event object. This event corresponds to a Java event object such as a *PropertyChangeEvent*. The event object is instantiated manually (as in the example) or is taken over from a Java GUI event (e.g. an action event). The second parameter is a constant that specifies the event propagation inside the controller hierarchy. Available constants are JGController.DEFAULT for default propagation (first the local model, then downward and upward), JGController.LOCAL only the local model, JGController.DESCENDING only downward and JGController.ASCENDING only upward.

```
fireEvent(new PropertyChangeEvent(<attribute>, "data", <oldValue>, <newValue>),
            // type of event
            JGController.DESCENDING);
            // controls event propagation (controller constant)
}
```

Example 5 Sending an descending property-change event for the *data* property

The controller looks for models which implement the according listener methods which would be *<attributeName>_dataChanged(PropertyChangeEvent e)* in case of example 5.

The event propagation is defined with the controller constants but can be controlled by the event handling methods: with the use of a return parameter of type Boolean which indicates whether the event handling should be continued (value *true*) or termi-

nated (value *false* or no return parameter). In the case that the event propagation terminates no other model is considered for event handling.

Note that the return parameter is not implemented in the sample model implementations in example 1 and 2. Fig. 7 illustrates the event handling process of the action event of the service *addItem* of *ListBoxModel*. The service triggers the action event with default propagation (①) which is handled by the model's default event handling method *addItem_actionPerformed(ActionEvent e)*. As it returns *true* (②), the event moves to the child controllers and the corresponding models (③). The next model in question is the model *CustomerEditModel* which implements the event handling method. The method is invoked by the controller and returns *false* so that event handling process to terminated (④).

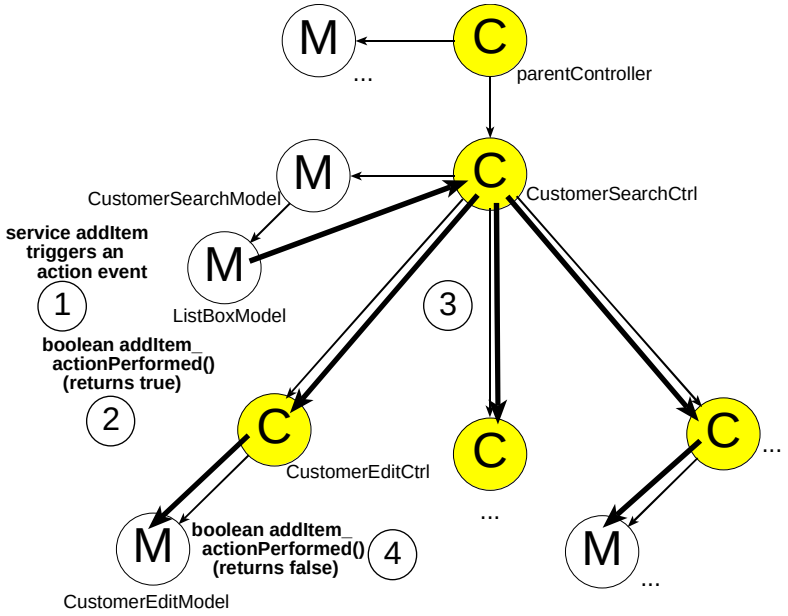

Figure 7 Propagation of JGadgets events

Exchanging information between models: A model that needs information of elements from another model, does not access that model directly but uses the controller method *getModelElem(<nameOfElem>, <model>)*. The *nameOfElem* parameter refers to the name of the model element (attribute, service or sub model). The *model* parameter refers to the model and can be specified as a String or Class object. If a model parameter is specified, the model element is sought there, if it is null, the element is sought globally. The method returns the references of the found attribute and service as enumeration. The returned model elements are accessed and changed like a local model element. In larger applications each model may contain an ID to be referenced unambiguously. This has not been implemented yet.

3 The Architecture of JGadgets

This section first outlines the components of JGadgets, i.e., its classes and interfaces. Based on this description, the core design aspects of JGadgets such as the connection and update mechanisms between models and views as well as the encapsulation of the GUI are discussed.

