Stijn Gysemans

Preface

Who should read this book

How this book is organised

Documentation conventions

Community

Feedback and support

Acknowledgements

Many thanks to Filip Croes, Jonas Scheers, Jeroen de Keyser and Ulli Matiz for correcting and reviewing the document.

1. Xml2gui

1.1. Introduction

Xml2Gui is a framework that can be used to create your swing interface in XML. Graphical user interfaces are ideal to represent in XML because of its hierarchical structure. On top of that, xml2GUI includes packages for validation, data (both xml data and object data are supported) and layout.

The base ~~contains~~ consists of one xml-file and the controller. As you can see, we have embraced the MVC[1] pattern. Now there is a clean separation between the view, the controller and the model. The view is our xml-file, the controller is a java file and the model is an xml or object data source.

This project was first created as a school project but was later extended. The first release was in June 2004 and the second major release will be in June 2005.

1.1.1. Simple hello world

Let's create a "hello world" in our xml2gui. First, we'll make a xml-file called hello.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<frame>
    <label text="Hello world"/>
</frame>
```

We also need a controller class. This class is essential, because it initialises the xml-file. It extends the JFrame class.

```
package tss.xml2Gui.demos.manual.helloworld;
```

[1] Model View Controller: Patterns of Enterprise Application Architecture by Martin Fowler

```
import tss.xml2Gui.XmlToGui;
import tss.xml2Gui.customJComponents.DataForm;
import javax.swing.*;


public class Controller extends DataForm{
    public Controller(){
        new XmlToGui(this).convert(this.getClass().getResource("hello.xml").
toString());
        this.setSize(100,100);
        this.setVisible(true);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {
        new Controller();

    }
}
```

If we run the controller class, we should now see the Hello World on you screen! (Figure 1: Hello World) Looks great, huh!

Figure 1: Hello World

The constructor creates an instance of Xml2gui class. It gives the current frame as a parameter. Afterwards it uses the method “convert” in which you specify the place of the xml-file.

As you can see, the xml-file is clearly legible and understand. Normally we use JLabel and JFrame in swing environments but actually, here also! The parser uses a TagLibrary to search for the correct JComponent. Almost all the standard JComponents are already registred into the tag library such as JLabel and JTextField, but they are registered under “Label” and “Textfield”. That’s why we don’t use the prefix.

Text is a “property” of JLabel. When the parser detects an attribute, it automatically invokes the appropriated property. This means: we don’t have to learn a new language or framework. We still use swing but in another much better way!

1.1.2. Some basic interaction

Let’s go on by creating a small program that reacts on user response. The program displays the name we have filled out in a textbox.

The xml is fairly straightforward.

```
<?xml version="1.0" encoding="UTF-8"?>
<frame title="User interaction" layout="flowlayout">
    <textfield id="txtInput" columns="30"/>
    <label id="lblOutput"/>
    <button id="btnDisplay" text="Display"/>
</frame>
```

For the controller has the main part remained the same, but we have added three local variables.

package tss.xml2Gui.demos.manual.basicInteraction;

```
import tss.xml2Gui.customJComponents.DataForm;
import tss.xml2Gui.XmlToGui;
```

```
import javax.swing.*;
import java.awt.event.ActionEvent;

public class Interaction  extends DataForm{
    public JButton btnDisplay;
    public JTextField txtInput;
    public JLabel lblOutput;

    public void btnDisplay_click(ActionEvent e){
        lblOutput.setText(txtInput.getText());
    }
    public Interaction(){
        new XmlToGui(this).convert(this.getClass().getResource
("interaction.xml").toString());
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(400,400);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new Interaction();
    }
}
```

As you can see, This is all you have to do. There is no action listener you need to implement. Now there is a clean separation between the UI en the controller.

The public variables listed above are “linked” with the XML-file automatically through the parser. If you look thoroughly you can see that we use the ID property as the connection. Id is an extra attribute which you can now add to each component in the xml-file. This should be unique.

1.2. Some background

1.2.1. FindComponent method.

If you don’t like the public variables you can create the controller in another way. We have to modify the controller class slightly . We create a new local variable called Parser.

```
package tss.xml2Gui.demos.manual.findComponent;

import tss.xml2Gui.customJComponents.DataForm;
import tss.xml2Gui.XmlToGui;
import tss.xml2Gui.Parser;

import javax.swing.*;
import java.awt.event.ActionEvent;

public class Controller  extends DataForm{
    public JButton btnDisplay;
    private Parser parser;


    public void btnDisplay_click(ActionEvent e){
        JLabel lblOutput = (JLabel)parser.findComponent("lblOutput");
        JTextField txtInput = (JTextField)parser.findComponent("txtInput");
        lblOutput.setText(txtInput.getText());
    }
    public Controller(){
        XmlToGui xml2gui = new XmlToGui(this);
        xml2gui.convert(this.getClass().getResource("gui.xml").toString());
        parser = xml2gui.getParser();
```

```
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(400,400);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new Controller();
    }
}
```

1.2.2. InitListener

Now we still have got to get rid of the public button. If we simply remove the button as a public field, the method can't be invoked. So we need to work in the old way: action listeners. Another problem rises: if we add the addActionListener to the constructor of the class, the program will display a NullPointerException. This is because between the call to the constructor and the end of the parsing, we are not sure if the button has already been initliased. This is the reason why you have to implement the initlistener. The method in this interface is called after the parsing has been done.

