Generic GUIs
User Manual

Version: 2.5
vom 26.11.2016

Inhalt

1 INTRODUCTION 6
1.1 PURPOSE OF THIS DOCUMENT 6
1.2 REFERENCES 6
1.3 OPEN ISSUES 6
1.4 RECOMMENDATION FOR A ONE-DAY-TRAINING 6
2 HELLO WORLD EXAMPLE 7
2.1 WHAT ARE GENERIC USER INTERFACES 7
2.1.1 Attributes, Properties, Fields and JavaBean Conventions 7
2.2 THE FIRST GENERIC GUI (EXAMPLES-MANUAL/HELLOWORLD) 7
2.3 LAYOUT FILES 8
2.4 ADDING BEHAVIOUR (EXAMPLES-MANUAL/HELLOMETHODS) 9
2.5 PREVIEW 9
3 SUPPORTED DATA TYPES 11
3.1 SIMPLE DATA TYPES 11
3.1.1 Displayed Properties 11
3.1.2 Field Labels 11
3.2 STRUCTURED DOMAIN OBJECTS (EMBEDDED CLASSES) 11
3.3 ARRAYS & COLLECTIONS 11
3.3.1 Sorting 12
3.3.2 Generics & Polymorphy 12
3.4 EXAMPLE (EXAMPLES-MANUAL/DATATYPES) 12
4 VALIDATIONS 14
4.1 LENGTH LIMITATION (STATIC) 14
4.2 ALLOWED CHARACTERS 14
4.3 LIMITED RANGE OF VALUES 14
4.4 DEFINED FORMAT 14
4.5 MANDATORY DATA 14
4.6 DECIMAL PLACES 15
4.7 DYNAMIC VALIDATION 15
4.7.1 Validate Methods 15
4.7.2 Choice Methods 15
4.7.3 Validate Annotation 16
4.8 EXAMPLE (EXAMPLES-MANUAL/VALIDATION) 16
5 DIALOG FLOW 18
5.1 DISPLAYING A FOLLOW-UP MASK 18
5.2 MODAL DIALOGS 18
5.3 MESSAGE BOXES AND CONFIRM BOXES 18
5.4 ONE OBJECT – MULTIPLE WINDOWS 18
5.5 WIZARDS 20
EXAMPLE (EXAMPLES-MANUAL/DIALOGFLOW) 20
6 GUI TUNING 23
6.1 DISABLING OF FIELDS AND BUTTONS 23
6.1.1 Static disabling 23
6.1.2 Dynamic disabling 23
6.2 FIELD ORDER 23
6.3 BUTTON ORDER 23
6.4 AUTOMATIC CLOSING OF MASKS 24

6.5 ENABLING OF BUTTONS IN CASE OF VALIDATION ERRORS 24
6.6 WINDOW TITLES 24
6.7 HIDING FIELDS AND BUTTONS 24
6.8 REMOVING ENTRIES FROM COLLECTIONS AND ARRAYS 25
6.9 SELECTING ENTRIES IN COLLECTIONS AND ARRAYS 25
6.10 DISPLAYED LABEL 25
6.11 DATE FORMATING 25
6.12 USING SHORTCUTS / MNEMONICS 26
6.13 CHANGING THE LOOK AND FEEL 26
6.14 SETTING TAB- AND VIEW/MENU ICONS 27
6.15 CHANGING THE DEFAULT LAYOUT 27
6.16 EXAMPLE (EXAMPLES-MANUAL/LAYOUT) 27
6.16.1 Coloring schemes 28
6.17 SPLITTING UP A MASK 29
6.18 MANUAL GUI TUNING 29
6.18.1 Bringing manual and automatic changes together 31
6.19 INPUT ASSISTENCE 31
6.20 DETECTION OF INTERACTIVE CHANGES 31
6.21 EXAMPLE (EXAMPLES-MANUAL/PIMPMYGUI) 31
6.22 SEPARATE GUI CLASSES 34
6.23 DECORATORS 35
6.24 METHOD WITH PARAMETERS 35
6.25 EXAMPLE (EXAMPLES-MANUAL/PIMPMYSEPARATEGUI) 35
6.26 CHANGING THE SYNCHRONIZATION BEHAVIOUR 37

7 INTEGRATION WITH HAND-WRITTEN GUIS 39

7.1 PREREQUISITES FOR HAND-WRITTEN GUIS 39
7.2 INTEGRATION 39
7.2.1 Non-modal mask 39
7.2.2 Modal dialog 39
7.3 DIALOG FLOW TO A GENERIC GUI 39
7.3.1 Non-modal mask 39
7.3.2 Modal Dialog 39
7.4 MULTIPLE HAND-WRITTEN SUB WINDOWS 40
7.5 EXAMPLE (EXAMPLES-MANUAL/SELFMADE) 40
7.6 COMBINING GENERIC AND HAND-WRITTEN ASPECTS 42
7.6.1 Views from layout files 43
7.6.2 Example (examples-manual/customized) 43

8 INTERNATIONALIZATION 45

8.1 INTERNATIONALIZATION OF THE JAVA BEAN VALIDATION 45
8.2 PROGRAMMATIC LOCALIZATION 46
8.3 AUTOMATIC LABEL EXTERNALIZATION 46

9 SIMULATION MODE FOR FAST PROTOTYPING 47

9.1 SIMULATION INSTANCES 47
9.2 LOCALIZATION OF MODEL-BASED EXAMPLE CONTENT 47
9.2.1 JComboBox 47
9.2.2 JTable 48
9.2.3 JList 48
9.3 EXAMPLE (EXAMPLES-MANUAL/SIMULATION) 48

10 INTEGRATION WITH A BACKEND 49

11 CONFIGURATION BY GENGUI.PROPERTIES 50

11.1 CONFIGURATION PARAMETERS 50
11.2 SPECIALIZATION OF CONFIGURATION PARAMETERS 50
12 APPENDIX 52
12.1 EXCHANGING THE WINDOW MANAGER 52
12.1.1 Scalable generic web applications with AjaxSwing 52
12.2 CUSTOMIZING CODE GENERATOR TEMPLATES IN JFORMDESIGNER 53

History

Version	Activity	Author	Date
1.0	Translation to English	Jan Lessner	27.10.2011
1.1	Replacing „.“ By „_“ (preperation for improved I18N)	Jan Lessner	05.03.2012
1.2	Internationalization by jfd.localization explained	Jan Lessner	06.03.2012
1.3	Property merge utility mentioned	Jan Lessner	07.03.2012
1.4	Window title assembly explained	Jan Lessner	09.03.2012
1.5	Correction of some broken references and typos	Stefan Feldkord	05.07.2012
1.6	Generation of resource bundle files added	Jan Lessner	24.07.2012
1.7	Eager synchronization confirmation added	Jan Lessner	22.03.2013
1.8	@Modal explained as alternative way to display modal dialogs	Jan Lessner	10.05.2013
1.9	Documentation for simulation mode added	Jan Lessner	28.12.2013
2.0	Preperation for Open Source distribution: - Intercept renamed to Decorate	Jan Lessner	11.06.2014
2.1	Feature updates of version 1.2 documented: - touched() in case of invalid input - focus change in JTextAreas by tabSize 0 - propagation of @Eager and @Lazy	Jan Lessner	05.03.2016
2.2	Details of version 1.3 documented: Coloring, focus initialization	Jan Lessner	25.03.2016
2.3	Directory names adapted to new examples structure of version 1.4	Jan Lessner	28.03.2016
2.4	Width and height attributes for @Modal annotation added	Jan Lessner	03.04.2016
2.5	Various corrections (thanks Sven)	Jan Lessner	26.11.2016

1 Introduction

1.1 Purpose of this document

This document contains the user documentation for the framework for generic GUIs. It starts with a simple HelloWorld example as a first introduction. This example is evolved in the following chapters to explain the various other features of the framework.

If you are only interested in a first insight, you should read the first steps introduction instead which is available online at http://gengui.sourceforge.net/gengui-first-steps.

1.2 References

Reference	Reference Identification	Title / Comments
[FSI]	http://gengui.sourceforge.net/	Gengui first-steps introduction
[LIS]	http://en.wikipedia.org/wiki/Liskov_substitution_principle	Liskov Substitution Principle
[IDW]	http://www.infonode.net/index.html?idw	Website of the toolkit Infonode Docking Windows
[HV4]	http://docs.jboss.org/hibernate/stable/validator/reference/en/html_single/	Java Bean Validation, reference implementation is Hibernate Validator 4
[JBV]	http://download.oracle.com/docs/cd/E17410_01/javaee/6/api/index.html?javax/validation/constraints/package-tree.html	Overview of Java Bean Validation constraints

1.3 Open Issues

Open Issue	Responsible	Status
Replace „licence plate“ with „licence number“	?	Open

1.4 Recommendation for a One-Day-Training

For a very quick introduction in about one hour you should start with the first steps documentation which is available as PDF and online at http://gengui.sourceforge.net/gengui-first-steps. This intro is much younger than this manual here and benefits from years of experience with explaining the principals of Naked Objects and gengui. It especially addresses the differences to the classic MVC pattern.

Reading the complete manual and working through all examples takes about 3 days assuming that all required tools are available and installed. For a short training of about one day it is recommended to work through the following chapters:

- Chapters 2 to 5 completely
- Chapters 6.1 to 6.15, optionally the example of 6.16.
- The other chapters should roughly be scanned at least by reading the captions, to get an idea about what aspects are covered by the framework.

Concerning the examples it is strongly recommended to work them out practically in the sense that you actually write and change code. This avoids to have understood the features only theoretically. If you find the time, it is a good idea to create a completely different user interface by your own.

2 Hello World Example

The source code of all examples in this manual is located in directory `examples-manual` of the gengui project. Every chapter mentions the sub directory which is relevant for the example. E.g. the directory `examples-manual/helloworld` includes the first Hello World example.

Every chapter also includes the required Java call to start the example. In general generic user interfaces are displayed by a call like that:

```
java gengui.infonode.RootFrame <class #1> <class #2>
```

To display all examples at once, the call simply must be extended by the class names of the objects to be instanciated for display.

IMPORTANT: The directory `examples-manual` must be part of the CLASSPATH on application start so that the framework can find the individual settings in the form of configuration and layout files. Additionally, the directory `resources` must be on the CLASSPATH as well. It contains the file gengui.properties with a reasonable basic configuration for the framework.

2.1 What are generic user interfaces

2.1.1 Attributes, Properties, Fields and JavaBean Conventions

When working with the gengui framework, it is important to follow the JavaBean conventions for accessing attributes. An “attribute” is a data member of a Java class which – according to the JavaBean convention - is accessed by appropriate getter and setter methods. It is best practice that the identifier following the “get”- and “set”-prefix, corresponds to the name of the attribute, whereby the first letter is capitalized.

An attribute

```
private String name;
```

is accessed from outside the class by the member functions

```
public String getName();
public void setName(String name);
```

The notation with the leading capital letter is often referred to as a *property* of the class. A class may very well contain virtual properties, i.e. properties that do not refer to an attribute of the corresponding name but e.g. to an assembly or type conversion of other attributes. The actual names of attributes are therefore a secondary aspect. Many frameworks that are based on Java reflection, work with the JavaBean convention and assume its correct usage by the developers. This is also true for gengui.

So if this documentation uses the term “property”, it refers to the convention above. The terms “field” refers to components on the screen, i.e. a text field, a check box, a combo box etc.

2.2 The first generic GUI (examples-manual/helloworld)

To display a first generic user interface, we simply need a pure Java bean (a “POJO” if you like that term).

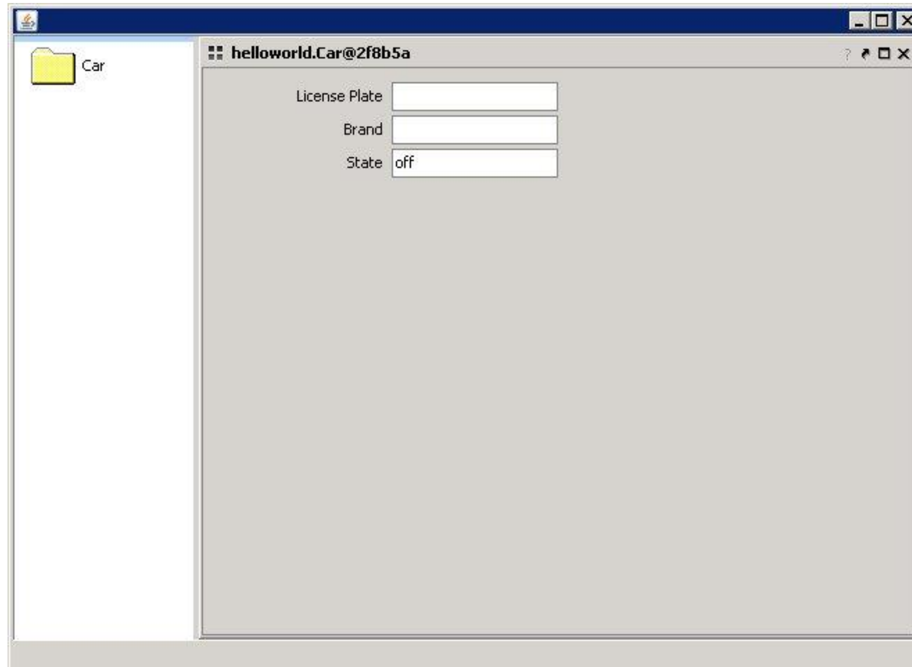
```
public class Car {

    private String state;
    private String licensePlate;
    private String brand;

    public Car() {
        this.state = "off";
    }

    public String getLicensePlate() { return licensePlate;   }
    public String getBrand() { return brand; }
    public String getState() { return state; }
```

```
        public void setLicensePlate(String licensePlate) { this.licensePlate = licensePlate; }
        public void setBrand(String brand) { this.brand = brand; }
        public void setState(String state) { this.state = state; }
}
```

To bring an object of this class on the screen, the framework's main class is started as follows (don't forget `examples-manual` and `resources` on the CLASSPATH!):

```
java gengui.infonode.RootFrame helloworld.Car
```

Clicking on „Car" causes the following screen to come up:



So the Java class was visualized in a straight-forward way without implementing a single line of GUI code. Clicking on "Car" a second time causes another object of that class to be instanciated and displayed in a second window. The two masks allow to display and enter data for the two cars.
The main routine of the RootFrame class accepts only the names of classes with default constructors which therefore can be instanciated from scratch when pressing an appropriate button in the main menu. The constructor of RootFrame also allows passing already instantiated objects. Some other parameters are supported as well (check the Javadocs if you want to know more) but most of the application configuration is placed in the file `gengui.properties`.

2.3 Layout Files

After having started the Hello World example the first time, you may notice the creation of a file `helloworld_Auto.jfd` in the working directory of the application (Eclipse users may have to refresh the project first ;-). This file contains the layout definition for the car GUI. The file includes information about the placement of all displayed fields and buttons. Whenever the framework displays an object, it first checks if a corresponding layout file is already available. If the file can be found, it is used to layout the mask; otherwise the framework creates one on the fly.
Subsequent changes of the Java bean (new property[1], new method, removing a property etc.) cause the layout file to be out of sync with the source code. This causes an exception at runtime when using the frameworks default configuration. This default behaviour is sometimes annoying during early development phases where frequent refactoring takes place. Therefore it can easily be changed by configuration (`jfd.retention.strategy` in `gengui.properties`). The most interesting mode is 'merge' which allows to add new components without destroying existing layouts.

[1] For the term „property" siehe section 2.1.1

JFD files can be loaded and edited by the layouting tool JFormDesigner. For the moment it is sufficient to know that these files are around and what they are good for. Following chapters will refer to these files and explain how maintain them and how to configure their generation in the framework. The following examples are configured in a way that JFD files are *not* created, so the examples' Java bean structures may safely be modified.

2.4 Adding behaviour (examples-manual/hellomethods)

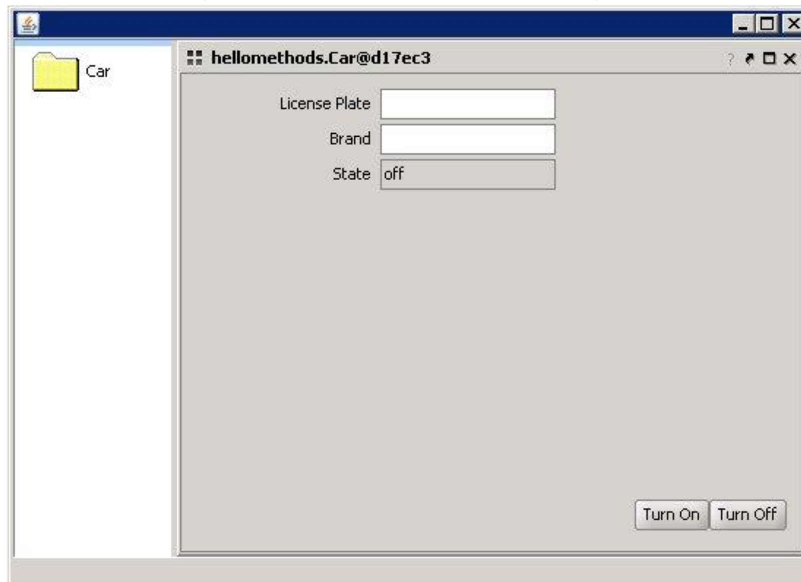
The state of cars was arbitrarily changeable in the last example. Now the car should better provide methods for reasonable state changes. The methods should only allow to turn the car off and on.

```
public class Car {

        // Attribute definitions, constructor, getters/setters omitted here

        public void turnOn() {
                this.state = "on";
        }

        public void turnOff() {
                this.state = "off";
        }
}
```

To ensure that the state is not modified interactively without activating the methods, we remove the setter method for the attribute. The new domain class `hellomethods.Auto` is added to the argument list of the RootFrame:

```
java gengui.infonode.RootFrame helloworld.Auto hellomethods.Auto
```

Adding the methods and removing the setter produces the following GUI:



ATTENTION: The framework adds buttons only for methods without parameters. A pattern for the activation of methods with parameters is described in section 6.24.