3.1 Components of JGadgets

The framework JGadgets consists of only a few classes and interfaces (see UML diagram in fig. 8): The class JGController, the two empty interfaces JGModel and JGView, the classes JGAttribute, JGService and the GUI classes, such as JGButton, JGTextField or JGList which are sub classes of JGComponent. Note that the core classes of JGadgets do not extend other Java classes. They are derived directly from the root class Object (see UML diagram in fig. 9).

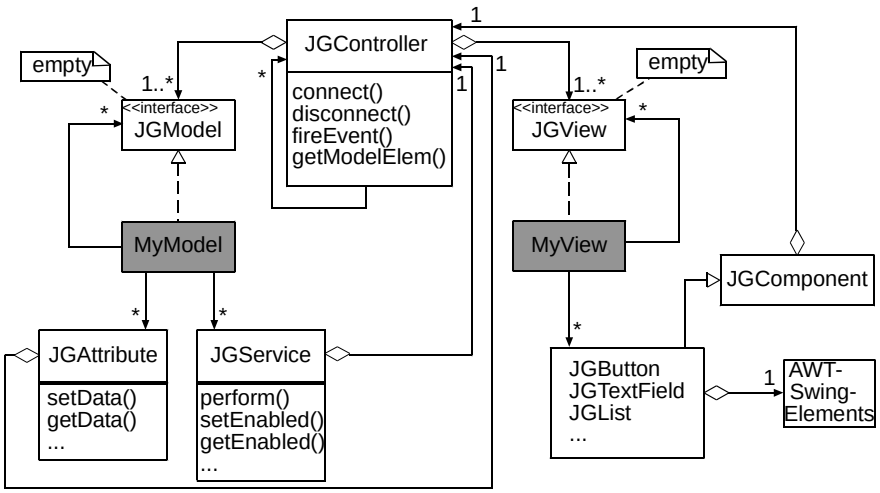

Figure 8 Simplified class diagram of JGadgets.

JGController is the core part of JGadgets. It has basically four methods: *connect()*, *disconnect()*, *fireEvent()* and *getModelElem()*. JGController has a reference to the models and views its manages, to its sub controllers and to its parent controller to maintain the controller hierarchy.

The framework JGadgets defines two abstract components which we call hot spots [8]. These are the (empty) interfaces JGModel and JGView which have to be implemented by the adaptation classes which are the models and the views. They are represented by MyModel and MyView in fig. 8. MyModel contains instance variables of JGAttribute, JGService and sub models. MyView contains instance variables of the JG-GUI classes as well as sub views. Note that the model and view itself do not have a reference to the controller. A model (view) accesses its controller by using the controller reference of any model (view) element (i.e. JGAttribute, JGService and JGComponent)

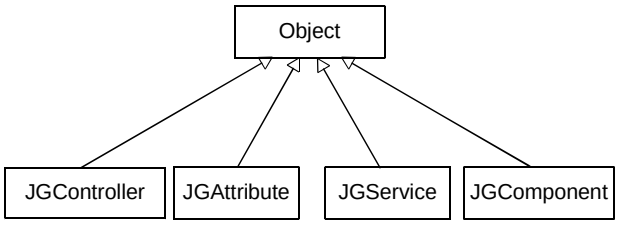

Figure 9 Core JGadgets classes are derived from the Java Object class.

JGadgets defines the JG-GUI elements which are required for the views, such as text fields, labels, buttons and grid controls. The JG-GUI classes extend JGComponent and encapsulate the corresponding elements of the used GUI library. Currently JGadgets

supports AWT and Swing[1]. The design allows a migration of JGadgets to future Java GUI libraries.

3.2 Automated Connection of Model and View

The *connect()* respectively the *disconnect()* method is invoked by JGadgets whenever there is a change of the controller hierarchy and when a model or view has been added or removed. When an attribute or service has been added to or removed from a dynamic model, the method connects the added or disconnects the removed attributes and services.

To connect a view to the model, the method iterates recursively over the attributes and services of the model and sub models as well as over the GUI elements of the view, extracts the names with reflection, checks model and view elements for name equivalence and sets a mutual reference for corresponding elements. For performance reasons model and view elements, as well as the event handling methods are put into hash tables with their names as the key.