```
package tss.xml2Gui.demos.manual.initListener;

import tss.xml2Gui.customJComponents.DataForm;
import tss.xml2Gui.data.InitListener;
import tss.xml2Gui.data.InitEvent;
import tss.xml2Gui.Parser;
import tss.xml2Gui.XmlToGui;

import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class Controller extends DataForm implements InitListener{

    private Parser parser;
    public Controller(){
        XmlToGui xml2gui = new XmlToGui(this);
        xml2gui.convert(this.getClass().getResource("gui.xml").toString());
        parser = xml2gui.getParser();

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(400,400);
        this.setVisible(true);
    }

    public void init(InitEvent evt) {
        super.init(evt);
        JButton btn=(JButton)evt.getDataForm().getParser().findComponent
("btnDisplay");
        btn.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                btnDisplay_click(null);
            }
        });
    }

    public void btnDisplay_click(ActionEvent e){
        JLabel lblOutput = (JLabel)parser.findComponent("lblOutput");
        JTextField txtInput = (JTextField)parser.findComponent("txtInput");
        lblOutput.setText(txtInput.getText());
    }
    public static void main(String[] args) {
        new Controller();
    }

}
```

In the init method, we first call the parent method (used in the dataform). We still can't use the local field parser because it is still null. That's why we get the parser reference from the initEvent parameter.

1.2.3. The converters

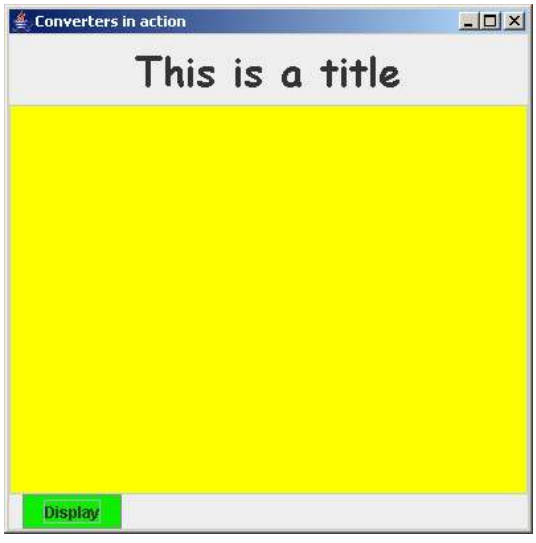
We have already used String properties as attributes (such as title) but what about integers, doubles, layouts and other data types? Xml2gui includes a number of standard converters. You can add you own by extending the class ParameterConverterBuilder.

How does Xml2gui know to which type ~~h~~ite needs to convert? He looks at the method (derived from the attribute) and finds the type of the parameter. With this type it can find the correct converter in the converter library.

Data type	**Converter**
Boolean	PrimitiveConverter
int	PrimitiveConverter
double	PrimitiveConverter
Float	PrimitiveConverter
String	StringConverter
Dimension	DimensionConverter
LayoutManager	LayoutConverter
Color	ColorConverter
Font	FontConverter
Icon	IconConverter
Image	IconConverter
Object	ObjectConverter

This is a view of all the converters in action:

```
<?xml version="1.0" encoding="UTF-8"?>
<frame title="User interaction" layout="borderlayout" size="400,400"
defaultCloseOperation="JFrame.EXIT_ON_CLOSE">
    <panel layout="boxlayout(BoxLayout.Y_AXIS)"
constraints="BorderLayout.CENTER">
        <textfield id="txtInput" columns="30" background="yellow"
enabled="false"/>
        <label id="lblOutput" />
        <button id="btnDisplay" text="Display" background="#10ee00"/>
    </panel>
    <panel constraints="BorderLayout.NORTH">
        <label text="This is a title" font="Comic Sans MS-BOLD-30"/>
    </panel>
</frame>
```



1.3. Layout

1.3.1. LayoutManagers

Off course there is no gui without a layout. Xml2Gui uses the stardard swing classes to create your lay out. The lay out is always managed by a jcomponent or container. You have to add the attribute layout to the parent and constraints attribute to the children.
See the previous xml file for a demo!

Current supported lay outs.

- Borderlayout
- Flowlayout
- Gridlayout
- Cardlayout
- Boxlayout.

1.3.2. Css

We all know css from webdesign. Here in Xml2gui, we also apply this for swing!

We use standard css files that you can edit in Ddreamweaver[2]. You can easily link you xml-file to an xml file:

```
<dataform title="Supplier" size="600,600"
defaultCloseOperation="JFrame.EXIT_ON_CLOSE" css="layout.css">
```

You can add a class to a component by adding the attribute cssClass

```
<pnlSearchComponent id="searchComponent" cssClass="searchlist"/>
```

```
.searchlist{
                    margin:10px;
}
```

Currently the following css tags are supported

css	**Swing equivalent**
Margin	Empty border
Font-size	Font is resized
Text-align	Component is align. If it's a label, the method setHorizontalAlignment is used.

The support for css will be extended in the future!

1.4. Data

1.4.1. Introduction

Xml2gui has built in support for 2 data types: xml and object data. For object data it currently has only support for our own persistency framework. There are a lot of built in classes to support the edit of data. In fact, you don't need to write any code to link your data to the gui! The most important class is the DataSource class. I'll first give an example of an object data source.

[2] Dreamweaver can be found on www.macromedia.com

1.4.2. Basic binding

In this manual we'll work with a Customer class. Here is the source listing:

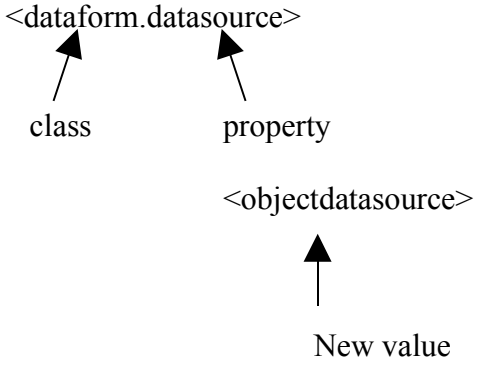
```
package tss.persistancy.demo.businessLogic;

import java.util.List;



public class Customer extends tss.persistancy.DomainObject{
    private String name;
    private String taxnumber;
    private String street;
    private String number;
    private String city;
    private String zipcode;
    private String email;
    private List<ContactPerson> contactPersons;

    public List<ContactPerson> getContactPersons() {
        if(contactPersons ==null){
            contactPersons = getHis(ContactPerson.class);
        }
        return contactPersons;
    }

    public void addContactPerson(ContactPerson p){
        p.setCustomer(this);
        getContactPersons().add(p);
    }
    public void setContactPersons(List<ContactPerson> contactPersons) {
        this.contactPersons = contactPersons;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Customer(String name, String taxnumber, String street, String number,
String city, String zipcode,String email) {
        this.name = name;
        this.taxnumber = taxnumber;
        this.street = street;
        this.number = number;
        this.city = city;
        this.zipcode = zipcode;
        this.email = email;
    }
    public Customer(){}
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getTaxnumber() {
        return taxnumber;
    }

    public void setTaxnumber(String taxnumber) {
        this.taxnumber = taxnumber;
    }

    public String getStreet() {
        return street;
```

```
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getZipcode() {
        return zipcode;
    }

    public void setZipcode(String zipcode) {
        this.zipcode = zipcode;
    }

}
```

The xml needed to use this class in a the gui is as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<dataform title="customer" size="600,600"
defaultCloseOperation="JFrame.EXIT_ON_CLOSE">
    <frameResource.dataSource>
        <objectDatasource name="customer"
className="tss.persistancy.demo.Customer"/>
    </frameResource.dataSource>
</dataform>
```

We first need to explain a feature of xml2gui that we haven't explained yet. If you have complex types as properties and you haven't supplied a converter, you can use the "."-syntax to set the property.



The object data source class needs the attribute "class" to function properly. This is your domain class that you want to edit or display. You can also give a name. For the moment this doesn't mean anything because there is currently only support for one data source at the same time.

Notice that we don't use the JFrame class, but instead, we use the DataForm class which supplies our special needs for handling data.

Before we can actually display something on the screen, we have to specify the mappingschema and connection information. (For more information on the Persistaency framework, consult the persistency framework manual.) We do this in the controller.

```
package tss.xml2Gui.demos.manual.data;

import tss.xml2Gui.customJComponents.DataForm;
import tss.xml2Gui.XmlToGui;
import tss.persistancy.Configuration;
import tss.persistancy.DefaultMappersFactory;
import tss.persistancy.ConnectionInfo;

import javax.swing.*;

public class Simple extends DataForm{
    public Simple(){
        initDatabase();
        new XmlToGui(this).convert(this.getClass().getResource("simple.xml").
toString());
        this.setVisible(true);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    }
    public static void main(String[] args) {
        new Simple();
    }

    public void initDatabase() {
        Configuration.setMappersFactory(new DefaultMappersFactory() {
            public String getPath() {
               return
"persistancy\\src\\tss\\persistancy\\demo\\MappingSchema.xml";
            }
        });
        Configuration.setConnectionInfo(new ConnectionInfo(){
            public String getDatabaseName() {
                return "eindwerkJava_demo1";
            }

            public String getPassword() {
                return "sql";
            }

            public String getPort() {
                return "1433";
            }

            public String getSqlServer() {
                return "localhost";
            }

            public String getUserName() {

                return "stijn";
            }
        });
    }
}
```