If a method throws an exception, the exception text is displayed as displayed in an error message box. The exception text automatically walks through localization, so you may as well just provide a localization key that refers to a translation in a gengui property file. See chapter 8 for further details on internationalization. Throwing an exception without a message will silently be swallowed, assuming that the originating problem has already been handled in the code and doesn't need to be reported graphicaly. Runtime exceptions are treated the same way like checked exceptions but additionally are rethrown and thus will reach the JVM's default exception handler.

2.5 Preview

The two sections above demonstrate that is is fairly simple to display a domain object. Nevertheless there arise various questions from the examples:

- How can I avoid the activation of turnOn for an already running car?
- Is there a different way to disable fields rather than by removing or renaming the setter?
- How can I limit the brand to a particular set of reasonable values?
- Which other data types can be displayed by the framework?
- How can I ensure that the licence plate follows a particular format?
- May domain objects be nested?
- How can I store my domain objects persistently?

The following chapters give answers to these and many other questions.

3 Supported Data Types

The following sections describe the default visualizations for domain objects. These visualizations may additionally be influenced by various annotations (see following chapter).

3.1 Simple Data Types

The following basic data types are supported by gengui and are visualized as follows:

Data Type	Mapping
Integer/int	JTextField, only digits allowed, limited by Integer.MAX_VALUE
Long/long	JTextField, only digits allowed, limited by Long.MAX_VALUE
Boolean/boolean	JCheckBox
String	JTextField
java.util.Date	JTextField with SimpleDateFormat
java.math.BigDecimal	JTextField, only digits allowed
Enum	JComboBox
java.io.File	JTextField

These data types are also called „simple data types". If there is a limited value set specified for a property (see section 4.7.2), it is always mapped to a JComboBox, independently from the actual data type.

3.1.1 Displayed Properties

By default the framework displays all properties[2] of a class which have a public getter method defined. If there is also a public setter method defined, the displayed field is editable, otherwise it is read-only.

3.1.2 Field Labels

Every field is preceded by a JLabel with a text that is derived from the property name. Camel-case syntax causes a whitespace splitting (e.g. getNumberOfChildren() is displayed as „Number Of Children").

3.2 Structured Domain Objects (Embedded Classes)

Domain objects may be structured by one class referencing another class. The related objects are in this case displayed within the mask of the referencing object. To separate the fields from each other, the fields for referenced object are displayed under a separation line with the name of the reference as a caption.

3.3 Arrays & Collections

Arrays and collections are by default mapped to non-editable JTables. The JTable contains one row for each element of the collection / array.

Data type within the collection	Mapping
Simple data type (see list in section 3.1)	JTable with a single column Display of toString One button for each method of this type
java.lang.Object	JTable with a single column Display of toString One button for each method of this type
Structured domain objects	JTable with one column per getter method which

[2] For the term „property" see section 2.1.1

	returns a simple data type The column title conforms to the visualization rules from section 3.1 Display of the result from the getter method One button for each method of this type
Byte / byte	JLabel, showing the data in form of an ImageIcon.

If there is only a single method available for the contained class, this method can be activated by double-clicking the row of interest. To support double-clicks also in case of multiple methods, see section 6.18. For display alternatives see section 6.18.

3.3.1 Sorting

The created tables are generally sortable per column by double-clicking the column title. Exceptions are collections of type `java.util.List` and `java.util.SortedSet` making the assumption that choosing these collection types expresses a relevant order of the content. The distinction is based on the type of the property (strictly speaking: the return type of the getter method), not the actually instanciated collection type. E.g. if a property of type java.util.Collection returns an ArrayList as the actual implementation, the displayed table is sortable anyway. The default behaviour may be changed by adding the annotation @Sortable to the getter method.

3.3.2 Generics & Polymorphy

To achieve a detailed visualization of domain objects from a collection, the collection should be typed using Java generics. Otherwise the framework can only display “the least common denominator” which is toString for all contained objects.

The table displays only the data of the contained type. As a consequence, if the collection also contains objects of derived types, the additional properties of these types are not displayed. (e.g. a collection of vehicles contains cars and bikes with additional properties. The table displays only the properties of the vehicle base class).

3.4 Example (examples-manual/datatypes)

The following example shows a more comprehensive model for a car. The motor with its methods for turning it off and on is implemented as a separate class. There is a number of additional attributes with different data types now. All trips which have been made with that car are associated by an array.

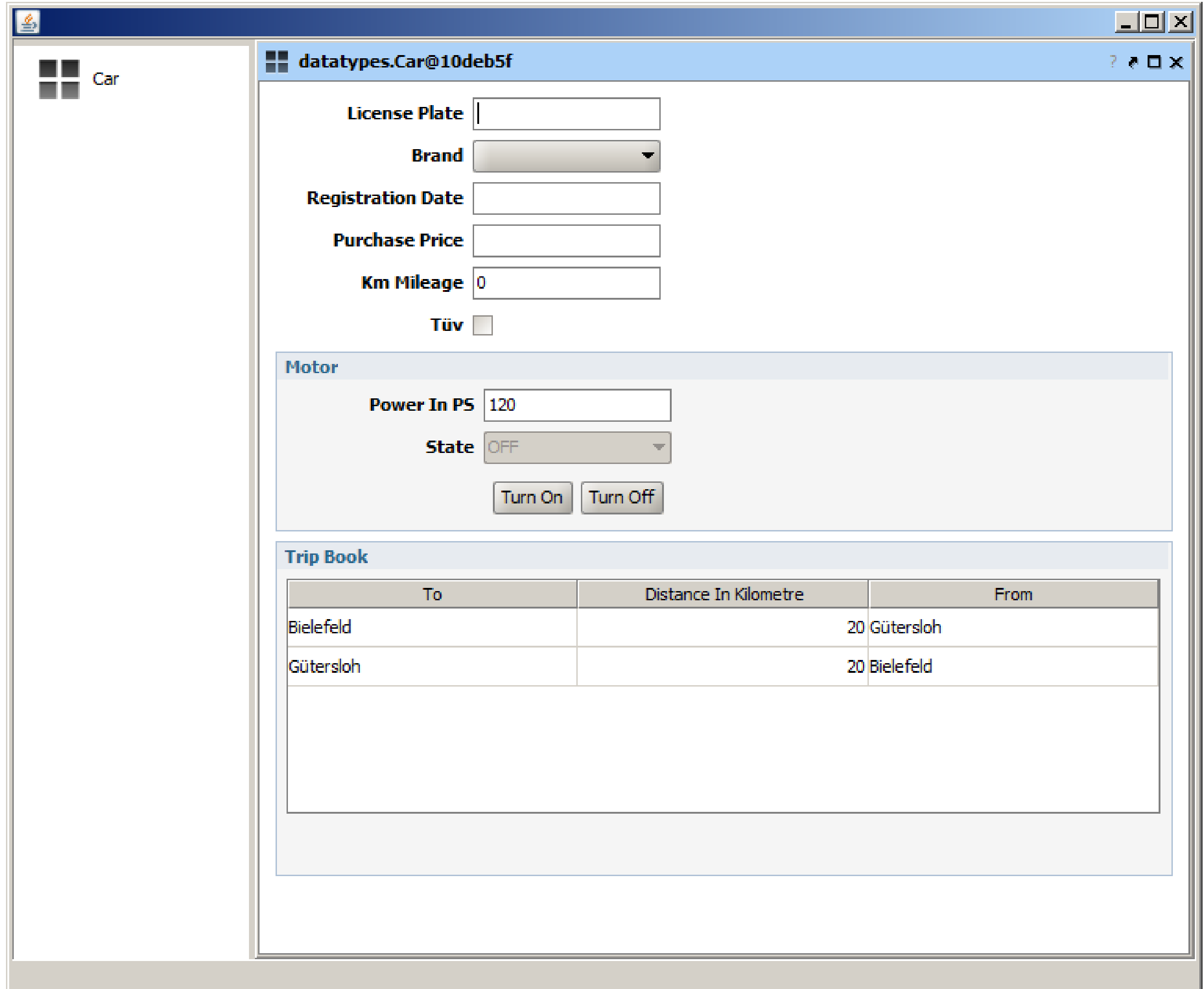
```
public class Car {
	private String licensePlate;
	private CarBrand brand;
	private long kmMileage;
	private BigDecimal purchasePrice;
	private boolean technicalControl;
	private Date registrationDate;
	private Trip[] tripBook;
	private Motor motor;

	// constructor, getters/setters omitted here
}

public class Motor {
	private int powerInPS;
	private MotorState state;

	// constructor, getters/setters omitted here

	public void turnOn() { this.state = MotorState.ON; }
	public void turnOff() { this.state = MotorState.OFF; }
}

public class Trip {
	private String from;
```

```
        private String to;
        private int distanceInKilometre;
        // constructor, getters/setters omitted here
}
```

Start of the example:

```
java gengui.infonode.RootFrame datatypes.Auto
```

The class is displayed like this:



4 Validations

In many cases it is required to limit the user's data entry or to validate the input. For these cases, the framework provides some features which are introduced in this chapter.

Validation can generally be classified in static and dynamic validation. Static constraints can be expressed by annotations in the domain class. If the validation depends on particular situations resp. the domain object's state, the dynamic approach through special validation methods provides more flexibility.

All validation annotations must be added to the getter methods of the properties.

Generally the framework performs static validations through the standardized Java Bean Validation (current implementation: Hibernate Validator 4, see also [JBV] und [HV4]). The bean validation annotations are also recognized e.g. by JPA2 and JSF2 implementations. Using the standard thus allows performing validations on different layers, both automatically and programmatically.

4.1 Length limitation (static)

Realization:

The length of a property can be limited using the **`@Size`** annonation. It allows specifying both a minimum and a maximum length.

Supported data types: String, Collection, Map, Array

Effect:

Maximum length: the appropriate fields accept only the specified number of characters

Minimum length: The minimum length is checked when leaving the field. If the input is too short, the field is tagged to have invalid input. Method buttons are disabled until the input is corrected.

4.2 Allowed Characters

Realization:

The set of allowed input characters can be limited using the **`@Charset`** annotation from the package **`gengui.annotations`**. The annotation expects a string including all the allowed characters.

Supported data types: all (Object)

Note: validation is based on the toString result of the object returned by the getter method.

Effect:

The apprpiate fields accept only the specified characters.

4.3 Limited Range of Values

Realization:

The range of values for a numeric property can be limited by the **`@Min`** and **`@Max`** annotations.

Supported data types: all numeric types except double and float

Effect:

The range of values is checked when leaving the field. In case of a range violation, the field is tagged to have invalid input. Method buttons are disabled until the input is corrected.

4.4 Defined Format

Realization:

The input format for a property can be specified in a details way using the **`@Pattern`** annotation. The annotation expects a string with a regular expression which the property value must match..

Supported data types: String

Effect:

The input is matched agains the regular expression when leaving the field. In case of an expression violation, the field is tagged to have invalid input. Method buttons are disabled until the input is corrected.

4.5 Mandatory data

Realization:

A property can be tagged as a mandatory using the **@NotNull** annonation.

All data types which can be null are supported. In the gengui framework, the annotation also works for primitive data types as the actual data is usually assembled by converting a text input to a value of the required type.

Effect:

When leaving the field, the framework checks if the field contains any input. An empty field is tagged to have invalid input. Method buttons are disabled until the input is corrected.

4.6 Decimal places

Realization:

The number of decimal places can be specified using the **@Digits** annotation.

Supported data types: BigDecimal, BigInteger, String, byte, short, int, long (und Wrapper)

Example: @Digits(integer=5, fraction=2) means: 5 digits before the decimal point, 2 digits after the decimal point

Effect:

When leaving the field, the framework checks whether the input has a valid number of digits.

Less decimal places: The field is filled with missing 0 characters, if this doesn't exceed the maximum length, otherwise the field is tagged to have invalid input. Method buttons are disabled until the input is corrected.

Too many decimal places: The field is tagged to have invalid input. Method buttons are disabled until the input is corrected.

4.7 Dynamic Validation

It may happen that the validation rules for an object depend on its state. For these cases, the framework provides a dynamic validation feature based on naming conventions.

4.7.1 Validate Methods

These methods are called by the framework when leaving a field and *before* updating the appropriate domain object's property. Validate methods must have the following signature:

```
public String validate<NameOfProperty>(<TypeOfProperty> newValue)
```

The return value of this method is an error text which is displayed in the status bar of the root frame in case of a validation error. A return value of null expresses a successful validation causing the interactive input to be transferred to the domain object afterwards. E.g. the input for a property "price" may be dynamically validated by a method called "validatePrice".

Adding the annotation @Forced to the property's setter method causes the validation to execute *after* the input is transferred to the domain object. I.e. the input is transferred in any case, as long as the static validation rules are all passed. This procedure must of course be used with care and is of interest for validation rules which are not related to single properties but to interdependencies between multiple properties. Especially in cases of XOR dependencies the validation of which may cross the correction of invalid input in certain situations.

The optional parameter of this annotation allows to specify the properties the fields of which should be reset to the object's value in case of a validation error. An empty list for this parameter causes the @Forced annotation to be ignored and all field content is validated the usual way.

4.7.2 Choice Methods

These methods allow a dynamic limitation of the value set for a property. Choice methods must have the following signature:

```
public <TypeOfProperty>[] choice<NameOfProperty>()
```

The return value is the list of supported values for the current situation resp. the current object's state. If a choice method exists, the property is displayed by a JComboBox. This is also true for properties returning structured data types. In this case the combo box displays the result of the toString method of these objects, or the title method if available (see also section 6.6). Choice methods for structured types cause a different kind of data manipulation. With a choice function, the user manipulates the reference while the content of the referenced object remains unchanged. To change both content and reference, the same attribute must be made available through two properties with only one of these properties having a choice method.

If a property may be null, the corresponding choice method must contain a null value as a valid selection. Otherwise the framework assumes that null is not allowed in the current situation even though the data type may generally support null values.

It is important to know that the framework compares the current property value with the values from the corresponding choice method to check its validity. Structured data types therefore must define a valid equals() method or the choice method must return the *identical* value set with each call, not only an *equal* one. The comparison takes place whenever the user interface and the domain objects in the background are synchronized. This happens usually quite often, e.g. when a field loses the focus or a button is pressed (see section 6.26).

4.7.3 Validate Annotation

The **`@Validate`** annonation allows to tag a property as requiring dynamic validation. In this case, the framework ensures that there exists an appropriate validate method. This avoids e.g. that the validation is missing due to a typo in the method name.

Attention: Don't mix that up with `javax.validation.constraints.Valid` (see [JBV])!

4.8 Example (examples-manual/validation)

The following code snippet shows an extended Car class, which makes use of the features described above.

```
public class Car {

        // Constructor, attributes and uninteresting getter and setter methods omitted here

        // see section 4.4
        @Pattern(regexp = "[A-Z]{1,3}-[A-Z]{1,2} [1-9][0-9]{0,3}")
        public String getLicensePlate() { return licensePlate; }

        // see section 0
        @NotNull
        public CarBrand getBrand() { return brand; }

        // see sections 4.6 and 4.1
        @Digits(fraction = 2, integer = 19)
        @Size(min = 0, max = 10)
        public BigDecimal getPurchasePrice() { return purchasePrice; }

        // see section 4.3
        @Min(0)
        @Max(1000000)
        public long getKmMileage() { return kmMileage; }

        // see sections 4.2and 4.1
        @Charset("45")
        @Size(max = 1)
        public int getNumberOfGears() { return numberOfGears; }

        // see section 4.7.2
        public CarBrand[] choiceBrand() {
                //e.g. filter because system configuration says: German brands only
                return new CarBrand[] {
                        CarBrand.OPEL, CarBrand.VW,
                        CarBrand.MERCEDES, CarBrand.BMW  };
        }

        // see section 4.7.1
        public String validatePurchasePrice(BigDecimal value) {
                if (brand != null && brand == CarBrand.OPEL &&
                    isPurchasePriceGreaterThan(value, 20000))
                        return "Too expensive";
                else
                        return null;
        }
```

```
}
```

Start of the example:

```
java gengui.infonode.RootFrame validation.Auto
```

The following screenshot shows the result. Fields with the red error marker violate the constraints.

5 Dialog flow

5.1 Displaying a follow-up mask

Implementing dialog flow with gengui is very simple: every result from a method call is displayed in a new mask. The new mask appears as a new window in the application's root frame and is placed over the initiating window or – if present – over a window for another domain object of the same type. The initiating window is not removed but stays in the background so the user can move back to it.

5.2 Modal Dialogs

Modal dialogs can be displayed in two different ways. The API-free and therefore more test-friendly and portable way is to annotate a method with **`@Modal`** which causes the method's return value not to be displayed as a new window (as above) but as a modal dialog on top of the initiating window:

```
@Modal public Trip newTrip() {
        return new Trip(this);
}
```

As usual, every method of the returned object's class becomes a button in the dialog. By default, clicking one of the buttons closes the modal dialog. If the initiating domain object is supposed to be manipulated by the modal operation, the method's return object must become aware of the initiator, usually by simply passing the initiator in the modally displayed object's constructor. An example can be found in section 6.24.

The modal dialog's size is automatically calculated from the mask to display. Alternatively you may specify the width and height as attributes of the annotation which is mainly of interest for wizards (see section 5.5).

As an alternative, there are a few API functions available for that:

```
Object modal(Object domainobject);
```

This API is provided by the GUIServiceI interface which is referenced in some of the following sections as well. The example in section 0 demonstrates how to get access to this interface from within the code.

The modal() operation displays the passed domain object in a modal dialog. The return value of the activated closing method is passed over to the caller as the result of the modal() operation. The API-based way to display a modal dialog is especially of interest if a domain object's method must initiate *either* a new window *or* a modal dialog depending in the object's state. In this case, the @Modal annotation is of course not suitable.

5.3 Message boxes and Confirm boxes

Message and confirm boxes can be displayed using the `message` resp. `confirm` methods of the GUIServiceI interface.

5.4 One object – multiple windows

A domain object may be displayed in multiple sub windows by either grouping the content by means of the **`@Group`** annotation (see section 6.16.1) or by manually splitting up the content of a layout file into multiple files. The latter procedure is more work but provides a bit more flexibility.