Corresponding elements are checked for type compatibility. If the check is positive, the state of the view element is updated with the state of the model element; otherwise, an error message pops up and the elements are disconnected; e.g. an integer attribute and a list view are not compatible.

3.3 Automated Synchronization of Model and View Elements

Whenever a property of a model or view elements has been changed, its counterpart is automatically updated. If the property is bound, additionally a JGadgets property change event is trigged by the model element. Fig. 10 illustrates the synchronisation of the data property of the attribute *lastName*. Note that this is automatically done by JGadgets.

If the user changes the text of the text field *lastName*, the view field handles the Java text event (see also subsection 3.4) and makes an attempt to update the model by invoking *lastName.setData(newValue)* where *newValue* is the new value of the text field (①). The attribute validates the new value (②). If it is compatible, the view(s) is/are updated (③). Otherwise an error message pops up and the view is reset. In case of a successful update, the model element generates a JGadgets event (*dataChanged* event) by invoking the controller's *fireEvent()* method (④). The controller invokes the corresponding event handling method *lastName_dataChanged()* of the model (⑤). Note that the parameters are omitted for simplicity.

If the data property of the attribute is changed in the model, a validation of the new value is also necessary because the model developer could have assigned a wrong value to the attribute (②). Further the model element might not be active (property *enabled* = false). In neither case the model would be updated.

[1] The KVM which is an implementation of the Java 2 Micro Edition for the Palm Pilot does not contain the reflection API used in JGadgets. Thus JGadgets in its current implementation is not portable to a Palm Pilot device.

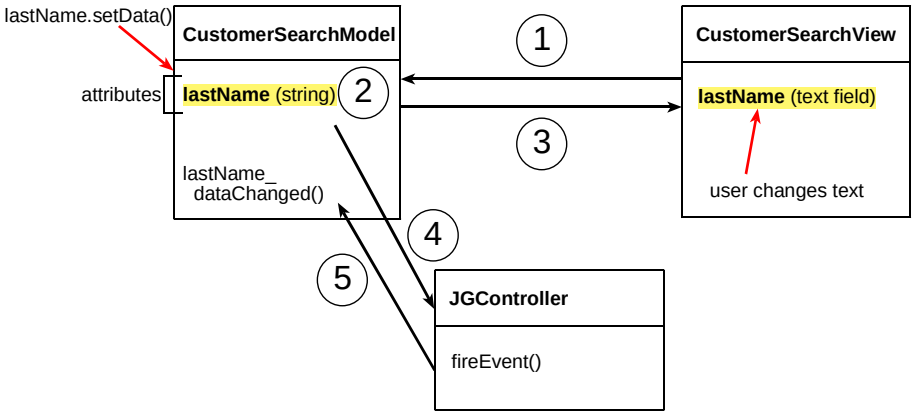

Figure 10 JGadgets Event Handling

Note that the automation is mainly implemented in the *setData()* method of the attribute which performs the type check, updates the views and triggers the dataChanged event. The controller method *fireEvent()* which is reflection-based looks for the event handling method using the internal hash tables, invokes the method and decides, depending on the return parameter, whether to continue or terminate the event handling process.

3.4 GUI Encapsulation

The GUI components of JGadgets encapsulate the used Java GUI library. The components start with JG- and extend the class JGComponent which is a descendent of Object. For each GUI library there exists a separate set of JG-components with the same functionality but different implementations. For example there is a JGTextField for AWT and Swing. The JGadgets components are separated from the Java GUI components through the Adapter pattern ([9] and fig. 11). Clients which rely on different GUI libraries thus use a different set of JG-GUI components. When a newer version of a Java GUI library is used, it is sufficient to adapt the JGadgets GUI components. Model and view remain unchanged.