Now we'll start of by displaying a simple list of all our records.

```
<?xml version="1.0" encoding="UTF-8"?>
<dataform title="customer" size="600,600"
defaultCloseOperation="JFrame.EXIT_ON_CLOSE">
    <frameResource.dataSource>
        <objectDatasource name="customer"
className="tss.persistancy.demo.businessLogic.Customer"/>
    </frameResource.dataSource>
    <scrollpane>
        <scrollpane.viewportview>
            <listRepeater  id="lstList">
                <listrepeater.visualDataSourceManager.itemStyle>
```

```
                    <label text="*bind(name)"/>
                </listrepeater.visualDataSourceManager.itemStyle>
            </listRepeater>
        </scrollpane.viewportview>
    </scrollpane>
</dataform>
```

Again, there are a few new things in the xml. We have used a list inside a scroll pane to display our data.

The list is a special class called ListRepeater. This class has a property VisualDataSourceManager. That class is responsible for displaying the correct data to the user. We can use three different styles: itemStyle, alternateItemStyle and selectedItemStyle. Right now, we only use ItemStyle.

As a style, we specify a jComponent, a label for example. The text is bound to the data source using a special syntax.

```
*bind(<bindname>)
```

<bindname> is actually just a property of our business class. We can't only use name but also street, number, city,...

1.4.3. Display data in form-way

Now let's create a form in which we can display the data.
The xml is as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<dataform title="customer" size="300,200"
defaultCloseOperation="JFrame.EXIT_ON_CLOSE">
    <frameResource.dataSource>
        <objectDatasource name="customer"
className="tss.persistancy.demo.businessLogic.Customer"/>
    </frameResource.dataSource>
    <panel layout="gridlayout(0,2,10,10)">
        <label text="id"/>
        <textfield text="*rebind(id)"/>
        <label text="name"/>
        <textfield text="*rebind(name)"/>
        <label text="number of orders"/>
        <textfield text="*rebind(numberOfOrders)"/>
        <label text="city"/>
        <textfield text="*rebind(city)"/>
    </panel>
</dataform>
```

If you now start ~~up~~ the application, you'll see that the first customer is automatically displayed and the fields are filled in correctly.

It shows the first record because this is the default setting after opening a data form. You can override it by adding the following line of code to the init method of your controller class. Don't forget to implement the InitListener interface!

```
public class Form extends DataForm implements InitListener{
 …
public void init(InitEvent evt) {
    super.init(evt);
    this.getDataSourceManager().last();
}
```

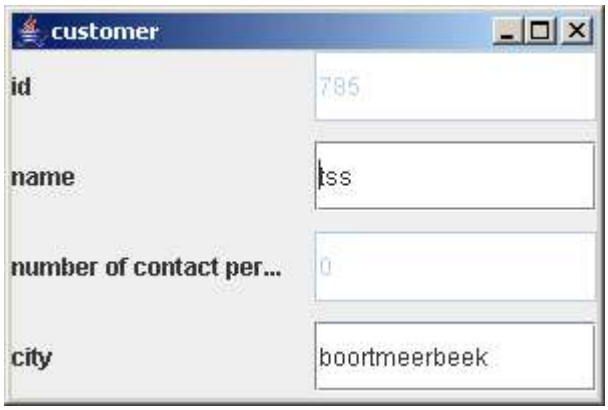
As you can see, the key class here is the dataSourceManager. Through this class you can manage your dataSource, the current record position, and so on.

We use rebind here because of the fact that it rebinds the value to the current position of the dataSourceManager. You should use bind in a collection (such as a list or a combobox).

You will still have to add the numberOfContactPersons property to the customer class!

```
Public class Customer extends DomainObject{
      …
      public int getNumberOfContactPersons(){
        return getContactPersons().size();
      }
      …
}
```

There will be two fields in the application that aren't editable (They will be grey). This has been automatically created by the parser because it can't find a set method for the property. It's obvious that you can't "set" an id because it's created when we insert a record. This is also the reason why getNumberOfOrders() can't be edited.



I think the time has come to add some navigation.

1.4.4. Adding navigation

As you can see, we have only displayed the first record but we want more! We want to navigate through all the records! Xml2gui has supplied a component for that, called NavigationControls. Just add the following line to your xml:

```
<navigationcontrols/>
```

When you run the application, there are 4 buttons displayed at the location where you have put the navigation controls. These include first, previous, next and last. What this component actually does is calling the DataSourceManager and invoking the appropriated methods. If you want to create you own fancy navigation control, just use the dataSourceManager!

If you are curious to see how we create an editable dataform, read on in the next chapter!

1.4.5. Adding editing

Data isn't only used to display to the client. We also need to edit this data. In Xml2gui this is really peanuts. Just add the control <editControls/> to you xml.