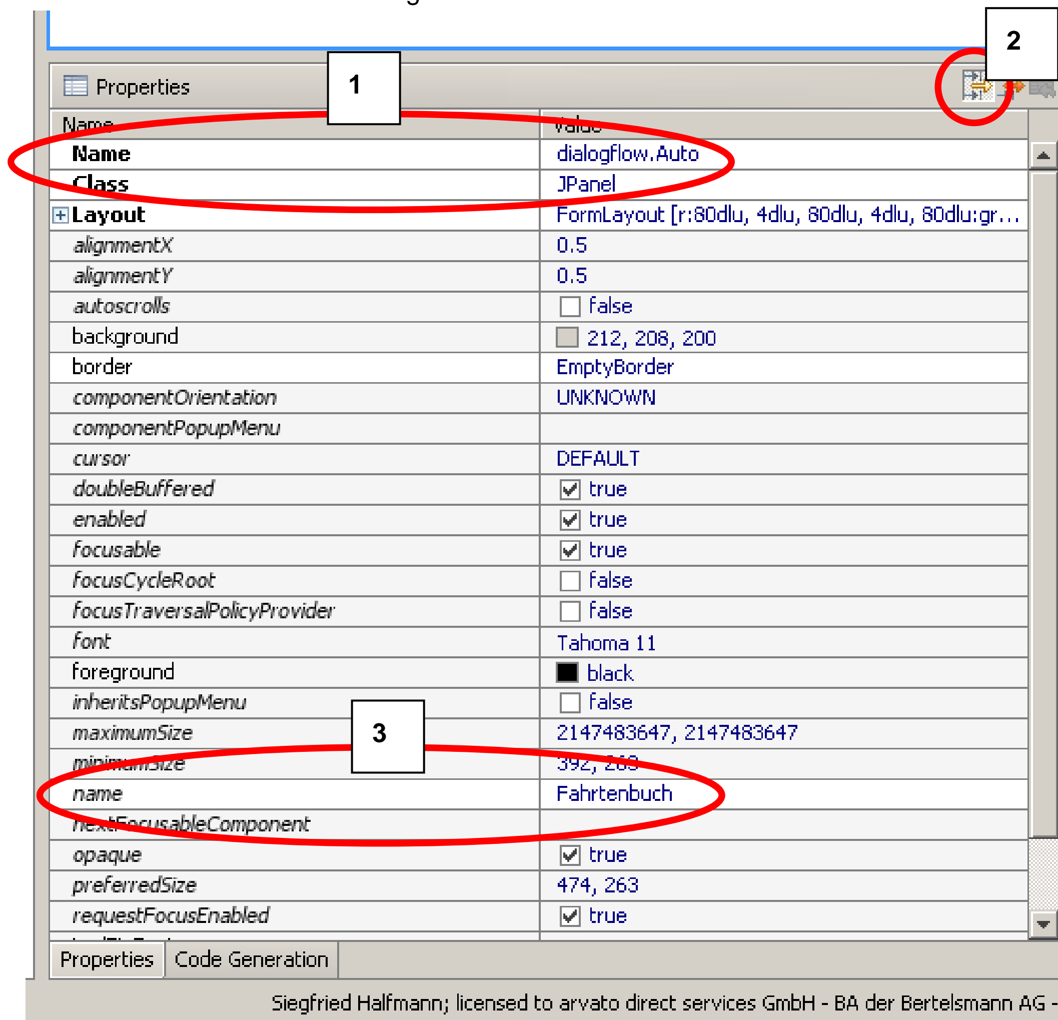
Step by step to multiple windows:

1. Code your domain object
2. Start the root frame with the configuration to create and keep layout files (jfd.retention.strategy=keep in gengui.properties)
3. Display the domain object
4. Stop the application
5. For every sub window which is supposed to be displayed later on: copy the created layout file *<domainclass>.jfd* and rename the copy according to the sub window: *<domainclass>_<windowname>.jfd* (e.g. you created foo_Bar.jfd, what's the name of the copy for the "stool" window? Right: foo_Bar_stool.jfd). This naming rule is just a recommendation –

the framework doesn't care what is written betewwen the dots. The file might as well be named foo_Bar_strip.jfd.

6. Open every sub window's layout file with the JFormDesigner
7. Remove all elements which are not supposed to be displayed in that sub window (e.g. bei removing individual elements or complete lines from the layout). **ATTENTION**: in the end, every element from the original layout must occur on at least one of the sub windows.
8. Change the Swing property "name" to the name of the sub window (ATTENTION: this can easily be mixed up with JFormDesigner's own "Name" property – see screenhot: bubble 1 = Name property of JFormDesigner → no need to change, bubble 2 = button to show the extended Swing properties → click it, bubble 3 = the name property from Swing → put the name in here which you like the sub window to be displayed at runtime).
9. Save all the JFD files
10. Start the root frame
11. Display the domain object again – now it is displayed in multiple windows which by default are tabulated one after the other. The windows can freely be arranged by the user within a surrounding window.

Screenshot from JFormDesigner:



If a domain object is displayed in multiple sub windows, the framework only calls the getter methods for properties which are actually displayed on the foreground window. This allows to for lazy-loading of data for other sub windows. E.g. if a customer is displayed by one sub window for its master data and one window for its orders, you may load the orders from the DB in the appropriate getter. The DB will only be accessed if the user actually changes to the order window for the first time.

It is generally allowed to display a property on multiple sub windows which allows e.g. to display common header information on all sub windows which belong together (e.g. name and surname of a customer one both, the master data and the order window). If the property needs interactive

modification, it is recommended to make only *one* of the corresponding fields editable. The other fields can be set as non-editable in the JFormDesigner. Fields which are marked as read-only in the layout file will not be made editable by the framework at runtime.

When having split one mask into multiple masks, it is important to ensure that all created layout files are available at runtime. If a property or a method could not be coupled to a graphical component due to missing layout files, the framework will throw an exception or generate a new default layout for a single mask (depends on the configuration).

5.5 Wizards

Wizards can easily be implemented by modal dialogs. By default, a modal dialog is immediately closed as soon as any of its method buttons is clicked, and the result object is returned to the caller when using the API-based approach. (see 5.2). A wizard however consists of a sequence of dialogs, so we simply have to avoid that the modal dialog is closed when pressing a button. This can be achieved using the **`@Closer`** annotation (see also section 6.4). Every method of a modally displayed object, that has the **`@Closer`** annotation attached, closes the dialog. Every method which is NOT annotated this way, leaves the dialog open. The return value of a non-annotated method is displayed as a new mask by the framework. The new mask is not displayed in a new window in this case but simply replaces the former mask in the same modal dialog. Implementing a wizard on this basis is straight forward.

What happens if a wizard object has only a single method `next()` which should not close the dialog but move over to the next wizard step? The method has no **`@Closer`** annotation and there is no other method, so the default behaviour says: close on `next()`. Usually a typical wizard always provides an abortion button in every step, so this problem should not occur often. However, for the rare special cases, the following operation of the GUI service allows to specify different default behaviour:

```
Object modal(Object domainObject, boolean modalClosingBehavior)
```

When using the @Modal annotation instead of the API function, the closing behaviour can be specified by the annotation's value attribute.

In a wizard it may happen that successively displayed masks have different sizes. The framework however creates the modal dialog by default with an optimal size for the first mask which may be too small for following steps. This can be avoided by starting the modal operation with the following function which allows specifying the initial size:

```
Object modal(Object domainObjekt, int width, int height)
```

The @Modal operation also supports an optional width and height specification. An alternative way to reserve enough space is to enlarge the first mask's preferredSize property in the JFormDesigner (assuming that you safe JFD files). This is actually the recommended way to keep from cluttering the domain classes with graphical details.

Example (examples-manual/dialogflow)

The following class shows a query mask which provides a set of Cars. To show the details of a car, each Car object must provide an appropriate method which simply returns the Car itself. This will already cause a dialog flow.

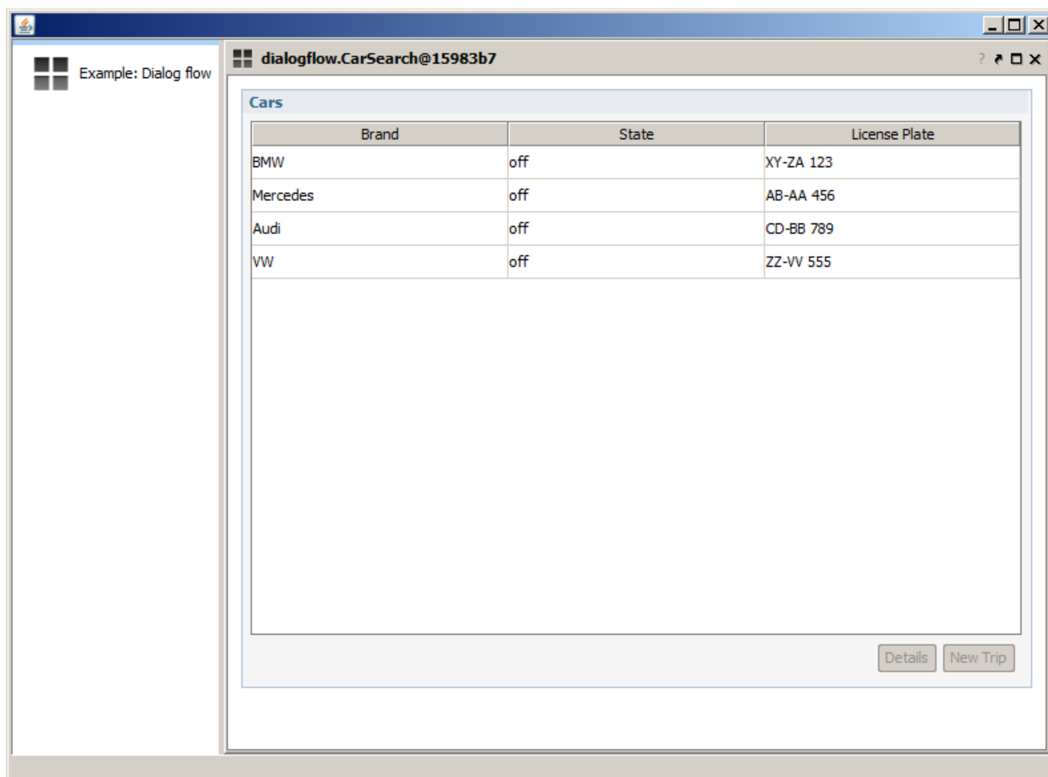
```
public class CarSearch {
	private ArrayList<Car> cars;

	public CarSearch() {
		// Fill up the list of cars...
	}

	public ArrayList<Car> getCars() { return cars; }
	public void setCars(ArrayList<Car> cars) { this.cars = cars; }
}

public class Car {
	private String state;
	private String licensePlate;
	private String brand;
	private Collection<Trip> tripBook;
	transient public GUIServiceI guiservice = new GUIAdapter();

	// Constructor, getters and setters omitted here
```

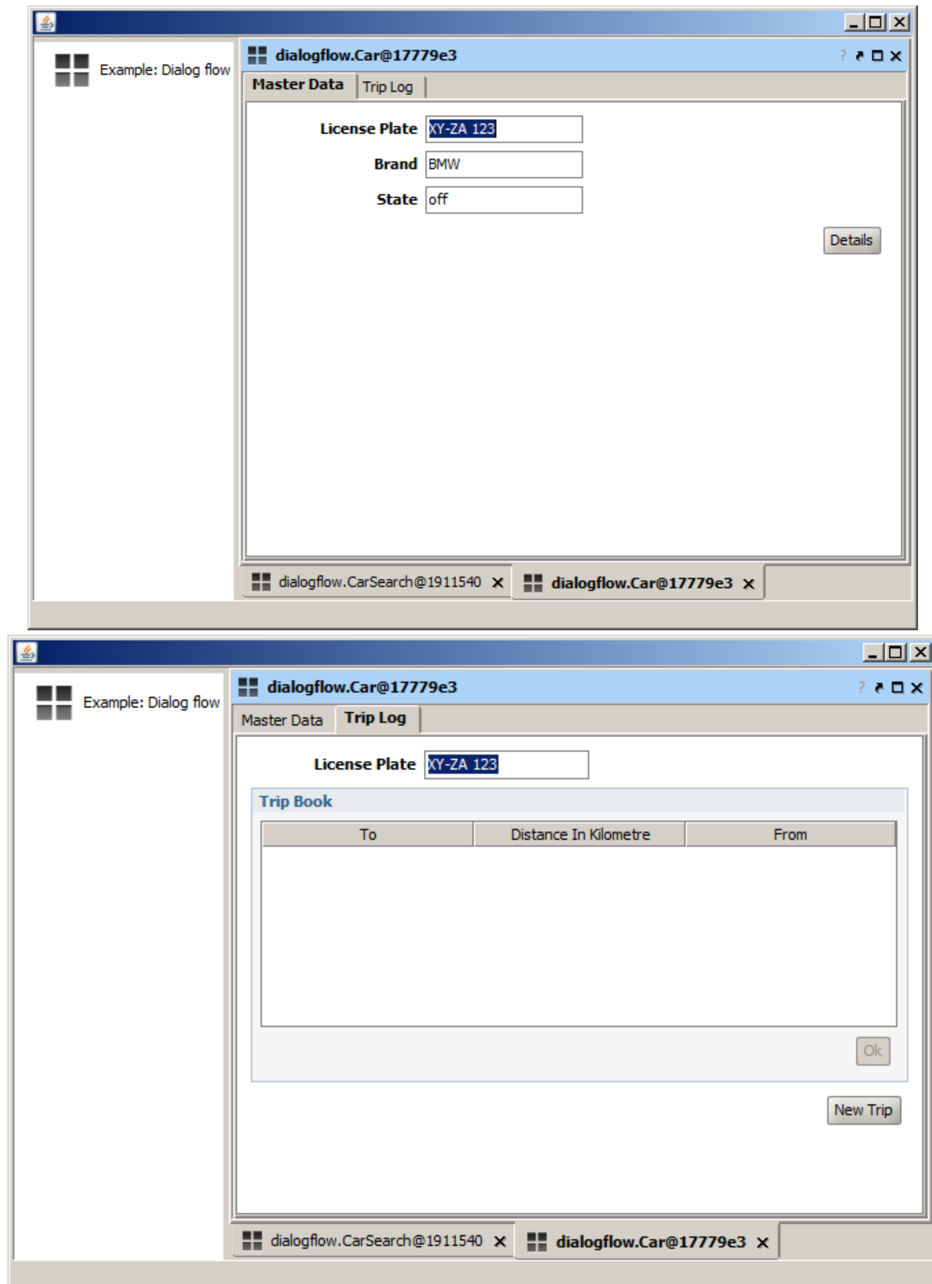
```
        // Dialog flow to display the details of the selected entry in the list
        public Car details() {
                return this;
        }

        // Modal dialog to add a new trip
        public void newTrip() {
                Trip newTrip = ((Trip)guiservice.modal(new Trip()));
                this.tripBook.add(newTrip);
        }
}
```

The method newTrip shows a modal dialog to edit a new trip.
Start of the example:

```
java gengui.infonode.RootFrame dialogflow.CarSearch
```

After having splitted the mask for Cars into multiple windows, the resulting user interface looks like this:





There are some suspicious details in the user interface which require improvement:

- The button “New Trip” appears in both, the CarSearch and the trip log window
- The button “Details” appears not only in the CarSearch but also in the master data window
- The field order in the table and in the modal dialog is awkward and unintuitive – in fact they are randomly arranged
- The captions of the windows have ugly technical names
- …

These things can be improved by various tuning techniques which are covered by chapter 6.

6 GUI tuning

6.1 Disabling of fields and buttons

Two ways of making fields read-only have already been introduced: omit the setter (or declare it private) or setting editable=false in the JFormDesigner. As both alternatives are not always reasonable and don't work for buttons, the framework provides additional features to disable fields and buttons statically (by annotations) or dynamically.

6.1.1 Static disabling

Realization:

A field may be statically disabled by adding the **`@Disabled`** annotation at the corresponding property's setter or getter method.

Effect:

Appropriate fields are displayed non-editable.

If the property is of a structured data type, the disabling appears recursively, i.e. all fields of the referenced objects and their children are read-only as well. However, this is not the default behaviour for structured data types if the setter is missing. So to keep developers from implementing setters just for the sake of having something to annotate, the **`@Disabled`** annotation may also by specified at the getter. Statically disabled structured properties are displayed without any method buttons as these could never be pressed and this don't make any sense.

A status disabling of method buttons is not supported. If a method should never be accessible interactively, it is more reasonable to hide it completely, using the @Hidden annotation.

6.1.2 Dynamic disabling

Analogous to field validation, the enabling and disabling of fields may defined through a disable method based on the object's state. This dynamic approach is also supported for method buttons.

Disble methods must have the following signature:

```
public String disable<NameOfProperty/NameOfMethod>()
```

The returned String specifies the reason fort he disabling of the field / button and will be displayed as a bubble help resp. in the status line of the root frame. A return value of null causes the field / button to be enabled. As for the static variant, the disabling appears recursively in case of structured data types.

6.2 Field order

The **`@FieldOrder`** annotation allows specifying for a class in which order the property fields appear in the mask. This is sometimes necessary because the masks are built from Java reflection traversal of the class structure. It is partially not predictable which field order comes out of this traversal even when it is performed multiple times within the same JVM. Therefore it can be helpful to define the order by an annotation. The annotation gets passed a comma-separated list of properties which also defines the focus traversal when stepping from one field to the other with the tabulator key. Tables for collections and array are generally excluded from the focus traversal. Alternatively the traversal order and the arrangement of fields may be defined independently from the code by means of the layout files.

The configuration parameter **`fieldorder.policy`** in the gengui.properties files allows configuring, if all properties must be present in the annotation. If properties are allowed to lack, the **`@FieldOrder`** annotation becomes an alternative way to hide properties.

The annotation **`@TableFieldOrder`** is similar to **`@FieldOrder`** but defines the order of columns when objects of that type are listed in a table. It also defines which properties are supposed to be displayed at all. E.g. it is allowed to also specify hidden properties (see 6.7) und referenced structured objects in the TableFieldOrder. Structured objects are displayed in form of their toString representation.