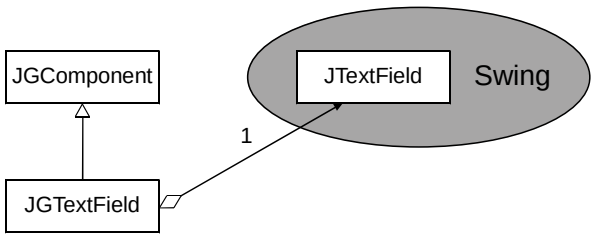

Figure 11 Applying the Adapter pattern for providing uniform JGadgets component interfaces

Generally speaking, the GUI elements must at least support the properties specified in table 1 in order to be synchronized with the attributes. They do because the parent class JGComponent implements the necessary methods. For some properties, such as *enabled* the default implementation of JGComponent is sufficient. For others, such as the *data* property, the JG-GUI components have to override the methods. Example 6 sketches the implementation of the class JGTextField for the Swing library (Example 6). The constructor, the method *get-* and *setData()*, which have to be overridden, as well as the necessary listener methods are shown.

The constructor instantiates the GUI element (a *javax.swing.JTextField*) in the field *guiComponent* and registers the JG-GUI element as a document listener. The *data* property is the text contained in the text field. It is accessed by the setter- and getter method which adapt the getter- and setter methods of the Java GUI element (*setText()* and *getText()*). The event handling methods for document events, such as *changeUpdate(DocumentEvent e)*, update the model element by invoking its *setData()* method.

```
package jgadgets;

import javax.swing.*;

public class JGTextField extends JGComponent implements DocumentListener {


  public JGTextField() {
    guiComponent = new JTextField();        // variable inherited from JGComponent
    ((JTextField)guiComponent).addDocumentListener(this);
  }

  public void setData(Object data) {        // set the text of the text field
   if (data != null)
     ((JTextField)guiComponent).setText(data.toString());
  }

  public Object getData() {                 // get the text of the text field
    return ((JTextField)guiComponent).getText();
  }

  public void changeUpdate(DocumentEvent e) {
           // event handling method, set the attribute data with the new text
    ((JGAttribute)this.getModel()).setData(getData());
  }

  public void insertUpdate(DocumentEvent e) {
    // the same implementation as in event handling method "changeUpdate"
  }

  public void removeUpdate(DocumentEvent e) {
    // the same implementation as in event handling method "changeUpdate"
  }
}
```

Example 6 JGTextField for the Swing text field

4 Conclusion

4.1 Summary

The goal of JGadgets is to augment Java libraries in a lean and convenient manner. JGadgets allows

- MVC programming using an arbitrary GUI library by using GUI-independent views
- Linking of models and views based on naming conventions.
- A simplified event handling mechanism for standard events based on naming conventions.
- while it does *not* limit the usage of existing Java classes.

Overall, the design and implementation of the framework architecture at hand is reasonably small: The byte code size and LOC of JGadgets, including the JG-GUI compo-

nents is summarized in table 2. The linking based on naming conventions ensures that developers benefit from JGadgets without being bothered by the framework.

Table 2

Size of JGadgets

JGadgets	**Size of byte code**	**LOC**
whole framework	80 KB	~5500
core classes: JGController JGAttribute JGService JGModel, JGView	25 KB	~2000
JG-GUI classes for AWT	25 KB	~1500
JG-GUI classes for Swing	30 KB	~2000

4.2 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). [12] 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.

4.3 Outlook

The JGadgets views only provide a static descriptions of the GUI. Thus we investigate in a more appropriate view description format based on XML. Such descriptions could be useful in the realm of small mobile devices where a full fledged JVM is not implemented, as is the case with the KVM. The migration to XML is done on top of JGadgets' naming convention. The view description is then a set of XML fields containing the information of the view gadgets. The device-dependent view would be generated with adaptation tables similar to the JG-GUI elements.

References

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

[2] Sun Microsystems, *Java 2*, http://java.sun.com/jdk

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

[4] Sun Microsystems, *KVM*, http://java.sun.com/products/kvm

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

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

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

[8] W. Pree, E. Althammer, Framelets as Architectural Building Blocks—Concepts & Case Study, *ST '98*, Paderborn, Germany, 1998

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

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

[11] M. Fontoura, W. Pree, B. Rumpe, *The UML Profile of Framework Architectures* (Addison-Wesley, 2000)

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