```
<editcontrols/>
```

This component creates a save, delete and new button on the dataform. These buttons allow the user to change the data in the form. Together with the navigation controls, we already have put together a nice form!

If you haven't noticed: we didn't go into any java code yet! Code-less data binding is the key!

Now the user can enter new data, that have to be validated of course, this is discussed in the next chapter.

1.4.6. Validation

On of the most important things in user interfaces is data validation. Nobody wants an exception thrown by the database because he can't insert a integer into a character field! Those validations should already have been done in the user interface.

Before using any validators we have to import a library: ValidationFactory. This tag library isn't ~~standard~~ included standardly.

```
<taglib className="tss.xml2Gui.validators.ValidatorFactory"/>
```

The class ValidatorFactory extends the TagLibrary class. If you want to know more about creating your own tag libraries, read ~~on~~ more in the chapter "Creating you own components"

It's time to abandon the control textfield and give a warm welcome to a new control called ValidatedFormattedTextField. This control comes with the xml2gui framework and adds some nice validation to the textfield.

The following xml is needed for the field:

```
<validatedFormattedTextField id="txtname" value="*rebind(name)"
validationText="">
     <validatedFormattedTextField.validator>
          <lengthValidator minLength="1" maxLength="14"/>
     </validatedFormattedTextField.validator>
</validatedFormattedTextField>
```

The validatedFormattedTextField has a property called validator. You can add different validators because it is a List. The validator itself is pretty straightforward and sets a minimum and maximum length for the field.

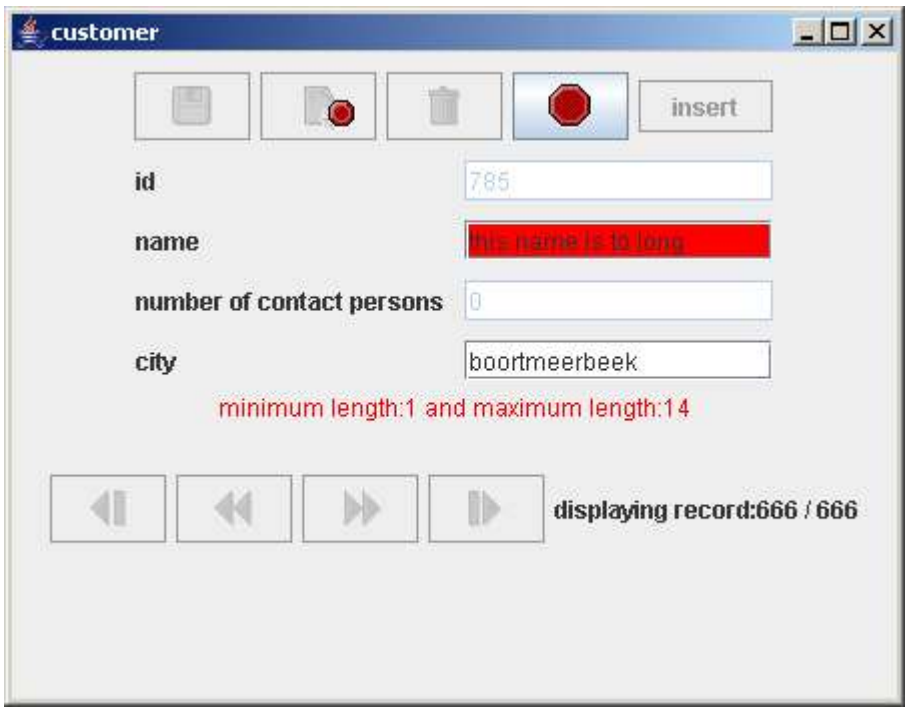
Xml2gui comes with a validator package which includes a ~~Belgium~~ Belgian tax number validator, a length validator, and a regular expression validator. Because we have created Xml2gui with extension in our mind, you can of course create you own validators by extending the Validator class.

```
public abstract class Validator {
    public abstract void validate(Object o) throws ValidationException;

    public abstract Element generateElement();
}
```

To display the error message, we have come up with the component validationSummery.

```
<ValidationSummery/>
```

Just add it to your xml and ~~this~~ it will display all the error messages you have in your form.



1.4.7. Searching

Now we can edit data, navigate trough data and validate in the UI, but we still can't search for a specific record! What we need is the component SearchComponent. This includes a field in which we can type our filter, a combobox to specify in which field we're searching, and a results list.