6.3 Button order

The **`@MethodOrder`** annonation allows specifying the order of method buttons in the mask. The configuration of **`methodorder.policy`** and the **`@TableMethodOrder`** annotation for the order of

buttons below tables work analogously to their equivalents for the definition of field orders. The same is true for the focus traversal. Buttons and fields of course make up a single focus cycle.

6.4 Automatic closing of masks

The **`@Closer`** annotation can be attached to methods to tag these methods as the end of an interactive process which therefore closes the mask the button is located on. This feature has already been introduced in conjunction with modal dialogs (see section 5.2). In modal dialogs, any method button click closes the dialog by default. This can be avoided by adding the **`@Closer`** annotation only to those methods which are actually supposed to close the dialog.

In non-modal masks, the default behaviour is the opposite, i.e. the window is kept open, when clicking a method button without annotation. Annotating the method with **`@Closer`** will automatically close the window as soon as the corresponding button is clicked.

The annotations also causes window closing if it refers to a dependent object being embedded I the mask of another surrounding object (see section 3.2). This behaviour can be changed by means of decorators (see section 6.23).

6.5 Enabling of buttons in case of validation errors

If a field in a mask cannot be validated successfully, all buttons on the same mask are disabled so that the corresponding methods of the domain object and all of its dependent objects are not accessible. This default behaviour can be changed for particular methods using the **`@Forced`** annotation. Methods being annotated this way can be activated even if a validation for a field on the same mask failed. If such a method is activated, all fields on the mask which currently have invalid input, are rest to the current property value of the underlying domain object.

Another way to decouple buttons states from validation results is to display a domain object on multiple sub windows. (see section **Fehler! Verweisquelle konnte nicht gefunden werden.**). The disabling of buttons in case of validation errors is limited to a single sub window. I.e. if a field validation fails, it only causes the disabling of buttons on the same window. The buttons on other windows are kept unchanged.

If the states of multiple buttons over different sub windows must be synchronized, you must explicitely implement common disable methods.

Usually the disabling of buttons in performed in a graphical form in the sense of Swing/AWT, i.e. the button becomes shaded grey and does not react on mouse clicks and key strokes. This will change if the button recides on a mask with any fields with “lazy synchronization” (see section 6.26). In this case the buttons remains graphically enabled but produces explanatory error messages when being pressed. I.e. it is *logically* disabled.

6.6 Window titles

The window title for a domain object is taken from its toString method. If this method is not implemented but derived from java.lang.Object, the class name is used instead. This is often suitable for the classes which make up the main many. If the toString method is not applicable for a window title, e.g. because it is already used for non-graphical purposes, you may implement a method with the following signature instead:

```
public String title()
```

If this method is present, its result is used as the window title instead. Each title also walks through localization, so you may as well provide only a localization key which refers to a gengui property file. See chapter 8 for more details.

If you want to apply the Java default from the java.lang.Object class (i.e. a String like <package.classname@memoryadress>) just implement the toString method and call super.toString().

6.7 Hiding fields and buttons

Occasionally it is required to hide particular properties or methods which are not supposed to be available in the user interface. The **`@Hidden`** annotation and the @...Order annotations have already been introduced for these cases. Properties expect the @Hidden annotation at the getter method.

Another feature which is related to hinding information is the @Shallow annotation. It prevents the frameworks traversal of object structures from running recursively. As a concequence, only properties of simple data types (see section 3.1) are displayed on the mask.

Method buttons can also be hidden dynamically by appropriate hide methods. These methods must have the following signature:

```
public String hide<NameOfMethod>()
```

the String return type is just defined fort he sake of consistency with disable and validate methods. If the result is null, the method button is displayed, otherwise it is hidden. The layout for the masks which use this feature, must of course be constructed in a way that the mask has a reasonable layout with and without the buttons. This may be a job for the manual layout finishing (see section 6.18). The layout generated by the framework by default buts the buttons in a panel with flow layout which is stylistically robust agains the disappearance of buttons at runtime.

A totally different alternative is the definition of exclusion name patterns through the configuration parameters **`method.exclude.patterns`** resp. **`field.exclude.patterns`** in the file gengui.properties. This technique is recommended if e.g. a general exclusing for stereotypically generated management functions and properties is required. E.g. a byte code instrumentation with aspectJ causes new methods to appear that have not been defined in the application code.

6.8 Removing entries from collections and arrays

The framework provides a naming convention to implement methods being responsible for the deletion of elements from a collection or array. These methods must have the following signature:

```
public void removeFrom<NameOfProperty>(<TypeOfCollection> selectedEntry)
```

If such a remover method is present, the framework displays an appropriate button under the table which allows to remove the currently selected entry. A remover method may be accompanied by a corresponding disable method. An alternative signature is the declaration of an array or ellipsis parameter like that:

```
public void removeFrom<NameOfProperty>(<TypeOfCollection>[] selEntries)
public void removeFrom<NameOfProperty>(<TypeOfCollection>... selEntries)
```

In this case, the corresponding table allows the selektion of multiple entries being all passed to the remover method when the button is pressed. Otherwise the table will only support single-line selection.

6.9 Selecting entries in collections and arrays

The operation `GUIServiceI.select` allows to programmatically select elements of collections and arrays in the user interface. The operation gets passed the array or collection object and the index to select. The selection as applied only once for the next GUI update and is kept automatically as long as the collection contains the selected entry or the selection is explicitly changed – interactively or by a new call of the select operation.

6.10 Displayed label

In certain cases it may be necessary to change the label for a field, a button or a whole domain object window in the user unterface. The label text can explicitely be specified using the **`@Prompt`** annotation. Properties being displayed as text fields (strings, numbers, dates), can also be provided with an example input text by means of the annotation's example attribute. This text will at runtime be displayed with a grey italic font in the corresponding input field as long as the user has not entered anything. E.g. the annotation

```
@Prompt(example="max.mustermann@web.de")
public String getEmail() { ... }
```

causes a visualization like this:

This feature is currently not supported for fields with lazy synchronization (see section 6.26)

6.11 Date formating

Date properties are by default displayed in the format DD.MM.YYYY. A differing format may be defined by adding a @Format annotation to corresponding getter method. The annotation allows to specify the display format in the SimpleDateFormat syntax, e.g. in the following way for displaying a date including the time

```
@Format("dd.MM.yyyy HH:mm")
public Date getOrdertime()...
```

This format is also accepted as an input format in case of editable fields. Instead of directly specifiying a format, the annotation's value may also be a reference on form of a configuration parameter name. The actual format is then defined in gengui.properties and may be internationalized this way.

Another way to set the default date format is to set it via the *dateformat* property in gengui.properties.

6.12 Using shortcuts / mnemonics

Shortcuts are a good way to improve the application's usability for more experienced users. Using shortcuts enables the users to interact with the application in an efficient way without having to switch between keyboard and mouse. Swing provides a simple way to implement shortcut-like features called 'mnemonics'. Mnemonics are quick-access letters that can be added to a JMenuItem or JButton. Setting the mnemonic *'s'* for a JButton with the text *"Save"* will have the result that the 's' in the Button's text is underlined (*"Save"*) to indicate that a mnemonic is set. The user can now activate the button using the *[ALT]* + *[S]* key-combo and does no longer need to use the mouse.

Fortunately JFormDesigner provides the functionality to provide buttons with the desired mnemonic-functionality. Therefore you simply enter the mnemonic key (any letter from the alphabet) in the *"mnemonic"* property in JFormDesigner for a JButton.

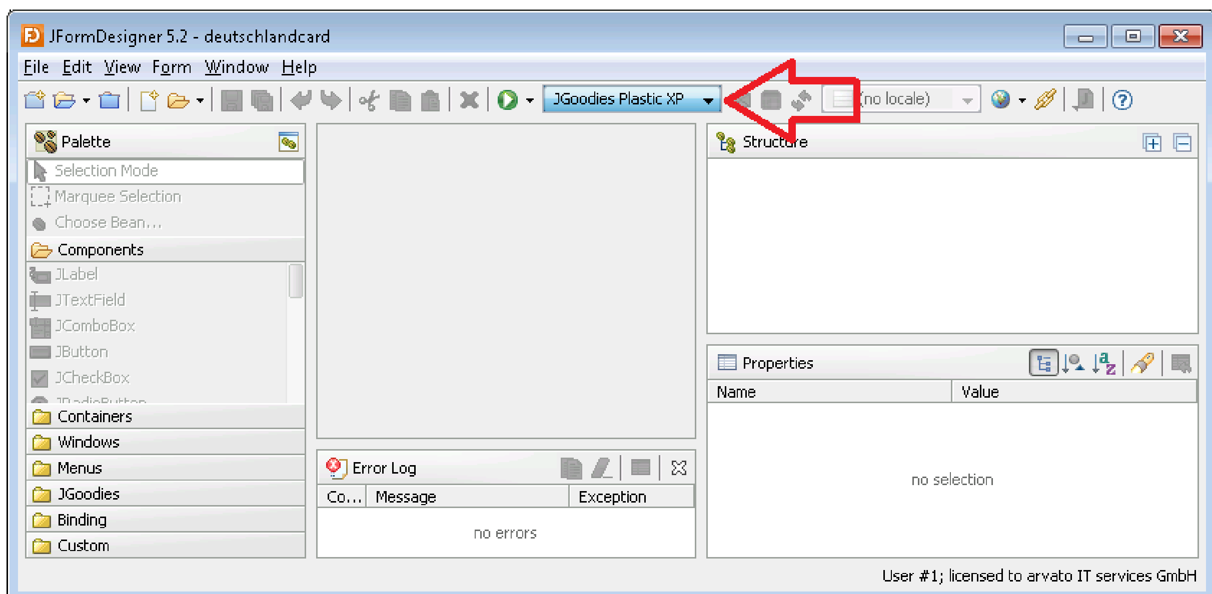
This feature can be a big boost of user efficiency and is achieved with almost no efford and therefore should be used intensely.

6.13 Changing the look and feel

Using a different look and feel can have several benefits. While some simply look better than the plain old grey windows or metal look and feel there are also a few functional enhancements like highlighting of focused input components. To change the look and fell in gengui you can simply set the *look.and.feel* configuration parameter in the gengui.properties file. However, this will not affect the rendering of the InfoNode DockingWindows components (tabs, window titles, etc.) as IDW uses its own mechanism to render its components. In order to achieve a consistent look and feel throughout the entire application you also need to set the *infonode.render.look.and.feel* property to *true*.

This mode is explicitly called "experimental" in the official IDW documentation and therefore should be used with caution. It is possible that rendering problems occur, though they haven't been discovered yet.

If you change the look and feel of your application it is recommened to do that in JFormDesigner accordingly. Otherwise you may apply layout changes which are not suitable at runtime.



6.14 Setting tab- and view/menu icons

You can set icons for tabs and views in gengui. These icons help the user to recognize tab and view contents easily and lead to higher usability. The icons can be set in the gengui.properties, where you can set domain class-specific icons as well as a default icon for all the other classes that have no specific icon.

Tab icons (*.thumbnail*) and view/menu (*.icon)* icons are set separately as they usually have a different size. To achieve a consistent look and to preserve usability you should mind that the icons are the same and only differ considering their size.

6.15 Changing the default layout

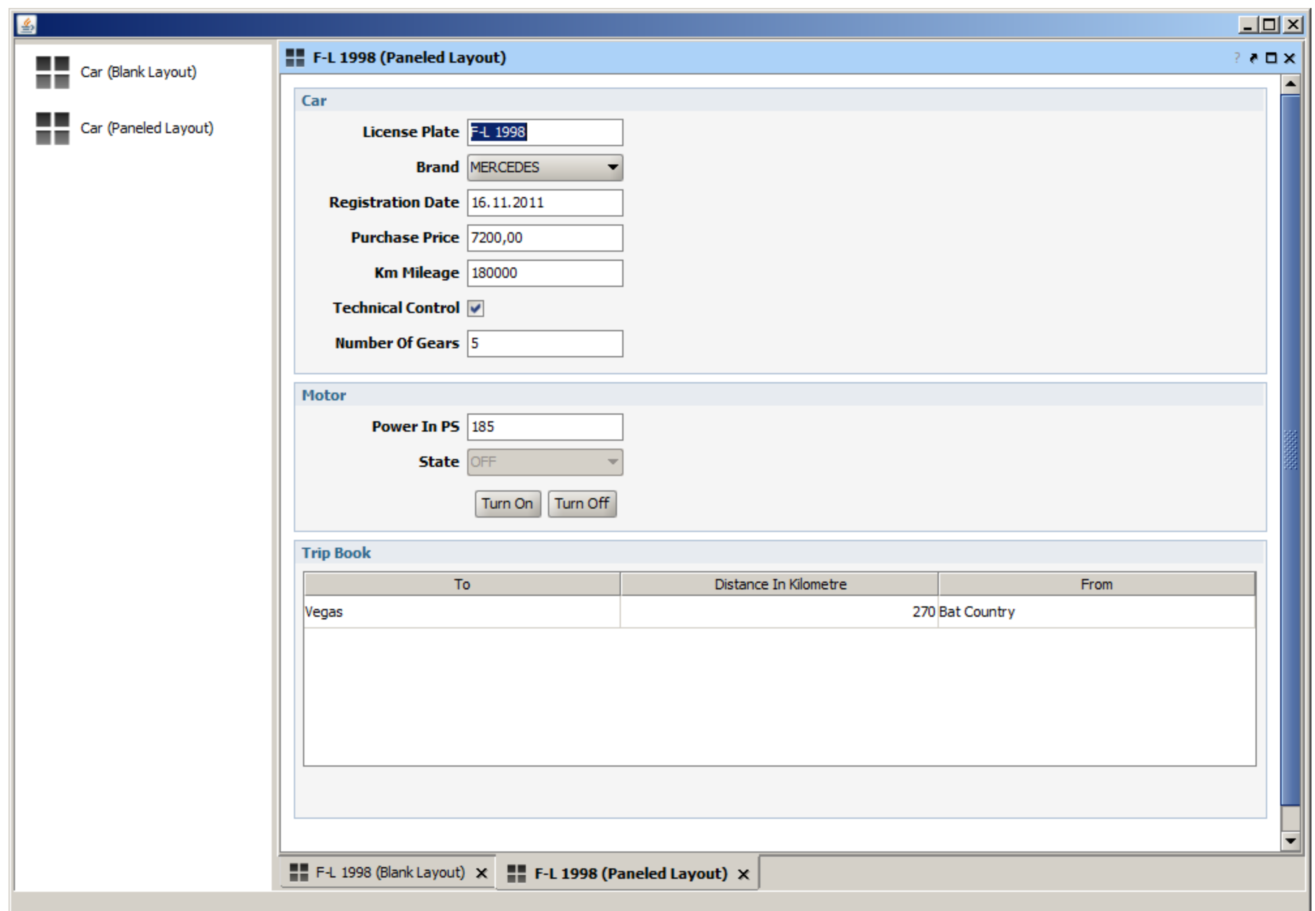
Gengui provides a way to define colors and several layout-specific parameters in the gengui.properties. To alter the design of generated layouts you need to change the *color…* and *formlayout…* configuration parameters. Gengui uses JGoodies FormLayout as layout manager. The constraints you need to set are FormLayout-specific and can be looked up in the official JGoodies documentation.

You can also find two examples of a panelled and a blank layout in examples-manual/layout

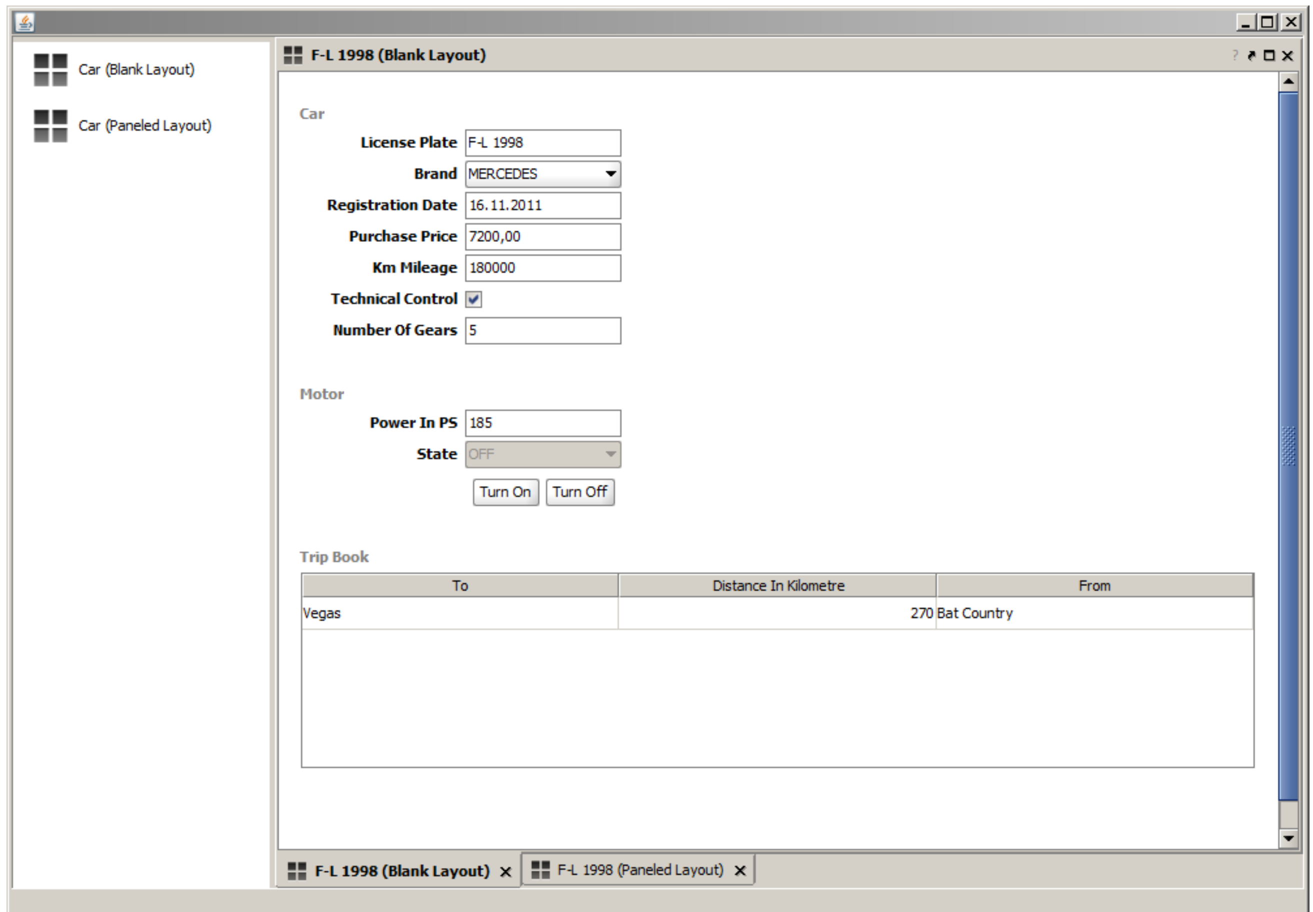
6.16 Example (examples-manual/layout)

The package *layout* contains two sub-packages which each presents a possible design of gengui's layout. The corresponding gengui.properties files are *layout_blank_gengui.properties* and *layout_panelled_gengui.properties.* Those two property-files differ in their constraints for the FormLayout as well as the colors. While the panelled layout uses blue shades to visually distinguish the individual input component groups, the blank layout paints everything in the same color and uses wider and narrower space between individual components to group them visually by their closeness.

