bbv behind things.

Report

Dynamic Properties

Version	**:**	**V 1.0**
Date	**:**	**February 27, 2008**
Classification	**:**	**Internal**

bbv Software Services AG, Blumenrain 10, Postfach, CH-6002 Luzern, Tel.: +41 41 429 01 11, Fax: +41 41 429 01 13, info@bbv.ch, www.bbv.ch

Document name: **DynamicPropertyReport.doc**

Document number: **0052**

Document type: **Template**

Authors

Name	Chapters, Sections	Initiale Date	Visa
Ueli Kurmann	creation	4.6.2007	

History

Version	Initials	Date	Description	Section
1.0	UK	4.6.2007	Initial creation	all

Table of contents

1 Introduction

1.1 Purpose

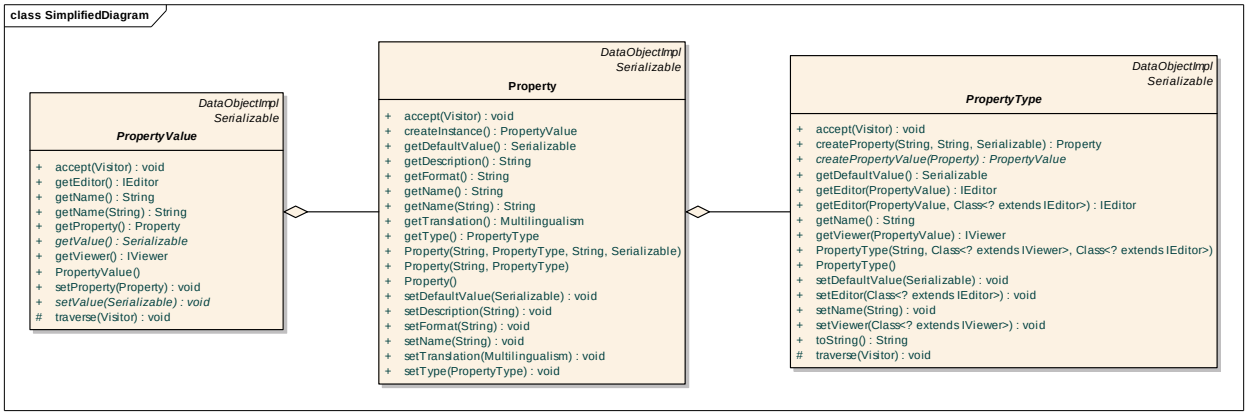
Applications need to be extended dynamically with additional properties without the adaptation of the underlying data model. Dynamic Properties enables the definition of simple, complex and indexed properties. Furthermore it is integrated into pmMDA.

1.2 Scope

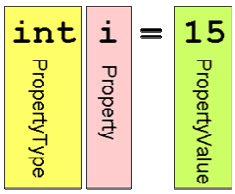
This document describes the architecture, design and the usage of the Dynamic Property framework.

2 Architecture

A property has a type and a value as shown in the class diagram below.

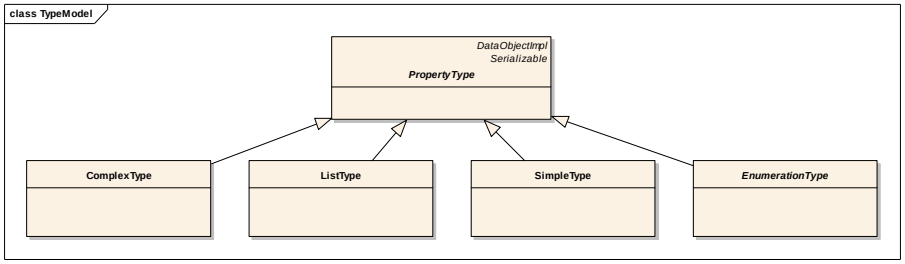


This model can be compared with the well known variable declaration. For example in Java the declaration of an integer value has the form ***`int i = 0`***. The following sketch shows the mapping of this declaration to the data model.



2.1 Property Types

We distinguish the following four property types as illustrated in the class diagram. The property type contains a method to instantiate corresponding properties of this type.