```
<?xml version="1.0" encoding="UTF-8"?>
<dataform title="customer" size="600,600"
defaultCloseOperation="JFrame.EXIT_ON_CLOSE" layout="borderlayout">
    <frameResource.dataSource>
        <objectDatasource name="customer"
className="tss.persistancy.demo.businessLogic.Customer"/>
    </frameResource.dataSource>
    <taglib className="tss.xml2Gui.validators.ValidatorFactory"/>

    <editcontrols constraints="BorderLayout.NORTH"/>
    <pnlSearchComponent id="searchComponent" constraints="BorderLayout.WEST"
layout="boxlayout(BoxLayout.Y_AXIS">
                <label text="Search"/>
                <label text="Search in:"/>
                <combobox id="cboFilterOn"/>

                <textfield id="txtFilter" columns="20"/>

                <scrollpane >
                    <scrollpane.viewportview>
                        <listRepeater visibleRowCount="100" id="lstList">
                            <listRepeater.visualDataSourceManager.ItemStyle>
                                <panel background="#9AB4F8">
                                    <label id="title" text="*bind({auto})"/>
                                </panel>
                            </listRepeater.visualDataSourceManager.ItemStyle>
                            <listRepeater.visualDataSourceManager.AlternateItemS
tyle>
                                <panel background="white">
                                    <label id="title" text="*bind({auto})"/>
                                </panel>
                            </listRepeater.visualDataSourceManager.AlternateItem
Style>
                            <listRepeater.visualDataSourceManager.SelectedItemSt
yle>
                                <panel background="#4265DD">
```

```
                                    <label id="title" text="*bind({auto})"/>
                                </panel>
                            </listRepeater.visualDataSourceManager.SelectedItemS
tyle>
                        </listRepeater>
                    </scrollpane.viewportview>
                </scrollpane>
            </pnlSearchComponent>

    <panel layout="gridlayout(0,2,10,10)" constraints="BorderLayout.CENTER">
        <label text="id"/>
        <textfield text="*rebind(id)"/>
        <label text="name"/>
        <validatedFormattedTextField id="txtname" value="*rebind(name)"
validationText="">
            <validatedFormattedTextField.validator>
                <lengthValidator minLength="1" maxLength="14"/>
            </validatedFormattedTextField.validator>
        </validatedFormattedTextField>
        <label text="number of contact persons"/>
        <textfield text="*rebind(numberOfContactPersons)"/>
        <label text="city"/>
        <textfield text="*rebind(city)"/>
    </panel>
    <panel constraints="BorderLayout.SOUTH">
        <ValidationSummery/>
        <navigationcontrols/>
    </panel>

</dataform>
```

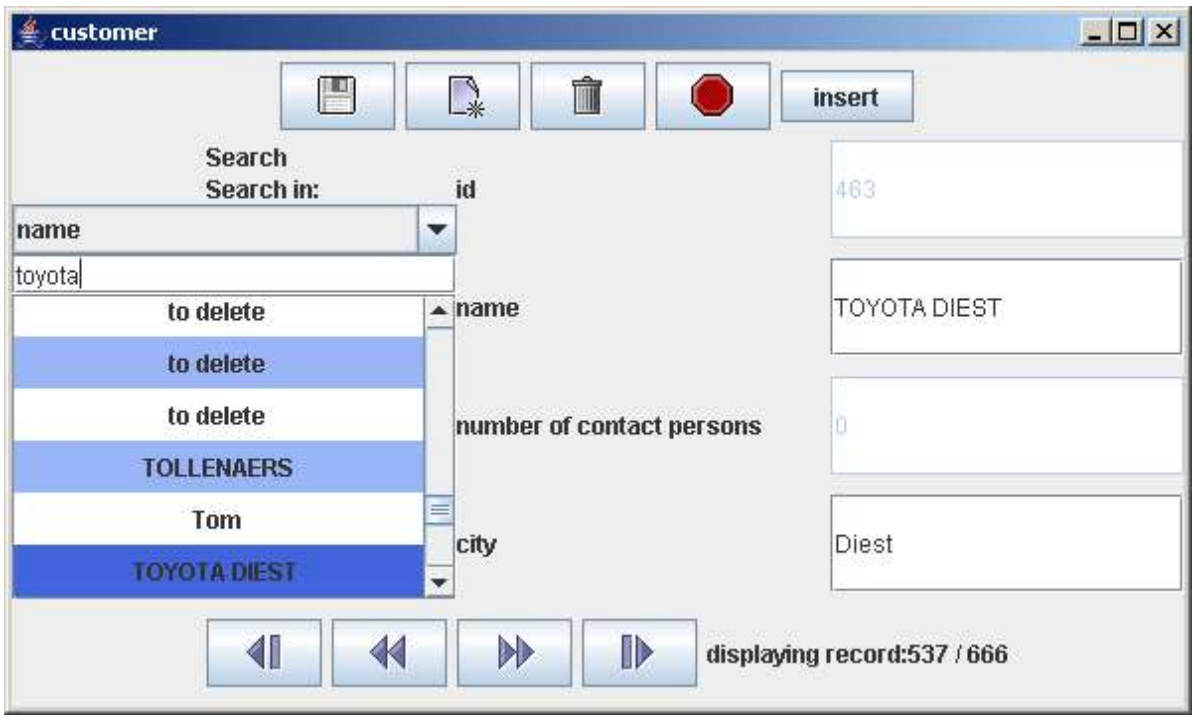
As you can see, the combobox is automatically populated with all the properties of our object. The search panel uses the following principal: search as you type. This means that when name is selected and if you type “a”, the first customer in the list ~~witch~~ whose name starts with an “a” is displayed.

There is one strange thing happening over here. Although we are using an external component (<searchcomponent>), we nest it with our own tags and ~~the~~ apparently the textfield has some actions bound to it although we didn’t specify them. -What actually happens is that the id “txtFilter” is bound to txtFilter in the pnlSearchComponent and not to our controller class! This is why it is really important to use the correct id, otherwise it can’t work!

There are three ids which are important:

- txtFilter: this is the input of the user in the search panel.
- cboFilterOn: the combobox which displays all the properties of the object.
- lstList: the list to display the results.

Feel free to move around the components and place them on the form wherever you want!



1.4.8. List layout

Don't we all want a nice looking application? Lists are sometimes difficult to read. This is ~~way~~ why we need templates. There is one for the selected item, one for the item itself and another one for the alternate item. One class is responsible for it. It's called VisualDataSourceManager.

The following xml makes it clear:

```
<listRepeater visibleRowCount="100" id="lstList">
            <listRepeater.visualDataSourceManager.ItemStyle>
              <panel layout="boxLayout(BoxLayout.Y_AXIS" background="gray">
                <label id="title" text="*bind({auto})" />
              </panel>
            </listRepeater.visualDataSourceManager.ItemStyle>
            <listRepeater.visualDataSourceManager.AlternateItemStyle>
              <panel layout="boxLayout(BoxLayout.Y_AXIS" background="white">
                <label id="title" text="*bind({auto})" />
              </panel>
            </listRepeater.visualDataSourceManager.AlternateItemStyle>
            <listRepeater.visualDataSourceManager.SelectedItemStyle>
              <panel layout="boxLayout(BoxLayout.Y_AXIS" background="red">
                <label id="title" text="*bind({auto})" />
              </panel>
            </listRepeater.visualDataSourceManager.SelectedItemStyle>
          </listRepeater>
```

1.4.9. Special rebind tag:{auto}

As you may~~be~~ already have noticed, there is a special rebind tag that is being used: "*rebind({auto})"

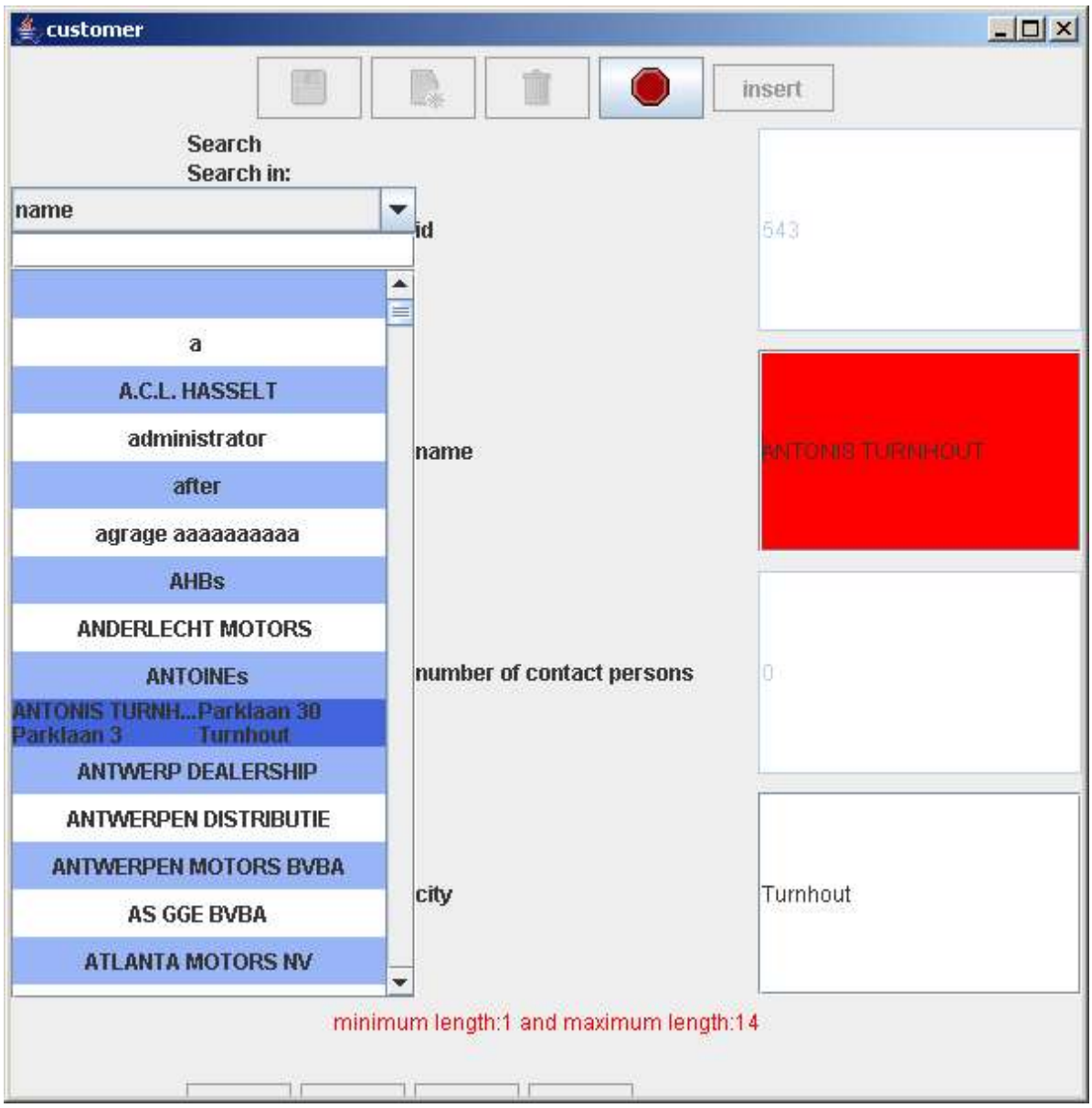
This means that you can use any field to display. This is dynamically changed trough the cboType. So if you choose to display "id", you select id in the combobox.

1.4.10. Some tricks with the item template

The different templates in the list aren't only capable of changing the background color of the different templates. In the following example we demonstrate it with another selected item style.

```
<scrollpane >
                  <scrollpane.viewportview>
                        <listRepeater visibleRowCount="100" id="lstList">
                            <listRepeater.visualDataSourceManager.ItemStyle>
                                <panel background="#9AB4F8">
                                    <label id="title" text="*bind({auto})"/>
                                </panel>
                            </listRepeater.visualDataSourceManager.ItemStyle>
                            <listRepeater.visualDataSourceManager.AlternateItemS
tyle>
                                <panel background="white">
                                    <label id="title" text="*bind({auto})"/>
                                </panel>
                            </listRepeater.visualDataSourceManager.AlternateItem
Style>
                            <listRepeater.visualDataSourceManager.SelectedItemSt
yle>
                                <panel background="#4265DD" layout="gridlayout
(2,2)">
                                    <label text="*rebind(name)"/>
                                    <label text="*rebind(street)"/>
                                    <label text="*rebind(number)"/>
                                    <label text="*rebind(city)"/>
                                </panel>
                            </listRepeater.visualDataSourceManager.SelectedItemS
tyle>
                        </listRepeater>
                    </scrollpane.viewportview>
                </scrollpane>
```

The result is pretty logical, ~~but~~ and as you can see, this component is very powerful.



1.5. Creating you own components

A component is actually a custom swing class that you can use in your xml. The cool thing here is that you can put

To create your own component you have to do three things:

- Create ~~A~~a java class which has an empty constructor and getters and setters (like a java bean).
- Implement the interface ParsableComponent: getXml and setXml is the interface we need to implement.
- Extends a Swing class

i.e.

```
package tss.xml2Gui.demos.manual;

import tss.xml2Gui.ParsableComponent;
import javax.swing.*;
import java.awt.*;

public class DateComponent extends JPanel implements ParsableComponent{
    public JLabel date=new JLabel("a date");
    private String xml;
    public DateComponent(){
```

```
        this.setBackground(Color.BLUE);
        this.add(date);
    }
    public void setXmlPath(String path) {
        this.xml = path;
    }

    public String getXmlPath() {
        return xml;
    }
}
```

Now you have to create your own tagLibrary.

```
package tss.xml2Gui.demos.manual;
public class MyTagLibrary extends TagLibrary{
    /**
     * register the tags which belongs to this taglibrary
     */
    protected void registerTags() {
        registerTag(DateComponent.class,"datecomponent");
    }
}
```

And add the tagLibrary to your xml-gui:

```
<?xml version="1.0" encoding="UTF-8"?>
<dataform title="ok!" layout="borderlayout">
    <taglib className="tss.xml2Gui.demos.manual.MyTagLibrary"/>
    <datecomponent/>
</dataform>
```