An example can be seen here:



Panneled layout (gengui default)



Blank layout

Of course both layouts can be reworked manually to achieve even better results. The pictures above only depict gengui's standard output in order to visualize the differences between two alternatives. You can go ahead and experiment further with different color- and layout styles to achieve results that go with your customer's taste.

6.16.1 Coloring schemes

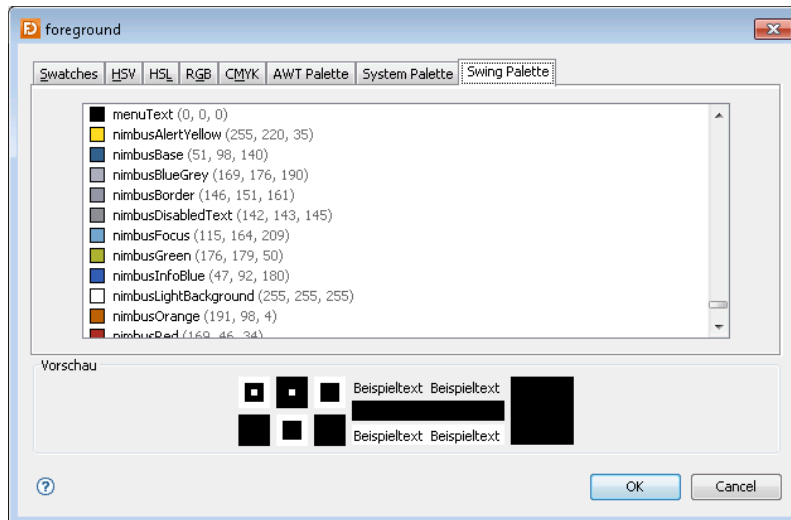
The framework allows to specify colors in the following different ways:

- By RGB color values in hex notation like *color.background.error=0xFFEEEE*
- By RGB color values including alpha value like *color.background.error=0xFFEEEE44*
- By using Swing color names like *color.foreground.line=GRAY*
- By referring system colors like *color.background.prompt=InternalFrame.activeTitleBackground*

The latter form is of interest if you need to prepare yourself for recurring coloring scheme changes. As a few of the colors are copied from gengui.properties to the generated layout files, changes in the color scheme may require to modify the existing files. You can avoid that by referring to system colors which you may override programmatically by calls like that in your application initialization:

```
UIManager.put("InternalFrame.activeTitleBackground", Color.YELLOW);
```

You may also refer to colors of the look and feel of your choice. If you have configured your look and feel in the JFormDesigner you find all its colors in the Swing Palette tab of the color picking dialog. E.g. when using Nimbus look and feel you will find according Nimbus colors to choose from.



The web page https://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/_nimbusDefaults.html explains whet all these colors are good for in Nimbus. Other look and feels don't provide such a comprehensive description, so you may have to play around a little.

6.17 Splitting up a mask

If the content of a domain class is too voluminous to be displayed on a single screen, the user interface can be split into multiple sub windows. Beside the feature of manual splitting of layout files as described in section 6.16.1 the methods and getter methods of properties may also be annotated with the **@Group** annotation to separate them into logical groups. Each group will be displayed in a separate sub window and will get a separate layout file created (if file creation is configured). Properties and methods which are not associated to any group, make up a default group. When an object with groups is displayed, the group windows are tabulated one over the other and the group names make up the tabulator captions. The default group gets the name of the class or the content of the class' Prompt annotation if present. The sub windows are displayed in a surrounding window within which the sub windows can be freely arranged by the user. As a difference to top-level windows the group windows can't be closed separately and they cannot be undocked.

The visibility of sub windows may depend on the object state and for this case can be controlled by a method having attached the **@GroupChoice** annotation The method must return an array of Strings which refer to the identifiers of the class' **@Group** annotations. The sub windows the identifiers of which are not present in the String array will be removed from the screen. The default group is adressed by a null value in the array.

6.18 Manual GUI tuning

In case the generated layout for a mask is not sufficient (which is a usual case!) the layout may subsequently be revised using the JFormDesigner. The configuration parameter **`jfd.retention.strategy`** in gengui.properties must be set to the value **`keep`**, **`merge`** or **`silentmerge`**, to cause the framework to actually create layout files and to keep it from overwriting these files. The layout is stored in files of type .jfd which can be loaded and edited with the GUI builder tool JFormDesigner. For domain objects with multiple windows it is important to ensure that all layout files are present to overcome a regeneration of files.

Within JFormDesigner the following GUI modifications may be beformed without any risk:

- Moving fields / buttons
- Enabling / disabling of fields / buttons
- Making UI elements invisible (not removing them!). Just be sure not to hide a *mandatory* input field which may initially be empty and therefore requires interactive input to by the user to go on.
- Adding tooltips
- Definition of focus cycles for fields and buttons using *nextFocusableComponent*. This is actually a "deprecated" feature of Java but it is a very helpful approach for generic GUIs. Its availability is ensured up to Java 8. To ensure that the focus traversal is actually recognized at runtime, the outer panel must be tagged as the focusCycleRoot and must refer to the first

component of the focus cycle in *nextFocusableComponent*. The referred component is the one which initially gains the focus when a mask is displayed for the first time.

- Definition of a window's default button by setting the Swing property defaultCapable to true. The button may refer to a method of a sub object but must not refer to a collection method.
- Definition of the double-click button for collections / array. By default, the first method of a domain class being contained in the collection, is accessible by double-clicking a row in the table. The "first" method is the one which occurs at first in the @MethodOrder or @TableMethodOrder annotation of the listed object's class. Alternatively you may set the Swing property *defaultCapable* to false for all buttons but the one which is supposed to be activated by a double click. The double click button may by invisible to allow a method only be activated by double click.
- Definition of table columns. JFormDesigner allows to provide a model for a table, which at runtime the framework adopts the number of columns, the order and the width ratio rom. The column captions of the table model must match the names of the domain class' properties – the actual captions may vary from that at runtime if the corresponding property has a @Prompt annotation attached. Possible exemplary content of the model is of course ignored at runtime and substituted by the actual content of the collection / array. If a model is defined in the layout file, it overwrites any other definitions by FieldOrder or TableFieldOrder annotations in the code.
- Deferring mapping of property types to field types. The following alternatives are supported:
    - JList instead of JTable
    - JComboBox instead of JTable
    - JPanel instead of JTable
      This causes the reduced read-only visualization of the collection elements to be substituted by their full-fledged visualization. The arrangement of the inner panels is defined by the panel's layout manager. Reasonable managers for this purpose are only VerticalLayout, HorizontalLayout, FlowLayout or BoxLayout. An example for this powerful feature is the styled and complete to-do list application from the first steps introduction [FSI]
    - JTextArea instead of JTextField
      This makes sense for long texts and especially multiline text. As a disadvantage, a JTextArea by default can't be left by pressing the Tab key as it is usual in JTextField. However, leaving text fields is an important feature as it is the indicator for gengui to synchronize the input to the underlying domain object. As a trick, you may change the field's tabSize property to 0 in the JFormDesigner. This tells gengui to use the Tab key for changing the focus rather than as input. Most texts don't need tabulators.
    - JTextPane instead of JTextField
    - JRadioButton instead of JCheckBox.
      Radio buttons have the advantage over check boxes that they are able to distinguish Null from False. They are therefore well-suited for properties of type `Boolean`. When exchanging a check box by a radio button you must add second radio button which represents the False value. The name of the radio button must be the same as the first button, followed by an arbitrary dot-separated suffix, e.g. „.false".
    - JRadioButton instead of JComboBox
      This is currently only supported for enum properties. In this case you must add one radio button for each possible enum value. The names of the buttons must be the same as for the original button followed by `.<Enumconstant>`. The first button represents the null value. If null should not be supported, you may hide that button.

  ATTENTION: whenever you change the type of a component, the name must be preserved (i.e. the Swing property name, see section 6.17). The easiest way to change the component type is by the morph-bean-feature of JFormDesigner which is accessible by clicking the field of interest and opening a context menu with the right mouse key.

The following things must never be done with the JFormDesigner in generated layout files:

- Removing GUI components (except when manually splitting up a mask into multiple windows, see section 6.18)
- Renaming GUI components

When using JFormDesigner for pur layout file tuning, it is recommended to switch off the code generation option (Window → Preferences, tab Java Code Generator, checkmark "Generate Java

source code" off). Otherwise the tool will always generate a corresponding Java source file whenever saving a layout file. Generating source code is only of interest in very special cases (see sections 7.6 and 12.2).

6.18.1 Bringing manual and automatic changes together

If a layout has once been automatically created and stored as a layout file, what is going to happen with subsequent extensions of the domain classes? Especially when the layout files have extensively been tuned, the manual work should of course not get lost. On the other hand, the framework cannot work with out-dated layouts which are lacking fields for new properties.

To solve this problem, the framework allows to automatically extend existing layout files. The configuration parameter `jfd.retention.strategy` in gengui.properties defines how the framework is supposed to work with layout files. A notable operation mode here is **`merge`**. If a layout file already exists but is lacking fields for new properties in the domain class, the framework automatically adds these fields. The required fields are appended at the bottom of the existing layout in a separate panel. The placement of the new graphical components is usually not suitable for immediate use but requires manual finishing so that the components appear at a semantically reasonable position. Nevertheless the merge mode saves the work for manually *adding* new fields but reduces the job to simply *moving* them around.

A similar alternative is the retention mode **`silentmerge`**. It also causes missing fields to automatically be added to existing layout files, but they are kept invisible. This mode is of interest when a project is near to its end and new properties are usually not supposed to appear in existing layouts. As most of the framework's configuration properties, the retention strategy may be defined globally, per package or per class such that you may use different strategies for differently evolved parts of the application (see section **Fehler! Verweisquelle konnte nicht gefunden werden.**).

6.19 Input assistence

The **`@Assisted`** annonation allows to express that editing a property should be eased by input assistents. Currently the framework supports a calender component for date properties and a file chooser dialog for file properties. The annotation may also be available for different types of properties. E.g. editing a numerical value may be assisted by a slider.

6.20 Detection of interactive changes

The GUIServiceI interface allows to programmatically find out if a domain object has interactively been modified. The application may use that e.g. to enabled/disable methods or to prevent a dialog from getting closed. The operation

```
public boolean GUIServiceI.touched(Object domainObject)
```

returns true if the setter of a property of the passed domain object or one of its dependent objects was executed due to a graphical interaction. If the content was actually changed, is out of the operation's scope. However, the framework usually keeps from invoking setters at all if a current input value of a field does not differ from the domain object's current property value. The operation also return true if there is pending invalid input present in a any of the input controls for the domain object.

The operation

```
public void GUIServiceI.untouched(Object domainObject)
```

allows to reset the framework's internal update flags for the passed domain object, thus an immediate subsequent call of `touched()` would return false.

6.21 Example (examples-manual/pimpmygui)

The following example shows some of the above concepts in action.

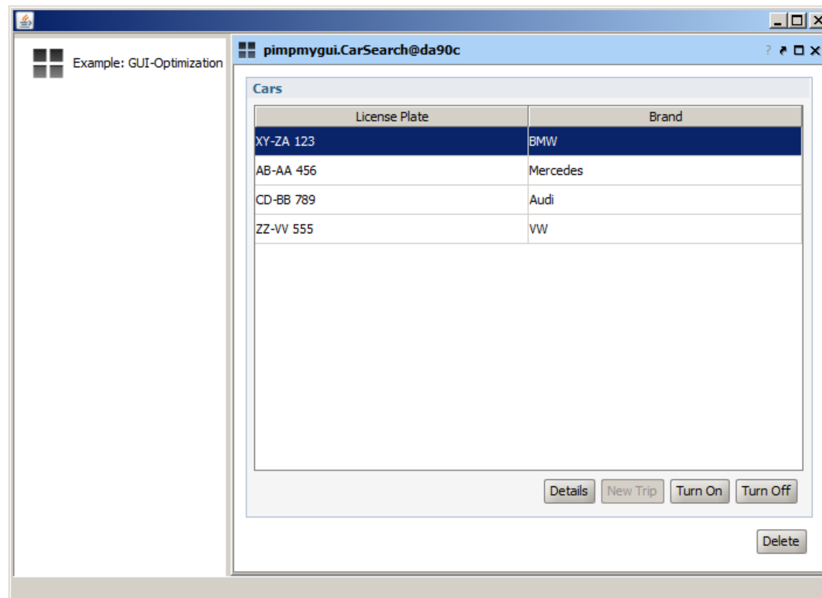
```
// Field order for both detailed display and table display (see section 6.2)
@FieldOrder("LicensePlate, Brand, State, TripBook")
@TableFieldOrder("LicensePlate, Brand")
public class Car {
```

```
	private MotorState state;
	private String licensePlate;
	private String brand;
	private String serialNumber;
	private Collection<Trip> tripBook;

	//Constructor and various getters/setters omitted here

	// Hidden property which is not visible in the user interface (see 6.7)
	@Hidden public String getSerialNumber() { return serialNumber; }

	// Read-only property which should only be modified by
	// using the start() and stop() methods (see 6.1.1)
	@Disabled
	public void setState(MotorState state) { this.state = state; }

	// Dialog flow from the car search to the car details view
	// Will be be hidden in the detail mask by manual finishing (see 6.18)
	public Car details() { return this; }

	public void newTrip() {
		Trip newTrip = ((Trip)guiservice.modal(new Trip()));
		this.tripBook.add(newTrip);
	}

	// New trips require the engine being started
	public String disableNewTrip() {
		if (state == MotorState.OFF) {
			return "Start the engine before you drive.";
		}
		return null;
	}

	// No matter how inconsistent the car's data may be, you can always stop the motor (see
6.5)	@Forced
	public void turnOff() { state = MotorState.OFF; }

	// A horribly complicated toString method which is not suitable as a caption in the GUI
	public String toString() {
		// Something very complicated is happening here, e.g. concatenation of all
properties
	}

	public String title() { return "Car: " + licensePlate; }

}
```

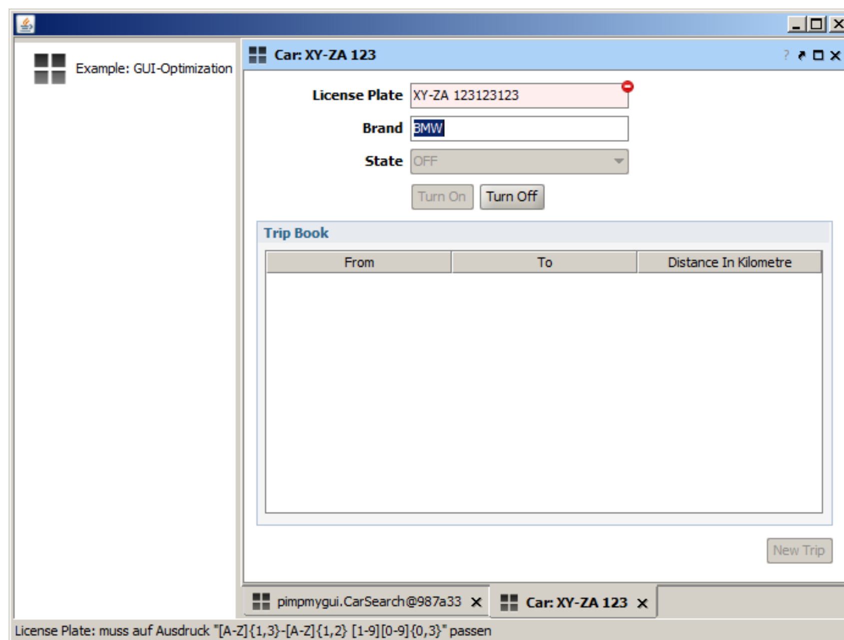
Start of the example:

```
java gengui.infonode.RootFrame pimpmygui.AutoSuche
```

The resulting user interface looks like the following. The buttons for “New Trip”, “Start”, and “Stop” would by default also be displayed beneath the table but they have been hidden by manual layout finishing. The table doesn’t include all the cars’ information any more but only the properties being listed in the TableFieldOrder annotation.