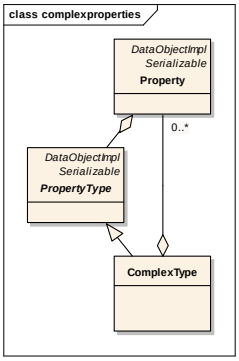


2.1.1 Simple Type

Simple types are the systems built in types as Boolean, Date, Double, Integer, String. They are mapped to the corresponding system type and are the base type system of the Dynamic Property framework. All simple types have to be serializable.

2.1.2 Complex Type

A complex type consists of arbitrary many properties. Complex types are analogue to classes and structs in programming languages. Inheritance is not supported.



At a first glance it may be surprisingly that complex types containing properties and not types. But this is the mapping also used in programming languages like Java and it is evident that members of complex types have to be named which is not possible using types only.

2.1.3 List Type

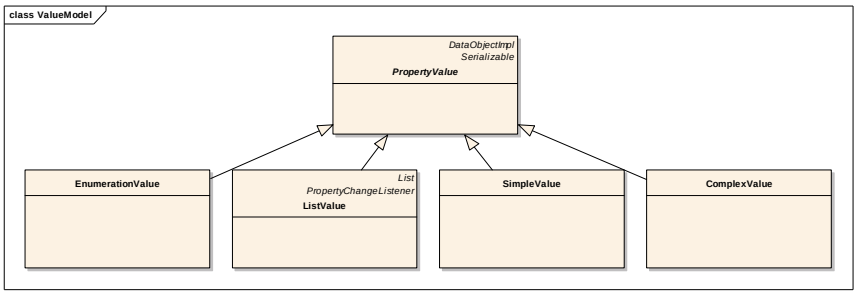
The list type is a typed indexed property. The elements of the list can be of any type.

2.1.4 Enumeration Type

This type is used to map reference codes. Both, flat and hierarchical enumerations are supported.

2.2 Property Values

Accordingly to the four different types there are also four property value implementations.



Property values should not be instantiated manually. Instances should be created on behalf of a property instance which contains a factory for this purpose. Furthermore property values may have default values which are defined by the property.

2.2.1 Simple Value

Simple types are serializable and therefore stored internally in a java.io.Serializable member variable.

2.2.2 Complex Value

Complex values are compounds of property values. Internally the values are stored in a map.

2.2.3 List Value

The list value has an anonymous property which handles the type of the list. The list values itself are stored in a collection of property values. The type list value implements the java.util.List interface to ensure standard access to particular elements.

2.2.4 Enumeration Value

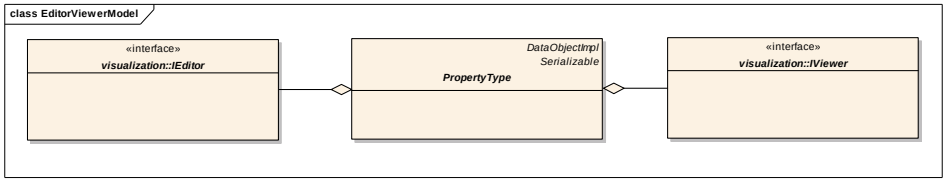
The value of an enumeration is represented by an integer whereas each enumeration has a unique key.

2.3 Multilingualism

Property names and enumeration elements are language dependent. These values are translated with the multilingualism object. The name can be requested with the getter method and the corresponding language key. If the translation does not exist the default name is returned.

2.4 Editor and Viewer

To allow the visualization of property values an editor and viewer has to be assigned to property types. The editor is used to manipulate the value whereas the viewer provides a read-only presentation of the value. Special editors and viewers are provided for complex and list types. They recursively refer to other editors and viewers. Editors and Viewers have to implement the IEditor or IViewer interface and return a swing JComponent.



3 Persistence / Hibernate

All core classes can be persistent and implementing the DataObject interface. Therefore standard pmMDA persistence mechanism can be used. The Hibernate mapping file was created manually and is located in the source repository.

4 pmMDA Integration

The Dynamic Property Framework is integrated seamlessly into pmMDA. In the context of packages and classes the tagged value “extendable” can be set. The DOG generator creates a list of property values (propertyValues) for DataObjects which are “extendable”.

A singleton class manages the assignment of properties to DataObjects. For each dynamic property an instance of the corresponding property value is created. The properties are accessed with getter and setter methods which take the corresponding property name as parameter.

5 DataObject Editor

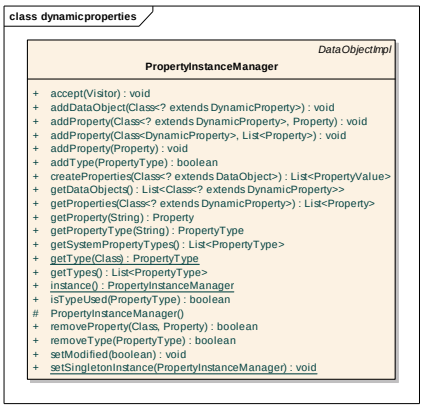
The DataObject Editor is a simple tool to manage the dynamic properties of DataObjects. It has basically two functions. On the one hand property types can be managed; on the other hand properties can be assigned to DataObjects. Types can be created, edited and deleted. Furthermore these types can be assigned to DataObjects. The editor has to be integrated manually into a pmMDA application.

6 User Guide

This section shows the usage of the Dynamic Property framework.

6.1 PropertyInstanceManager

The PropertyInstanceManager is a helper class which manages the available types and properties. Instantiated types and properties can be registered to be reused and the systems built in types are pre-registered. In the following sections the variable manager is a reference to PropertyInstanceManager.instance().



6.2 Create Property Types

The following code snippets illustrate instantiation of types.

```
/** Simple Type **/
SimpleType tInt = new SimpleType("int", Integer.class, StringViewer.class, IntegerEditor.class);

/** List Type **/
ListType tList = new ListType("intList", tInt, ListViewer.class, TableEditor.class);

/** Complex Type – name and birthday are properties registered in the property instance manager **/
ComplexType tPerson = new ComplexType("Person", ComplexViewer.class, ComplexEditor.class);
tPerson.addProperty(manager.getProperty("name"));
tPerson.addProperty(manager.getProperty("birthday"));
```

6.3 Create and Use Properties

The following code snippets illustrate the instantiation of simple properties.

```
/** Create Property **/
Property pName = new Property("name", PropertyInstanceManager.getType(String.class));

/** Create a Simple Property Value **/
SimpleValue pNameValue = (SimpleValue)pName.createInstance();

/** Set the Simple Property Value **/
pNameValue.setValue("Gosling");

/** Get the Simple Property Value **/
pNameValue.getValue();
```

The following code snippet shows the use of list properties.

```
/** Create Property **/
ListType intList = new ListType("intList", manager.getType(Integer.class));
Property numbersProperty = intList.createProperty("numbers", "", 0);

/** Create a List Property Value **/
ListValue numbers = (ListValue)numbersProperty.createInstance();

/** Add Elements to the List **/
numbers.add(1);
numbers.add(2);

/** Iterate over the Values **/
for(PropertyValue value:numbers){
  System.out.println(value.getValue());
```

```
}
```

The following code snippet shows the use of complex properties.

```
/** Create Complex Type **/
ComplexType complexType = new ComplexType("contact");
complexType.addProperty(new Property("name", manager.getType(String.class)));
complexType.addProperty(new Property("birthdate", manager.getType(Date.class)));

/** Create Complex Property **/
Property contactProperty = complexType.createProperty("myContact", "", null);

/** Create a List Property Value **/
ComplexValue myContact = (ComplexValue)contactProperty.createInstance();

/** Set Elements **/
myContact.setValue("name", "Gosling");
myContact.setValue("birthdate", new Date("19/05/55"));

/** Get Values **/
myContact.getPropertyValue("name");
myContact.getPropertyValue("birthdate");
```

6.4 DataObject Editor

```
/** Shows the DataObject Editor **/
new DynamicPropertyView(manager);

/** afterwards the manager has to be made persistent **/
```

7 Package Overview

ch.bbv.dynamicproperties	Manager classes.
ch.bbv.dynamicproperties.core	The DataObjects of the framework.
ch.bbv.dynamicproperties.objecteditor	Implementation of the DataObject editor.
ch.bbv.dynamicproperties.visualization	Interfaces and implementation of editors and viewers.
ch.bbv.dynamicproperties.pmmda	Helper classes used by the pmMDA integration.

8 UML Diagram