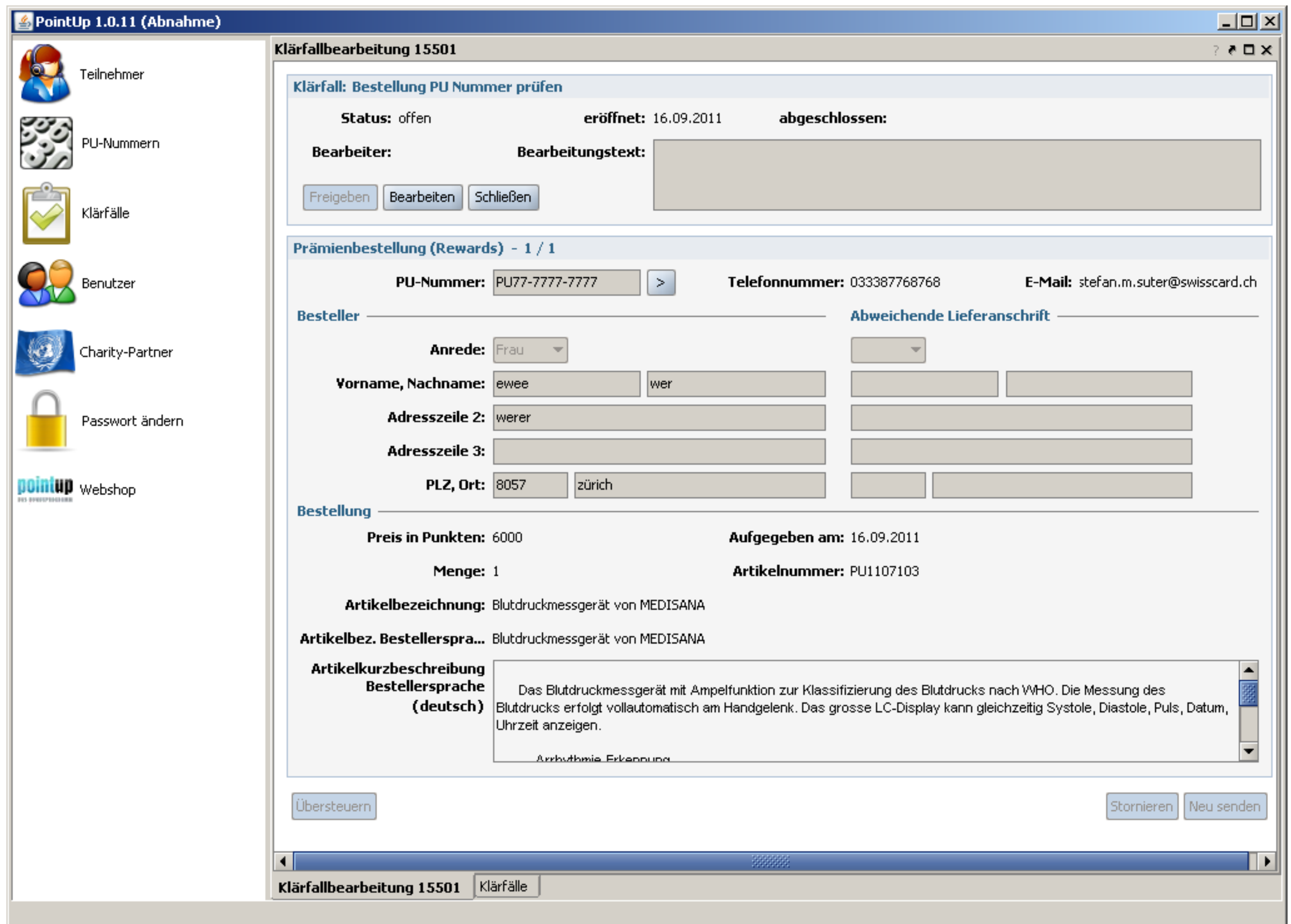
The detail mask has also been manually finished. The buttons “OK” and “Details” for the dialog flow have been hidden here. The buttons “Start” and “Stop” were placed directly under the state field, as they only modify the corresponding property.



The „Stop“ button is enabled allthough the current input is not completely valid (due to the @Force annotation).

The button “Add Trip” ist disabled / enabled depending in the car’s motor state.

A tuned user interface may look really nice and ergonomic. The following example shows a tuned mask for a complicated domain object from a real-world application.



6.22 Separate GUI classes

GUI tuning may end up in domain classes being blotted with a lot of GUI-related annotations. It may also contain GUI specific methods being concerned with user interface control (enabling, validation, dialog flow). E.g. in the last example the bean depends on the GUIServiceI interface to display a modal dialog.

In these cases it may be reasonable to free the domain class from GUI specific aspects and create a separate view class instead. These classes may be associated to domain classes by means of the `@Viewtype` annotation. The annotation gets passed a class which should be used as the graphical representation of the annotated class.

The view class serves as a wrapper around the actual domain class and it must provide a constructor with a parameter of the wrapped class' type or one of its base classes. In the simplest form, the view class simply has a getter and a setter for the wrapped Java bean. However, the view class will grow as soon as GUI specific aspects are moved from the domain class to its view class. E.g. the view class may have a delegate method for each method of the wrapped Java bean (including all getters and setters) so that the GUI annotations can be moved to these methods. In this case, choice and disable methods can also be implemented in the view class rather than the wrapped bean. In this regard the @shallow annotation may be of special interest to hide the wrapped class' sub structures and selectively display particular details aor split the display into multiple sub windows.

A view type may not only be specified in general for a class but also selectively at getters methods returning this type of object. This is helpful e.g. to display a class differently depending on the context (e.g. addresses of customers and addresses of orders). In this case the view type instance is kept alive be the framework as long as the getter method returns the *identical* (not just an *equal*) instance of the domain object. If the getter's return value changes, the framework automatically creates a new viewtype instance. Viewtypes being applied to getters should therefore be implemented stateless. The constructor parameter type of these viewtypes must correspond to the getter's return type. This allows also to wrap classes which are not implemented in the application itself and therefore cannot be instrumented by a @Viewtype annotation. This also allows to to wrap primitive types or collections and arrays.

6.23 Decorators

If you don't want to wrap all methods of the domain object in a view class, you may use decorators. This feature allows to override only individual aspects of the referenced class. A decorator substitutes the decorated method of a referenced object when being displayed in the user interface. E.g. when decorating the getter of a referenced class, the framework does not display the result of the referenced object's getter method but the result from the decoration method.

Decorators may be applied to all methods and all properties of a referenced class and is defined by the **@Decorate** annotation. If a decorator refers to a property, the annotation gets passed the property name[3]. This is allowed for both getters and setters.

Disable, hide, validate and choice methods refer to properties or methods and can be decorated in the same way. The decorated class may contain these control methods but it's not a must. Especially in respect of separation of GUI specifics, this feature is of particular interest, because it allows e.g. to implement a GUI-related disabler in a view class which doesn't have to be present in the domain class at all. If a decorator is defined, any static annotation-based constraints are taken from that method. This allows e.g. to hide properties of referenced objects by a getter decorator with the annotation @Hidden. Possible constraint annotations must be taken over from the original method to the decoration method if they are still supposed to apply. The annotations of the original method are no longer considered if a decorator is present.[4]

The **@Decorate** annotation can also be defined for methods other than getters and setters. In this case, clicking the appropriate button will no longer call the referenced object's method but the surrounding object's decoration method. You must keep in mind that the developer is responsible for the decision wether the decorator method must call the wrapped method or not. The developer is also responsible to ensure that differing constraints from the decorator method do not compromise the wrapped object's functionality. E.g. the rules of the Liskov substituition principle should be observed [LIS].

Decorators are a strong feature which e.g. solves the problem of the Weinhauer Address Dilemma. The address dilemma sketches the following problem:

There is a structured domain object (a postal address). The surrounding object (e.g. an order) knows the embedded object and is interested in certain modifications of it (recalculate the shipping costs depending on the country). The embedded object however does *not* know its surrounding object and therefore can't notify the outer object if its changes. Furthermore address objects also occur in different contexts (e.g. customer addresses) where other rules apply.

The solution is a decorator. The outer object decorates the setter method which is interested in (e.g. setCountry). This method is called by the framework on interactive modifications and the implementation of that method can react on these modifications as required.

6.24 Method with parameters

The framework currently does not support the interactive execution of methods with parameters. There are no buttons generated for these methods. However, there is a very simple pattern for this situation. The developer creates a factory class for every method with parameters which is supposed to be graphically accessible. This class serves for gathering the required parameters and an instance of that class is displayed as a modal dialog for interactive entry. The result of the modal dialog is passed as parameters to the method.

As the parameter factory class is usually a pure view class, it is recommended to encapsulate the pattern in a view wrapper and keep the actual domain object free from these GUI-related aspects.

6.25 Example (examples-manual/pimpmyseparategui)

The following example demonstrates the concepts from the sections above. The Car class was freed from GUI-related aspects. In the user interface it is represented by the CarView class as specified in the ViewType annotation:

```
@Viewtype(CarView.class)
public class Car {
	// Implementation omitted here
```

[3] Keep the capital first letter für properties in mind here (see section 2.1.1)

[4] This is a subject of future extension as it is often helpful if the decorator could derive some constraints from the original method

```
        // Method with parameter
        public void addTrip(Trip trip) {
                this.tripBook.add(trip);
        }
}
```

The interesting part is the implementation of the view class. The class wraps a Car and provides it for the user interface by a getter and a setter. The dependency on the GUIServiceI interface and the disabling of buttons has moved to the view class. The method `newTrip()` is the wrapper for the Car class' method `addTrip(Trip)` which requires a parameter of type Trip. Within the newTrip() method you can see how the factory class TripFactory is displayed as a modal dialog to.

The decorators demonstrate how method calls on the wrapped class can be decorated (setLicencePlate) and how a disabled functionality can be added to a property.

```
public class CarView {

        private Car car;
        transient private GUIServiceI guiservice = new GUIAdapter();

        public CarView(Car car) { this.car = car; }

        public Car getCar() { return car; }
        public void setCar(Car car) { this.car = car; }

        // Parameterless method wrapping a method with parameters
        // Uses the factory class to assemble the parameter
        @Modal
        TripFactory public void newTrip() {
                return new TripFactory(car);
        }

        // Disabling
        public String disableNewTrip() {
                if (car.getState() == MotorState.OFF) {
                        return "Start the engine before you drive.";
                }
                return null;
        }

        // Decorate the modification of the car's licence plate
        @Decorate("Car.LicensePlate")
        public void setLicensePlate(String licensePlate) {
                System.out.println("Decorator called");
                car.setLicensePlate(licensePlate);
        }

        // Decoration method for disabling interactive state changes.
        // This method is not present in the car class, but decoration is allowed anyway
        @Decorate("Car.State")
        public String disableState() {
                return "You can not set a state";
        }

}
```

The factory class is absolutely straight forward:

```
public class TripFactory {

        private Trip t;
        private Car c;

        public TripFactory(Car c) { this.c = c; this.t = new Trip(); }
```

```
        // Required for closing the modal dialog
        public void ok() { c.addTrip(t); }

        // Empty prompt avoids separation line in the dialog (just GUI tuning)
        @Prompt("")
        public Trip getT() { return t; }

        public void setT(Trip f) { this.t = f; }
}
```

6.26 Changing the synchronization behaviour

By default, the developer doesn't have to care about the synchronization of user interface and domain objects and vice versa. This is performed automatically by the framework right after committing the input for a field or pressing a method button. This is the simplest and safest strategy but it may lead to inacceptable bad responsiveness in client-server environments where the application logic is separated from the representation and accessed remotely. This effect becomes particularly obvious when using generic GUIs for a web application: the default synchronization behaviour leads to a server call for every single interactive modification, e.g. whenever the user steps from one field to another. This is usually not even affordable in fast intranet applications and if the synchronization is AJAX-based.

For internet applications and special time-consuming operations, the annotations **`@Eager`** and **`@Lazy`** allow to controll the synchronization behaviour between user interface and domain objects. The annotations are attached to the setters of properties where **`@Eager`** refers to the default behaviour of immediate synchronization per field. The annotation **`@Lazy`** delays the input synchronization until a method button is pressed. If there occur validation errors during synchronization for any of the fields, the method call will be denied with an appropriate error message.

The @Lazy annotation should of course only be attached to properties which can be modified without having an impact on other graphical elements. E.g. if the input for a zip code is supposed to trigger the automatic auto-update of the corresponding city name, the graphical behaviour is only reasonable if the auto completion takes place right after committing the zip input. But in most cases there is no such direct dependency of input and output.

Beside the explicit annotation of individual properties, the default behaviour can be defined by the configuration parameter **`synchronization.mode`** in gengui.properties (resp. in a package- or class-specific configuration file). E.g. web application should usually be configured for lazy-synchronization by default and define individual exceptions by @Eager annotations.

For eager-synchronized properties, the framework confirms an important detail concerning the update order: the framework will first synchronize all *other* properties which required an update, and then will call the setter of the property which caused the eager synchronization at the very last. I.e. the update logic within the eager property's setter-method can rely on all other attributes being already in sync with any (valid) input from the GUI. This is especially of interest for properties which are interrelated. The one being interactively modified at last, is the one that rules.

The lazy synchronization also has a side effect on the disabling of buttons: if a button resides on a mask which has at least one lazy-synchronized input field, the button is always kept enabled from a graphical point of view. However, as long as there occur relevant input validation errors, a click on the button will not call the method behind it but produce an explanatory error message. The reason is the fact that lazy fields synchronize the input at the time a button is pressed, so the buttons must of course be kept clickable. This effect on the button disabling doesn't make a difference from a logical point of view, i.e. the method associated with the button will only be called under the same integrity conditions which apply on masks where all input is eagerly synchronized.

A special meaning of the synchronization annotations applies for the buttons of collections. These buttons are usually enabled and disabled based on the current selection from the collection. This may cause selection changes (triggered by clicking around on the table or list) to take an inacceptable long time. For these cases the getter for the collection may be annotated with **`@Lazy`**. This will cause the buttons to be kept permanently enabled but producing an error message when there is no element selected from the collection or when the selected element considers the method behind the button to be disabled (i.e. the corresponding disable method returns a text rather than null). Annotating the

getter is important as the same annotation at the setter has a different meaning[5]. Additionally the setter may not be present at all.
For collection buttons apply the same defaults as defined by the configuration parameter **synchronization.mode**. I.e. if lazy synchronization is configured by default, the collection buttons are not graphically disabled, but you can switch this on selectively by adding **@Eager**.

The @Eager annotation may also be equipped with an integer value which causes an automatic synchronization after a period of idle time. The value determines the idle time in milliseconds. This feature is interesting for addressing unexperienced users who may not know that they have to press the Tab key to leave an input field – especially when there are only disabled buttons around waiting for the first input to be committed.

Synchronization behaviour by the annotations @Eager and @Lazy is propagated down the reference tree in the domain object hierarchy. I.e. if you want all the input fields of one object to be auto-synchronized, add an @Eager(500) to the object's setter rather than to all its attribute setters. This is especially helpful when related domain classes can't be modified or if the synchronization behaviour should defer, depending on the context, the object is displayed in.

[5] Usually this makes no sense for collection properties at all. However, in combination with a choice method the property would be represented by a combo box and the synchronization of it would be defined by the annotation at the setter.

7 Integration with hand-written GUIs

It may happen that a mask is so complicated that it cannot be created in a generic way. In these cases, the framework allows to integrate hand-written user interfaces as well. Before you actually take this last fallback, you should check the chance of combining generic GUIs with hand-written customizations. This is described in section Kapitel 7.6, however, this section here makes up the basis.

7.1 Prerequisites for hand-written GUIs

The hand-written GUI must be derived from JPanel as it will be displayed within the application's root frame. Additionally it must provide a constructor with the following signature:

```
public MyView(<Type of domain object to display> domainObject)
```

The parameter may be of exactly the same type as the domain object to display or any of its base classes. If the mask is supposed to be displayed as a modal dialog, it must provide the following constructor:

```
public MyModalView(<Type of domain object> domObject, ModalOperation mo)
```

The view class gets passed a ModalOperation object which is required to close the dialog later on. The view must call the operation `end(Object returnValue)` for this purpose. The passed value is what the caller gets as return value from

```
public Object GUIServiceI.modal(Object domainObject)
```

7.2 Integration

7.2.1 Non-modal mask

Domain classes which need to be displayed by a hand-written rather than a generic GUI are annotated with the **`@Viewtype`** annotation as explained earlier. The hand-written GUI class is passed as parameter to the annotation. At runtime the framework will instanciate the view type when an object of the annotated type must be displayed. JPanel-based viewtype annotations are currently not supported for getters as described in the section about *Separate GUI* classes but are restricted to the class level for top-level visualizations.

7.2.2 Modal dialog

For modal dialogs you just use the same **`@Viewtype`** annotation and simply pass the domain object to GUIServiceI.modal (see 5.2)

7.3 Dialog flow to a generic GUI

7.3.1 Non-modal mask

Moving from a hand-written to a generic follow-up mask requires the GUIServceI interface in case of non-modal masks. The method

```
public void GUIServiceI.display(Object domainObjekt, Object referenceObj)
```

allows to display a domain object with ist generic representation. The second parameter is a GUI component from the hand-written panel ort he panel itself. This parameter is used to place the new window on top of the initiating hand-written GUI. Passing null causes the generic mask to appear anywise.

7.3.2 Modal Dialog

Returning from a modal dialog to the calling generic GUI is achieved by calling the following method on the passed `ModalOperation` object:

```
public void end(Object result)
```

The parameter for the `end()` method is the return value which the caller gets returned from the `modal()` operation of GUIServiceI. The caller desides how to process the return value.

7.4 Multiple hand-written sub windows

If a domain object should be displayed by multiple hand-written sub windows, the class can be annotated with the **@Viewtypes** annotation (ATTENTION: plural!). The annotation gets passed an array of classes where each class represents one sub window. The user interface is constructed from instanciating every listed classes passing the underlaying domain object in the constructor. The panels will then be displayed in a surrounding window within which the panels can be freely arranged by the user.

7.5 Example (examples-manual/selfmade)

The following code snippet shows the relevant parts of a hand-written GUI.[6] You can see the constructor getting passed the domain object to display, and the implementation of an actionPerformed() method containing the return to a generic GUI.

```
public class CarSearchView extends javax.swing.JPanel {
        // GUI code omitted

        private CarSearch search;

        // Constructor
        public CarSearchView(CarSearch search) {
                this.search = search;
                initComponents();
        }

        private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
                Car selectedCar = (Car) jComboBox1.getSelectedItem();
                new GUIAdapter().display(selectedCar, this);
        }
}
```

The following class declaration shows how the hand-written GUI is associated with the domain class by the **@Viewtype** annotation:

```
@Viewtype(selfmade.CarSearchView.class)
public class CarSearch {
        // Content of the class omitted
}
```

The Trip class (annotated with Viewtype too) has a hand-written view class as well which is designed for modal display. This combination is shown in the following two code snippets:

```
public class Car {

        private Collection<Trip> trips;

        // Create a new trip using a modal dialog
        public void newTrip() {
                trips.add((Trip)new GUIAdapter().modal(new Trip()));
        }
}

public class TripModalView extends javax.swing.JPanel implements ActionListener {

        // GUI stuff omitted

        private Trip trip;
        private ModalOperation modalOperation;

        // Constructor for a modal hand-written view
```

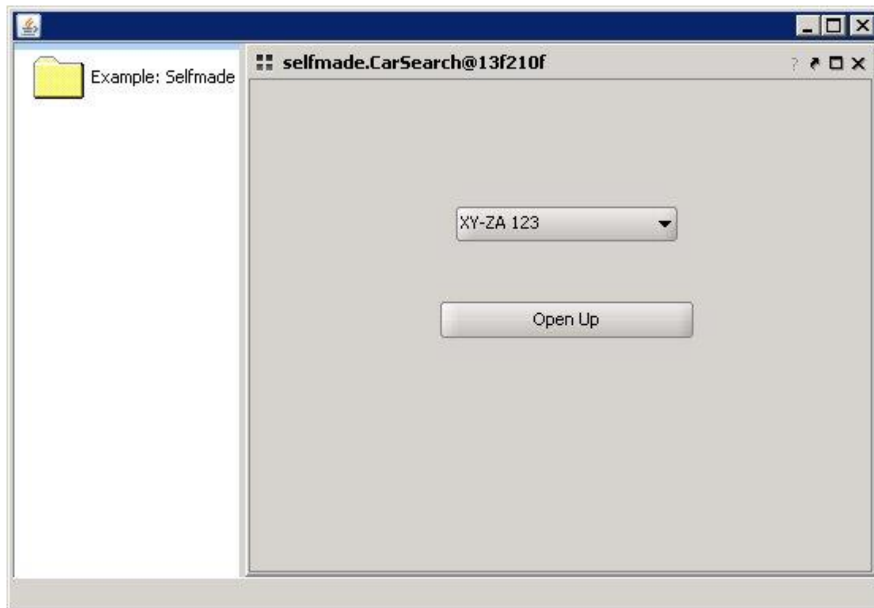
[6] DThe panels in these examples have been created with NetBeans. The acompanying files of type .form are available in directory examples-manual/selfmade, so that you can make you own experiments with these.

```
        public TripModalView(Trip f, ModalOperation mo) {
                this.trip = f;
                this.modalOperation = mo;
                initComponents();
        }

        // Return to the caller using the end() method
        // The Trip object was interactively modified before.
        public void actionPerformed(ActionEvent arg0) {
                modalOperation.end(trip);
        }
}
```

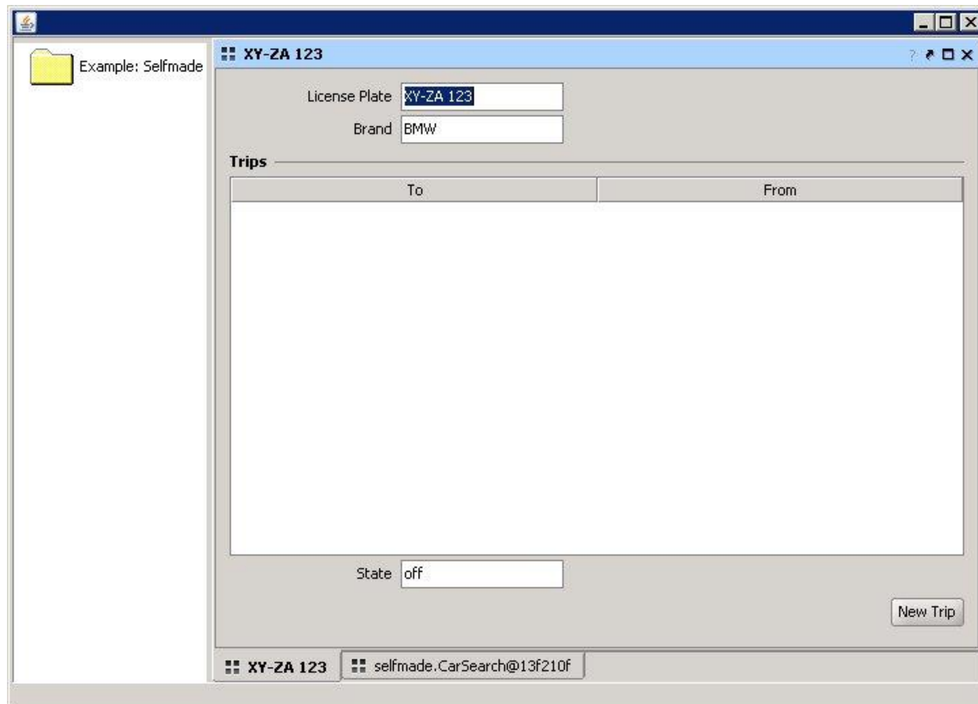
Starting the example:

```
java gengui.infonode.RootFrame selfmade.CarSearch
```

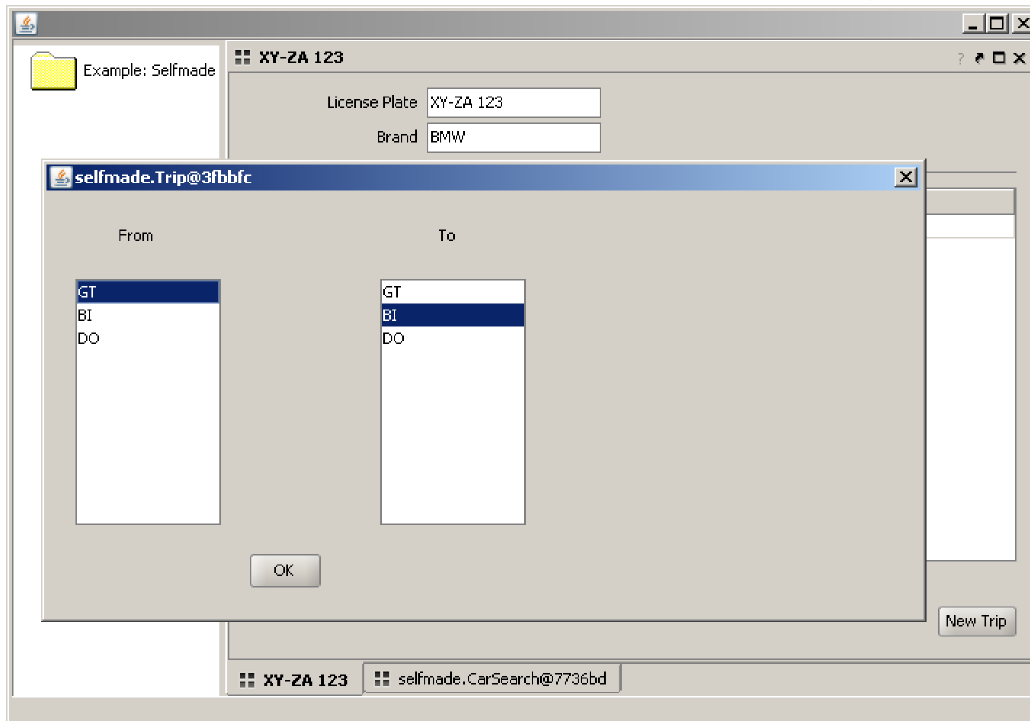
The hand-written user interface looks like this;



The generically displayed car appears as expected:



The following screenshot shows the hand-written modal dialog. Instead of input fields it provides a selection of predefined values for the trip's start and destination. The attentive reader may be surprised and say: „I could have achieved that in a completly generic way by providing choice methods“. That's absolutely right, but we'd rather like a hand-written modal dialog here ;-)



7.6 Combining generic and hand-written aspects

The framework provides some features to combine the advantages of generic and hand-written user interfaces. This is recommended when the structure of a generic user interface is generally suitable for a domain object but when there are only a very few extensions required for user interface control. E.g. auto-completion functionality for an input field may require key and mouse listeners while the generic coupling of user interface and domain object can be used as is. This way you will often find compromises which keep the development efford low even though the user interface is individually instrumented.

The simplest variant is to let the domain object implement the interface `ViewListener`. The interface declares only a single method which allows the framework to inform the domain object about the creation of a user interface for the object:

```
public Object viewCreated(Object domainObjekt, Object view);
```

When running in a Swing or AjaxSwing environment, the passed view is a JPanel array which may be instrumented by the domain object. The method may as well replace the whole view or decorate it. In this case, the method must return the result of that substitution / decoration in form of a new JPanel array. If there is nothing changed at all or in case of a pure instrumentation (like adding listeners) the method should return null or the view it got passed in as parameter. The components of a generic view have qualified names so that they are easy to find using the framework's `SwingUtil` class. If you perform instrumentation, you should keep in mind that the framework may cache the panels. The instrumentation code must therefore be written in a way that it removes already existing instrumentations if necessary.

If the domain class has a separate, generic viewtype class associated (see section 6.22), the ViewListener callback must of course be implemented in the view class. The ViewListener is only called for top-level visualizations of the domain object.

More instrumentation options are provided by the following function of the GUIServiceI interface which perfoms the generic binding between a domain object and its view (which is usually performed behind the scenes by the framework):

```
public void GUIServiceI.bind(Object domainObjekt, Object view)
```

When running a in a Swing environment, the passed view must be a JPanel that contains well-structured and named graohical components to display the domain object's properties and ist embedded objects. Creating such a panel from scratch is very tedious and error prone as you must follow all mapping rules which the framework applies for its own automatic creation process. The simplest way to get such a user interface is to initially let the framework create a layout file the generic way (see section 2.3). This layout file can later be loaded in the JFormDesigner to generate an appropriate Java class (see section 12.1 for optimal code generation). By means of the operation above, such a panel class can be bound to domain objects of the type the layout has originally been created for. What needs to be implemented is a Viewtype class for the domain class which does the following job in ist constructor:

- Instanciate the Java class generated from JFormDesigner
- Performing the required instrumentations for the graphical components of the panel
- Binding the panel object to the domain object (input parameter of the constructor) by GUIServiceI.bind
- Add the panel object to the (empty) panel of the view class

If the Viewtype doesn't need to contribute user interface components but provides only the invisible container for the panel object, the Viewtype can be derived from the base class gengui.util.Canvas. The method bind() is available with different signatures explained in detail in the Javadocs. The whole procedure above has the disadvantage that the panels cannot automatically be extended when the domain class is extended (see option `merge` for configuration parameter `jfd.retention.strategy`, section **Fehler! Verweisquelle konnte nicht gefunden werden.**). Hand-written user interfaces will always have this problem.

7.6.1 Views from layout files

If you want to safe the code generation with JFormDesigner, you may also read in the layout files and create panels at runtime. This can be achieved with the following operation in the GUIServiceI interface:

```
public Object GUIServiceI.createView(String domänenName)
```

The „domain name" ist he name of the layout file without extension and path. For domain classes being displayed by a single mask, this name is exactly the fully qualified domain class name. If a domain class is annotated with `@Viewtypes` and thus is displayed by multiple sub windows, each view type must specify the correct sub name according to the grouping of properties (see section 6.16.1). The advantage of this kind of view creation is that only the layout files needs manual maintenance and not the generated code which usually is more work. On the other hand the instrumentation may be harder as the graphical components are not available as members in the generated code. They must queried by traversing the component tree which is only poorly supported by Swing (e.g. `getComponents()` or `getComponent(int n)` etc). The gengui framework provides the utility class `SwingUtil` with some convenience functions for name-based component retrieval.

7.6.2 Example (examples-manual/customized)

The directory `examples-manual/customized` contains a variant of the Car application which makes use of the two variants above for the combination of generic and individual aspects. The CarSearch uses a generically bound view class which highlights buttons when the mouse is moved over the button:

```
public class CarSearchView extends Canvas implements MouseListener {

	private Color originalBackgroundColor;

	public CarSearchView(CarSearch search) {
		// Instantiate generated layout class
		CarSearchLayoutFromJFD layout = new CarSearchLayoutFromJFD();
		// Perform individual instrumentation
		layout.removeFromCars.addMouseListener(this);
		layout.Cars_newTrip.addMouseListener(this);
		// Put the panel in the view
		setContentPane(layout.customized_CarSearch);
```

```
        // Perform generic binding
        new GUIAdapter().bind(search, layout.customized_CarSearch);
    }

    // GUI code omitted
}
```

The class TripFactory gets a view class associated which closes the dialog for a new trip automatically without any further button click as soon as start and destination have been entered. The actual user interface is created by reading a layout file:

```
public class TripFactoryView extends Canvas implements FocusListener {
    private Trip trip;
    private JButton ok;

    public TripFactoryView(TripFactory factory, ModalOperation modal) {
        this.trip = factory.getT();
        GUIServiceI gui = new GUIAdapter();
        // Create panel from layout file
        JPanel panel = (JPanel)gui.createView(TripFactory.class.getName());
        // Put panel in the view
        setContentPane(panel);
        // Perform generic binding
        gui.bind(factory, panel, modal);

        // Perform individual instrumentation.
        // At this point, it is performed after generic binding,
        // so the special listeners are behind the listeners of the generic binding.
        // That's how we get data on focusLost of an already updated domain object.
        for (Component c: panel.getComponents()) {
            if (c instanceof JTextField)
                c.addFocusListener(this);
            if (c instanceof JPanel)
                ok = (JButton)((JPanel)c).getComponent(0);
        }
    }

    public void focusGained(FocusEvent event) {}

    public void focusLost(FocusEvent event) {
        if (trip.getFrom() != null && trip.getTo() != null)
            ok.doClick();
    }
}
```

8 Internationalization

Internationalization in general works the ordinary Java way by providing configuration files with the language code appended. Example:

```
gengui_de_DE.properties
gengui_en_US.properties
```

Depending on the configured locale the framework uses the corresponding configuration file. To use the internationalization, you may specify configuration parameter keys rather than the final text in **`@Prompt`** annotations. Alternatively you can simply define a prompt in a configuration file by adding a configuration parameter in the following format:

```
package.class.<propertyname>=...
```

Specifying the package and the class is optional, i.e. you may as well specify the property name alone. In this case, all fields for properties of this name will be displayed with the corresponding translation no matter which object they reside in. This is of interest for recurring domain names like “save”, “close”, “remove” etc.

By default, gengui generates the labels in form files as fix strings. I.e. if you keep the form files and change the locale later on, you will still see the labels for the locale which the forms were generated for. This can be changed by the configuration parameter **`jfd.localization`** . If the parameter is set to ‘global’ the JFDs will be created in a way that all labels refer to translation properties in gengui.properties resp. the locale-specific variants. The labels values can be edited with the resource editor of the JFormDesigner. Initial sets of properties can be added to the configuration files by the externalization feature of the JFormDesigner or by the framework’s built-in resource generator (see section 8.3). The references in the JFDs files follow the JFormDesigner’s naming conventions.

You can find an example for fully internationalized forms in examples-manual/i18n. The label translations are located in the files gengui_de_DE.properties and gengui_en.properties. Internationalized forms can be designed according to the following workflow:

1. Implement the classes
2. Provide a @Prompt for every visible method and property getter, specifiying a configuration key rather than the concrete text in a particular language
3. Provide the translation text for every configuration key in gengui.properties (default language) or gengui_<CT>.properties or gengui_<CT>_<LA>.properties for different languages. <CT> is an ISO countrycode and <LA> is a language code.
4. Create a JFormDesigner project including the folder of your gengui.property files as source folder. This will allow you to use JFormDesigner as a comfortable resource editor.
5. Set the framework’s localization mode to ‘global’.
6. Run the application to generate the JFD files. If you run into an exception saying something about missing resources, you may have a typo in your language configuration files or you just forgot a translation. In this case, open the JFD file in the JFormDesigner, click on Form→Localize…, and in the Dialog set the checkmark **Show only strings used in active form**. Look for missing properties and add them here. You may as well omit step 3 above, and provide all translations in the JFormDesigner.

When working in a full-featured DMD4000 project, steps 1 to 3 are automated by the D-GEN code generator and the localization mode should be ‘global’ by default. To avoid overriding customized labels, gengui provides the utility class **`gengui.util.propmerge.PropertyMerger`**. It allows merging the content of generated property files into existing customized property files while preserving all existing values and user comments.

8.1 Internationalization of the Java Bean Validation

Error messages produced by the Bean Validation are internationalized in the jar file of the Bean Validation implementation (hibernate-validator-4.0.1.GA.jar, org/hibernate/validator/ValidationMessages.properties). If the provided messages are not suitable for the project, a corresponsing ValidationMessages.properties file with customized messages must be placed somewhere on the CLASSPATH (see also [HV4]). Alternatively you may of course modify the file included in the library.

There are a few general messages included in gengui.properties because they refer to error situations detected by the gengui framework rather than the underlaying Bean Validation (e.g. `validation.selection`).

8.2 Programmatic localization

If localized strings must be assembled in the programm code, you may use the class gengui.util.I18nPropertyBasedImpl resp. the interface gengui.util.I18n. It provides programmatic access to the translation functions used internaly by the framework. Basically this class is built around Java's concept of property-based resource bundles. E.g. the files gengui.properties, gengui_de_DE.properties, gengui_en.properties and so on simply make up a resource bundle with the base name "gengui". The framework's I18n class just adds a hierarchical search order as this is explained for other configuration parameters in chapter 11. It also caches resource bundles for faster access at runtime. The localization example in directory `examples-manual/i18n` also demonstrates the programmatic way of localized string assembly.

8.3 Automatic label externalization

If you are not using tools like D-GEN to generate localization files, you may simplify the externalization of labels by a built-in generator. The configuration parameter **`jfd.localization.retention.strategy`** allows to specify if the framework should create externalized labels along with the localized JFDs and add them to existing resource bundle files. See the file gengui.properties for the supported generator modes. If the configuration parameter is not set but localization is switched on, the framework derives a reasonable value from the **`jfd.retention.strategy`** configuration. E.g. if domain class extensions are supposed to be automatically merged into existing JFDs (retention strategy 'merge') it makes sense to also automatically merge appropriate new label resources to existing resource files.

The labels will be added to the resource file being suitable for the locale the application is configured to. If the application's locale is set to "de_DE", the label properties will appear in `gengui_de_DE.properties` if present, otherwise in `gengui_de.properties`. If the current locale is the default locale, the framework will even consider `gengui.properties` as a valid place as long as the file is writable. So if you want to protect `gengui.properties` from being extended by label properties, keep the file read-only or make it available through a JAR file rather then a directory.

The automatic externalization is of course limited to those labels which are directly derived from the class structure. If you extend the JFDs manually e.g. by explanatory text or headlines, the localization of these must be appended and edited with the JFormDesigner's resource editor. However, if you may loose resources in your resource files, which are required by the JFDs, they will also be added automatically the next time you instanciate these JFDs in case the retention mode is set to 'merge' or 'overwrite'. In these cases the resource values are derived from the resource names and probably need some fine tuning.

Automatic externalization is demonstrated with the example in directory `examples-manual/i18n/generateresources`. It includes the same classes as the parent directory `examples-manual/i18n` but with hardly any @Prompt annotations and without JFDs and appropriate label resources in the delivered property files. When you run the example, the framework will create the JFD files and add the label properties to gengui_en.properties as the application is explicitly set to the locale en_US. The method startTheEngine() is annotated with @Prompt("general.method.start") refering to a resource identifier which is already present in the resource file. As the localization retention for this example is set to 'merge', this resource value is not changed but used as is.

9 Simulation Mode for Fast Prototyping

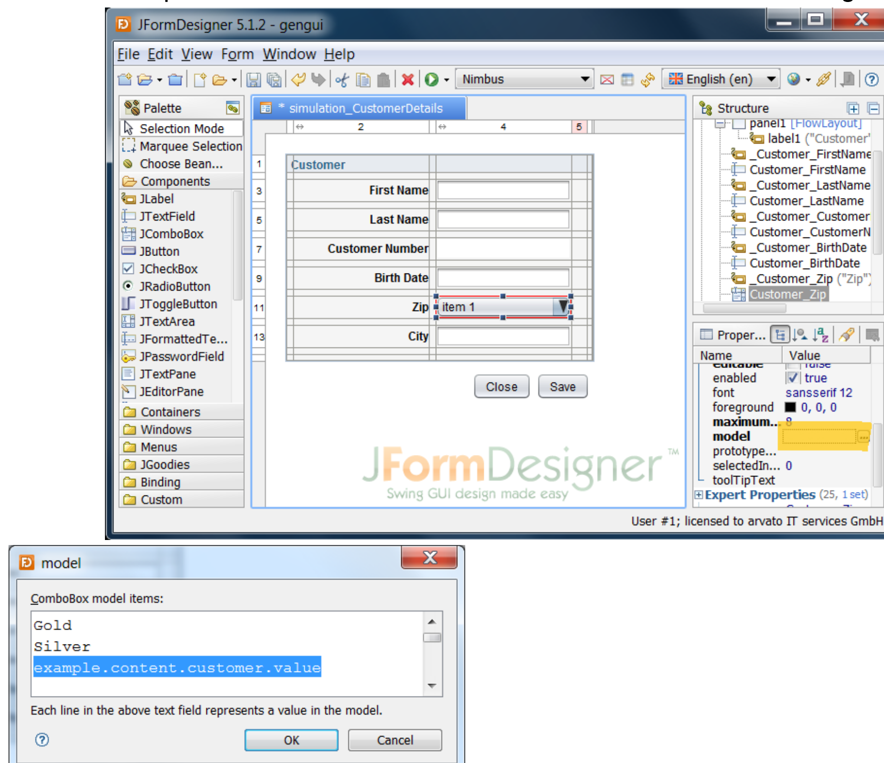
The generic approach offers an interesting opportunity for fast prototyping. The configuration parameter **`execution.mode`** allows to run an application in a simulated way. “Simulation” means that the user may click through the application whithout any domain object being instanciated at all. Based on the unique 1:1 relationship between domain classes and their graphical representations, the framework can usually derive the mask composition and mask flow from class structures and method signatures. In simulation mode it does so, giving the users a first impression of the later application based on minimalistic implementation work. The domain classes just have to provide correctly typed properties and methods without really implementing any logic. E.g. all methods may return null – only the *existence* and the *return types* of the methods are of interest for simulated execution. The JFD files may be populated with static example content for inputs fields to keep from showing only completely empty masks in simulation mode. The example content is ignored when the JFDs are used in live mode. The JFormDesigner also allows to localize example input except for tables, lists and combo boxes the content of which is based on models. See section below for localization of these components’ content.

9.1 Simulation instances

9.2 Localization of model-based example content

9.2.1 JComboBox

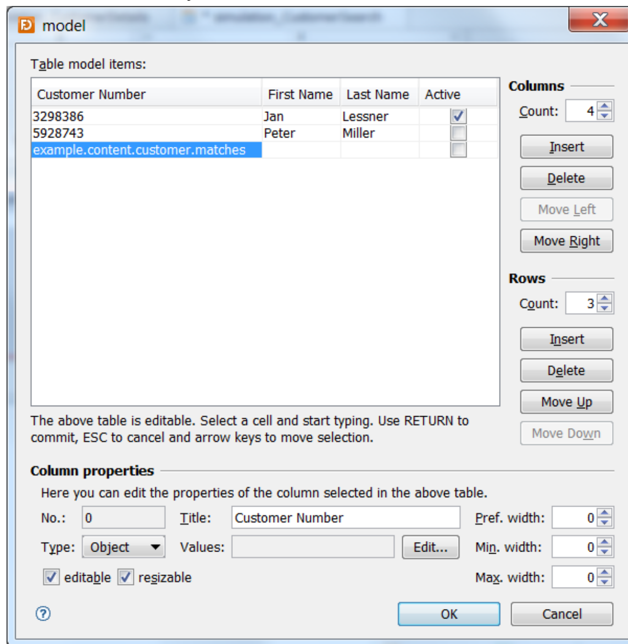
Select the combo box and add a model. As the last entry of the model values add the resource key which to read the actual values from at simulation run time. There may be more values listed just for immediate layout control while beautifying the mask. They will all disappear and be replaced at run time if the last value can be translated through the mask’s resource bundle.



The combo box values must be provided as comma-separated list in the resource file, with the first value being the one which is initially displayed in the mask, e.g.
example.content.customer.value=Unkown, Gold, Silver

9.2.2 JTable

Select the table and add a model. As the last entry of the model values add a line with only single cell filled with the resource key which to read the actual values from at simulation run time. There may exist more lines with plain content just for immediate layout control while beautifying the mask. They will all disappear and be replaced at run time if the last value can be translated through the mask's resource bundle. It does not matter which cell of the last line contains the key, so there is no conflict with boolean-typed rows which don't allow text input.



The two-dimensional content of the table may be provided

9.2.3 JList

Not yet supported

9.3 Example (examples-manual/simulation)

The directory `examples-manual/simulation` contains a variant of the Car application which makes use of the simulation mode. The classes have no logic implemented and therefore are only suitable for simulation mode. Clicking throw the application shows the example content which was directly added in the JFD files.

The class simulation.Trip is annotated with @SimulationInstance referring to a static Trip object. It is programatically initialized with "Hamburg" as starting point of the trip. That the simulation mode displays an object rather than a simulation can also be recognized by the input validation being performed when pressing the OK button.

10 Integration with a backend

Beside the user interface, an application usually also requires a backend. The backend technology should be hidden from the domain classes as far as possible. It should not matter if the backend simply consists of pure Java classes residing in the same JVM or if the backend is an AJB layer on a cluster of JEE applications servers. The developer should have the chance to replace the backend implementation by mock implementations to simplify unit testing.

These requirements are not GUI-specific and are adressed by the framework “spine” which is responsible for dependency injection and declarative transaction management. The user manual for spine was moved from this chapter to a separate document which can be found in Subversion under:

http://deadsdco1:8088/svn/DMD-Plattformstrategie_2008/trunk/GenericTransactionUnit/doc/Anwenderdokumentation GTU und Spine.doc

11 Configuration by gengui.properties

11.1 Configuration parameters

The gengui framework provides numerous configuration parameters which are defined by the configuration file `gengui.properties`. It is recommended to have a look on the configuration file included in the distribution and read through the file's documentation for an overview. The following list shows a few central parameters as an example.

Parameter	**Beschreibung**
jfd.retention.strategy	Parameter for the handling of layout files. E.g. what's happening with the layout if a domain class was extended (new property, new method) – will the layout be discarded, kept as is, extended? See section 6.18.1
jfd.file.path	Specifies the path for layout files. Alternatively you may use jfd.resource.path for loading layout files from the classpath which is helpful e.g. for Webstart applications. Using the classpath however prohibits the creation of layout files and is therefore only suitable for productive environment.
fieldorder.policy	How is the framework supposed to react if a property is missing in a FieldOrder annotation? Error message or ignore it?
global.exceptionhandler.class	Fully qualified name of an excption handler class. The class must implement the interface **`gengui.util.SevereExceptionHandler`**.
status.staytime	This parameter specifes how long a status message is displayed before the framework automatically clears the status bar.
window.width window.height	Initial width and height of the root window. If one of these configuration properties is missing, the root frame is maximized.
synchronization.eager.auto	Allows to specify if the content of fields with eager synchronization should by auto-synchronized when the user pauses for a moment during typing.

11.2 Specialization of configuration parameters

As it is not always adequate to apply configuration parameters globally to the whole application, the gengui framework allows to override the global configuration by package- or class-specific ones.

Overriding is achieved by providing specialized configuration files according to the following naming convention:

Configuration per class

`<fully_qualified_classname>_gengui.properties`

Configuration per package

`<fully_qualified_packagename>_gengui.properties`

Common configuration

`gengui.properties`

As it is indicated above, the dots of fully qualified names must replaced by an underbar. This is due to tha fact that the dot has a special meaning when reading property files as resource boundles. The configuration parameters are retrieved by traversing the configuration files in the following order, stopping at the first occurrence (i.e. the most specific configuration rules):

1. Locale-specific class configuration
2. Class configuration
3. Locale-specific package configuration
4. Package configuration
5. Locale-specific common configuration
6. Common configuration

7. Default values of the framework

12 Appendix

12.1 Exchanging the window manager

All examples above are based on calling the class `gengui.infonode.RootFrame`, which provides a predefined main window to place the generic user interfaces of the domain objects in. This class may completely be replaced by any class implementing the interface `gengui.WindowManagerSPI`. E.g. all dependencies to the commercial toolkit *Infonode Docking Windows* [IDW] are encapsulated in the package gengui.infonode and can be removed by exchanging the window manager. You may provide different window managers e.g. to integrate generic GUIs in existing applications or you may provide simplified visualizations for limited runtime environments.

Details for the implementation of an alternative window manager can be found in the comprehensive Javadocs of the interface `gengui.WindowManagerSPI`. For visualization of generic GUIs in a web frontend, the framework provides the class `gengui.ajaxswing.RootFrame` which is based on the toolkits AjaxSwing. AjaxSwing must be available at least in version 3 to run with gengui. The recommended version is 3.2.5.

12.1.1 Scalable generic web applications with AjaxSwing

AjaxSwing provides a special class loader which loads an application's classes in a session-specific way. This allows to safely store session-specific information in static members even if multiple sessions are running in a single JVM (see according concepts in the AjaxSwing documentation). The application code doesn't have to care for concurrent access on static variables, so that any kind of Swing application should immediately work on the web as well. Using AjaxSwing this way is therefore very comfortable but usually leads to poor scalability as many initializations and memory allocations are performed per session rather than per JVM.

The gengui framework itself is designed in a way that its libraries may be shared by multiple sessions. It therefore allows to design reasonably scalable applications as far as this possible at all with the AjaxSwing approach. The following points make up a little check list for the implementation of such applications:

1. To load classes only once per JVM and not separately per session, the classes must be available through the AjaxSwing configuration parameter `common.Classpath` rather than `agent.Classpath`. Alternatively AjaxSwing also allows moving libaries to <AjaxSwing installation>/wcapps/lib resp. indivual class files to <AjaxSwing installation>/wcapps/lib/classes. However this bears the risk that AjaxSwing's internally assembled CLASSPATH environment variable becomes too long. Environment variable are strongly limited on Windows.
2. As gengui and many other frameworks call application code via Java reflection, you must usually provide all classes and libraries of the application by `common.Classpath`. As a consequence, all non-final static members of the application and its used libraries must be checked for their multi-session applicability. The static members of interest can be queried by the framework's utility class `gengui.util.staticmember StaticMemberFinder`.
3. Static members fall into two different groups
   - The majority usually contains session-independent information (e.g. caches) und should therefore be shared between sessions. For these members you must ensure thread-safe access.
   - Some members may contain session-specific information, e.g. the current user name or dialog flow markers. This happens especially by bringing former fat client Swing applications to the web, because static members are absolutely reasonable for storing session information and are straight-forward for fat clients. For this kind of information the static members must be replaced by usage of session storage. The framework provides such a storage for key-value pairs by means of the static methods `put(String key, Object value)` and `Object get(String key)` of class `WindowOperation`. Unfortunately scalability doesn't come without code change at this point. In general it is recommended to analyze first, of the corresponding members are actually required at all. Static members are a kind of global variables which should be a rare exception in object-oriented programming with Java.
4. When all the relevant static members have been identified and examined, it is recommended to add the JUnittest `gengui.util.staticmember.StaticMemberJUnitTest` to the application's test suite. This test keeps track of the appearance of new and potentially dangerous static members in the project code based on a reference file. It forces the development team to

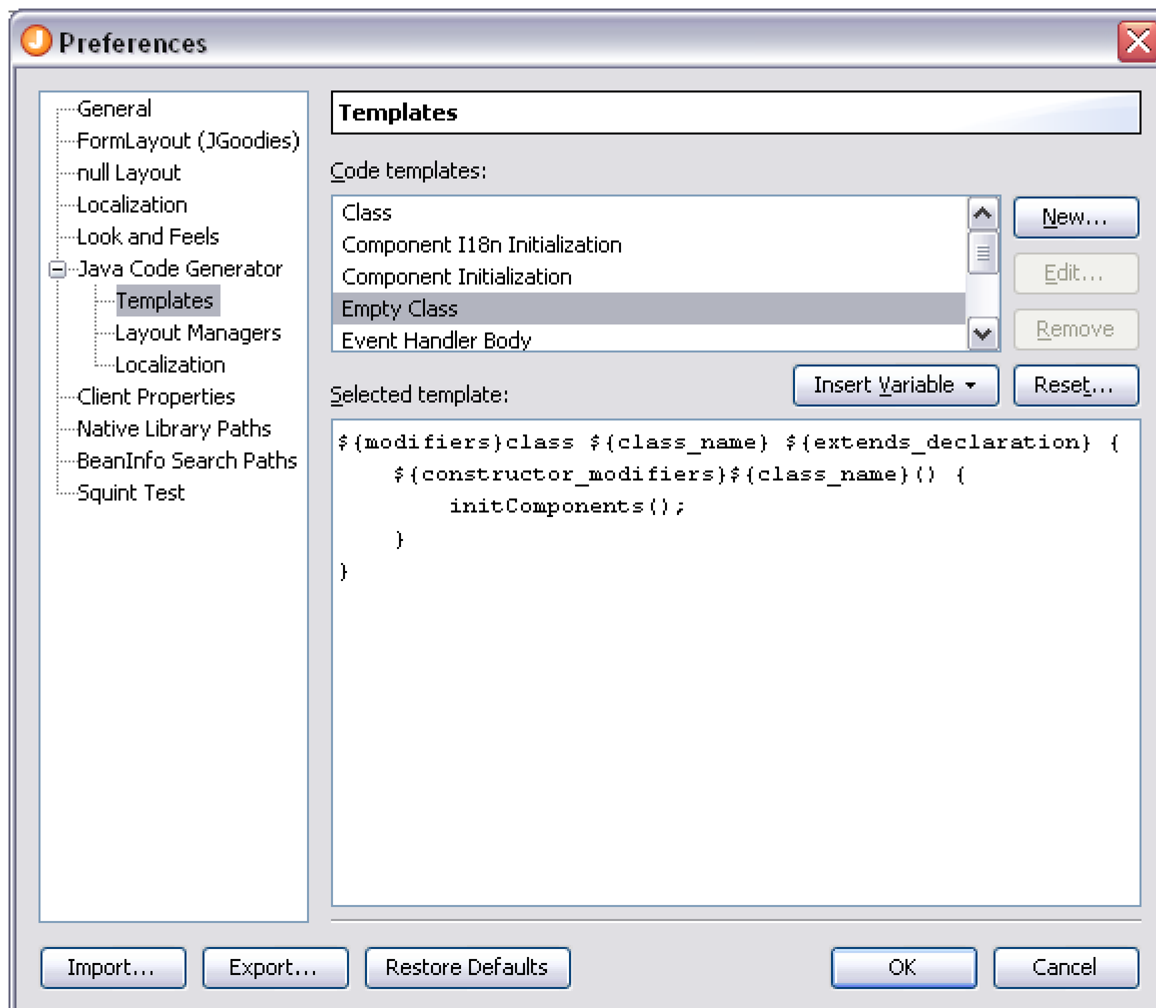
check these new members as described in the step before and either remove the member or add it to the reference file.

12.2 Customizing code generator templates in JFormDesigner

If you are using JFormDesigner to generate Java code (not required for pure layout file finishing!), the tool should be configured in a way that the generated source code doesn't need any manual changes when it is integrated with gengui. This is best achieved by initializing the components in the constructor and declaring the member variables public.

As the generated code by default doesn't contain a constructor, you can copy the **constructor template** code from the **Class** template (used for nested classes) into the template **Empty Class** (used for top-level classes).

The following screenshot shows the adapted template:



To generate member variables as public, click on „(form)“ in the structure panel, select the tabulator **Code Generation** and change the property **Default Variable Modifiers** from „private” to “public”. The following screenshot illustrates that: