The JANUS Application Development Environment—Generating More than the User Interface

Helmut Balzert, Frank Hofmann, Volker Kruschinski, and Christoph Niemann

Lehrstuhl für Software-Technik, Ruhr-Universität Bochum, Universitätstraße, 150, D-44780 Bochum, Germany
Phone: +49-(0)234-700-{6880, 6791, 5918, 7982}
Fax: +49-(0)234-700-6914
E-mail: {hb, hofmann, krusch, niemann }@swt.ruhr-uni-bochum.de, janus@swt.ruhr-uni-bochum.de
WWW:http://www.swt.ruhr-uni-bochum.de/forschung/veroeffentlichungen.html

Abstract

The increasing pressures of competition demand greater productivity and quality in the development of software. These goals are attainable by generating as much as possible and programming as little as necessary. Beginning with an OOA modeling of the problem domain component, this article will show how the user interface as well as the linkage to data keeping can be generated through an integrated approach. In addition, a client/server configuration is also possible. A OOA model upon which two generator systems are installed is the basis for generating.

Keywords

User interface generation, OOA model, object oriented database, rapid prototyping, application framework.

Introduction

The ever increasing demands on the productivity and quality of software development necessitates extensive automated support for application development. If one examines object oriented application development (figure 1), the way from the problem domain to an object oriented analysis model (OOA model) cannot be automated. This step shall continue to belong to one of the most ambitious tasks of software development.

If an OOA model is created, it forms the basis for any additional steps of development. The concepts available today describing an OOA model (class, inheritance, association, aggregation, object life cycle, interaction diagrams, subsystems, see also [Coad91a, Booch94, Rumbaugh91]) allow close to real-world situation modeling of the problem domain.

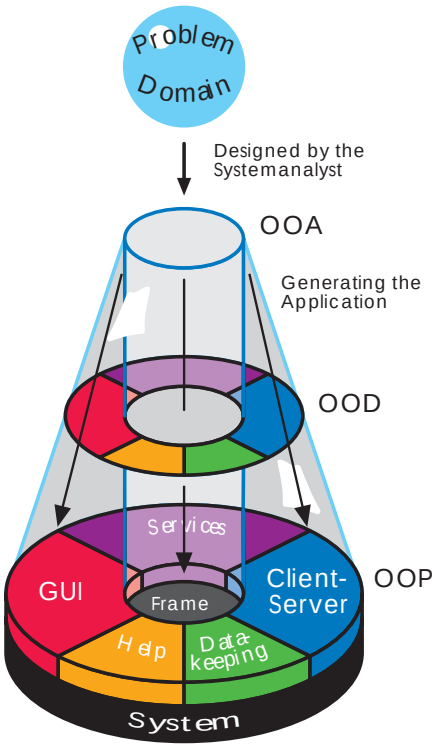

Figure 1. The way to an application starting at the problem domain

The following must be done to obtain a usable application from a OOA model:

- Integration into the system software of the target system.
- Design and connection of the user interface to the problem domain components.
- Connection to the desired data base management system (DBMS).
- Design and connection of the help system.
- Creation and connection of various services (e.g., multiple user administration, client administration, etc.).

Analyzing the jobs to be completed, one ascertains that a large part of these tasks can be automated by generators. The term "automated" is intentionally used instead of "automatic". Automated is intended to express that generating does not run fully automatically, but rather that the developer retains the possibilities to intervene and make decisions during the generating process.

Therefore, the optimal goal consists of generating nearly all additional necessary tasks from an OOA model. The semantics of a problem domain component are principally incapable of being generated, i.e., the technical semantics have to be implemented by the software developer. He uses the desired programming language (inner column of figure 1).

Even in this area, however, much can be generated. Today's OOA/OOD tools allow the corresponding program frameworks to be generated from the OOA model, e.g., the tools Together/C++, Paradigm Plus and ObjectiF. To have a practical benefit of generating system components the developer needs an integrated system which will combine all fragments.

Furthermore, it is not enough to generate all components from the same starting point (e.g., an OOA-model) an integration of all generated parts can be done automated. Therefore we have developed the JANUS Application Development Framework (JADE[1]). It is a further development of the JANUS-system [Balzert93, Balzert-94, Balzert95a, Balzert95b]. The JANUS-system was capable of generating and animating a graphical user interface from an OOA model using the capabilities of an UIMS.

The advanced system now produces the user interface, the code frame for the application domain, the database schema, further services (e.g., a help system, printing facility) and 'last but not least' the connection between all these parts. The starting point is still an OOA-Model. JANUS requires the model in a well defined input language, the JDL (JANUS Definition Languages) which is an extension of ODL and IDL. To avoid that the user has to code his OOA model using this language directly, we have built interfaces to some popular OO CASE tools. Currently JDL can be exported by the case tools Paradigm Plus and Together C++.

The result of the generation process is a ready-to-work-with application offering basic functionality. The user is able to create and modify objects of classes defined in OOA by using entry forms. If a corresponding relationship (association or aggregation) exists in the OOA model the user can establish links between objects, too. Additionally a list view of all objects that have been created for each class is provided.

Functionality for sorting and deleting objects is also generated. All data entries are kept persistent in an underlying database. Until now the software developer has not written a single line of code. The only work that has been done was defining an exact OOA model of the application's problem domain.

The generated program will however be the fundamental frame of a final system. A programmer will have to complete the application. He has to implement the operations defined in the OOA model to provide the application's core functionality. Additional features—especially regarding the GUI—can be added to the generated code. To ease this JANUS generates C++ source code for all parts of the program. These can be edited and compiled the normal way. This paper describes the concepts of integrating all parts. It gives examples of the transformation process and its results.

---

[1] This JADE system has nothing in common with JADE [VanderZanden90] but the name. It seems that we have no luck in choosing the right name for our system.

1 As to the Situation

The situation today is characterized by increasing attempts to automate separate areas of the software development process. Class libraries in combination with a graphical editor are used today in the development of GUIs. GUI class libraries are hierarchically organized and provide predefined interface objects at higher abstraction levels. The activation of the underlying window system is undertaken by internal operations and remains hidden from the developer. The design of the GUI using this technique leads to two results:

- A code frame will be generated in the desired programming language (usually C++). The combined interface objects can be created dynamically using this code. The I/O operations of these objects have to be manually linked to the OOA model.
- Characteristics of interface objects such as position, size, labeling, and shape will be placed in resource files. Each resource object contains an identification through which the connection to the objects implemented in the programming language is made. A special resource translator transforms the resources into object code, which will later be linked to the application.

It was shown under the JANUS system [Balzert93, Balzert94, Balzert95a, Balzert95b] that a GUI can be generated and subsequently animated from an OOA model based upon expert knowledge of software ergonomics.

However, the linkage to data keeping in particular is missing in order to attain a usable application. When using an object oriented database (OODB), the object model is defined in an Object Definition Language (ODL). The developer separates the declaration (data and interfaces) of an application from the implementation. A declaration preprocessor for the ODL takes over the following tasks:

- The ODL is transformed into a declaration conforming to a programming language which then can be translated by a compiler together with the implementation of the application.
- A database with the database schema obtained from the ODL declaration is created in which the object model of the application is also established as a meta schema.

The implementation of the technical semantics of the OOA model occurs in the selected programming language. To handle persistent objects, an Object Manipulation Language (OML) is provided by an external library. This library comes with the chosen database management system. With this, the programmer can manipulate persistent objects with the same concepts (pointer, list,...) known from the programming language as usual.

The declarations transformed in the programming language and the implementation are translated by the compiler into object code. The runtime system ODBMS is added to the object code during linkage so that the finished application can ac-

cess the predefined database. To allow the generation of persistent classes, all persistent classes have to be marked in the OOA model. All other information is already present to generatively couple an object oriented database.

A corresponding coupling to a relational database is similarly possible by utilizing the respective class libraries, e.g., DBtools.h++. Appropriate transformation rules are describe (e.g., in [Blaha94]).

A three-layered-architecture comprising a GUI layer, the actual application layer, and data keeping layer arises as the software architecture. The application layer not only contains the implementation of the technical semantics but also contributes the connection between the GUI layer and data keeping. In particular, the overall goal is to encapsulate the layers as closely to one another as possible in order to make an appropriate client-/server distribution feasible.

It has now been shown that a partial generation does not appropriately take the global aspects of the application environment into consideration. If, for example, solely the interface is generated without taking the coupling of data into consideration, it will lead to problems in subsequent application development. The communication between the GUI level, application level, and data keeping level has to be manually established. This requires detailed knowledge of the code at all levels and is costly. An integrated overall plan is therefore necessary.

2 An Integrated, Technical, and Comprehensive Plan

As mentioned above a OOA model is the basis for the generation process. In the moment the JANUS generator system uses only information given by a class diagram representing the application's object model of the problem domain. The elements of these class diagram (classes, attributes, operations, relations, etc.) have to be specified in detail.

To provide this information to the generator system, the OOA meta model in figure 2 has been developed. The OOA model of a specific application is a single instance of the OOA meta model. The meta model can be instantiated by a special file using the JDL grammar. JDL input files describe an OOA model CASE tool independent and implementation language independent.

Not only problem domain and database specific characteristics but also GUI relevant properties are represented by the OOA meta model. It is important to mention that all GUI relevant properties have default values which work very well in most cases. But the OOA analyst (or a consulted GUI specialist) should have the possibility to override these values to customize the generated GUI whenever needful.

The meta model was expanded with characteristics of the object models of the OMG [OMG91] and ODMG [Loomis93]. As before, the most important common concepts of the object oriented methods according to Rumbaugh [Rumbaugh91], Coad/Yourdon [Coad91a] and Booch [Booch94] are found in this model.

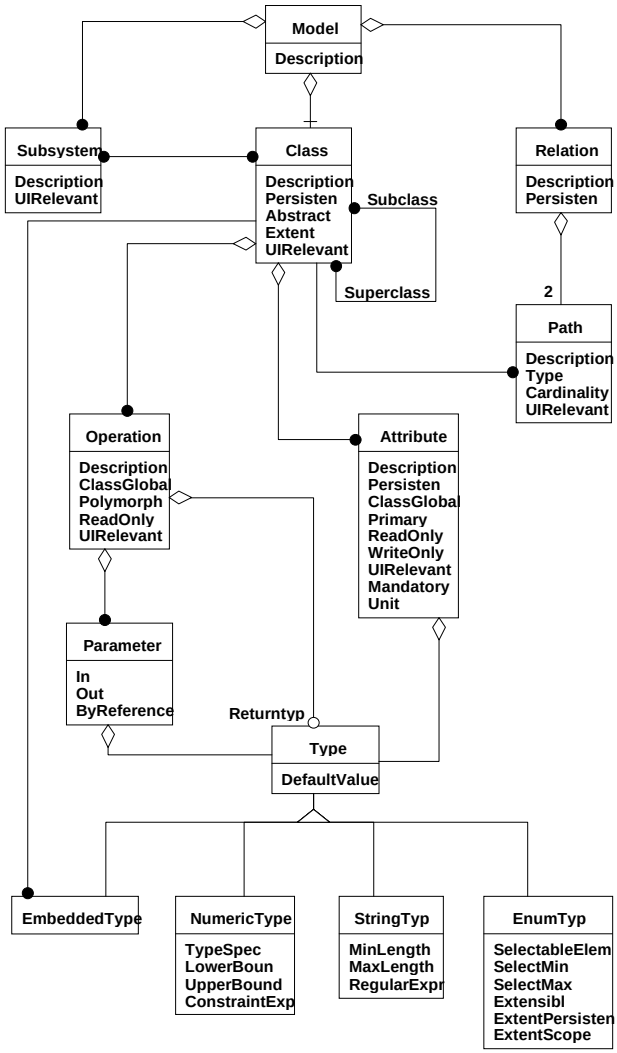

Figure 2. Meta model as the generator's base

The information contained in the meta model is utilized by a GUI- and an App-generator (application generator). The meta model's operations (not included in figure 2) take over the control and coordination of these generator systems.

The App-generator creates the code frames for the OOA model as well as the connection to the database and, if requested, additionally the client-server distribution. In addition to the code frames, the operations for read and write access to all attributes are also generated. On the one hand, for each attribute, one operation is created for read and write access to the attribute value. On the other hand, each class contains one operation which assists in reading and/or writing each attribute belonging to this class irrespective of its type. If the systems analyst has set restric-

tions (e.g. ranges or dependencies between attributes), they will be checked for the applicable attribute before write access.

The GUI generator similarly rests upon the meta model. The previously mentioned JANUS system [Balzert93, Balzert94, Balzert95a, Balzert95b] was expanded and modified for this. Only the generation of static layout and some aspects of dialog dynamics have been supported up until now. This information is placed in a resource file.

It is additionally necessary to create GUI code, which accesses the resources and connects the individual interface objects with the objects and attributes of the OOA model. For each dialogue identified in the OOA model a class is generated in order to automatically couple the created interface with the OOA model. By coupling with the App-generator, the operations supplied there become available for access to the attribute values irrespective of type. Thus, it only then becomes possible to generate an interface closely coupled to the OOA model with justifiable expense, and therefore, to obtain a complete, detailed application.

In the next section we will describe a simple example that shows the input and the output of a generated application and explains some important actions that have to be taken to get a running application

3 A Simple Example

Figure 3 depicts the OOA model of a simple sample application. The OOA model had to be entered into a CASE tool manually. Generally a complete OOA model consists of at least the graphical class diagram and the textual specification of all attributes and services.

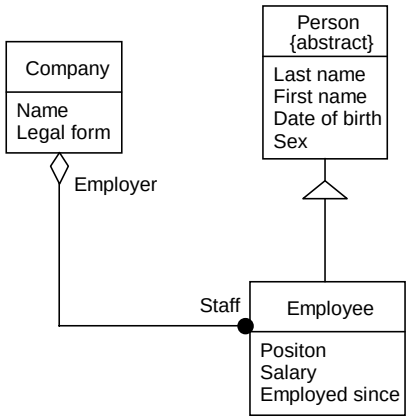

Figure 3. OOA model of the simple example

In order to support a systems analyst in this task, a form was drafted that makes the information accessible by the generator system. As an example, one can indicate the following specifications for the some attributes of the class *Company*:

Attribute **Name:**
Ergonomic Name: Company Name
Type: String, 30 characters maximum
Mandatory Attribute
Part of the Primary Key
persistent

Attribute **Legal_Form:**
Type: Enumeration, not expandable,
0 to N selections
Selection Possibilities: inc, ltd, corp, co-op
no default value
persistent

If the system analyst has finished his work he can export the OOA model form the OOA CASE tool in form of a JDL file. Appendix A shows the JDL file corresponding to the OOA model of figure 3. Now the generation can be carried out with this JDL file. The result is an application whose generated code for the problem domain component corresponds to the OOD model depicted in figure 4.

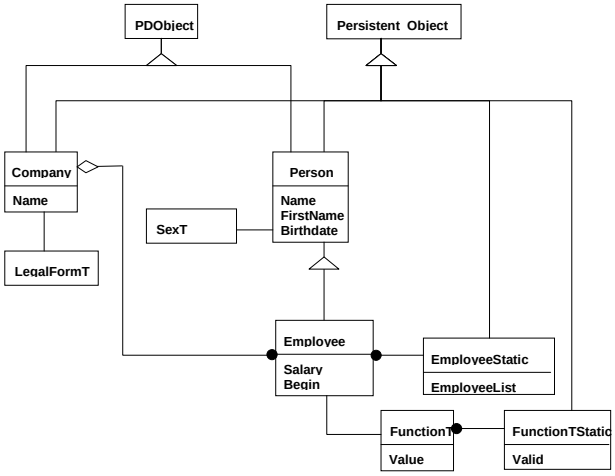

Figure 4. Simplified OOD model of the generated application

Each class of the OOA model inherits *PDObject*, an abstract base class. The operations provided by *PDObject* for typeless access to the attributes are of course redefined in the classes of the OOA model. They serve primarily to connect to the user interface and are described later.

All classes with persistent attributes or relations have to inherit from the class *d_Object*. This is a *mixin*-class which implements the manipulation components of the ODMG conforming database. As a general matter, each class has to know the instances it produced. Therefore, each class is assigned a list which collects references to all objects of that class. This list is implemented as a class attribute. Since the ODMG standard does not support persistent class attributes, an aid must be constructed. An object of the *EmployeeStatic* class saves the persistent class attrib-

utes of employees, being the necessary lists in this case. The generated code ensures that only one instance of *EmployeeStatic* exists in the database at any given time (depicted only for the *Employee* class in figure 4).

Another aid is necessary for the description of enumeration types. A new class has to be generated for each enumeration type since the characteristics of the enumeration types (multiple choice, expandability) go beyond the concepts provided by the programming languages.

If one examines the attribute *Function* of the class *Employee*, another problem becomes apparent. This enumeration type is to be expandable, and the expansions are to be persistently cared for in the entire application. Therefore, the lists of the expansions have to be a persistent class attribute of the data type *FunctionT* and, thus, have to be administered in a further help class, *FunctionTStatic*.

The GUI generator creates the user interface itself as well as its linkage to the problem domain component. First, an interaction object corresponding to the attribute type in OOA is selected for each attribute to appear on the user interface. Further, the generator first evaluates the information prepared in an instance of the meta model. If the selected interaction objects would not fit into the window, the generator decides to use either less space consuming interaction objects (if such alternatives exist in its knowledge base) or to split the window into sub windows.

The position of the individual interaction objects in the dialogue windows is determined by the class definition in the object model and by the evaluation of inheritance hierarchies [Balzert93].

Associations and aggregations are transformed in lists. Figure 5 depicts two entry forms generated for the example described above. The class *Person* is abstract and therefor does not appear as an own dialog. It is however visible as a group in the entry form of *Employee*. Since an aggregation with the employees exists in the fundamental specification of the class *Company*, a list of the employees employed by the company is found in the Company dialog.

Conversely, the class *Employee* is the part-of class of this aggregation. Therefore, the employer does not appear in an employer's entry form. The transformation of a relationship to the GUI selected here is just one of the many possibilities. Software ergonomics can both globally prescribe the transformation to be used as well as subsequently change it in particular cases. Of course, this not only applies to the depiction of relationship, but also applies to other aspects of user interface generation (color preferences [Heintzen95], fonts, selection of interaction objects [Bodart94c],...).

The generator system additionally connects the GUI with the code frame representing the problem domain. When selecting "Employee create..." from the application's main menu the entry form for employee is opened. In the same moment a new problem domain object employee is created and linked to the entry form.

Now the user is able to enter the new employee's data. By clicking the OK button the entered data will be transferred from the GUI to the linked problem domain object. If all values are valid (as specified in the OOA model) the entry form closes and the linked object receives the message to store itself in the database.

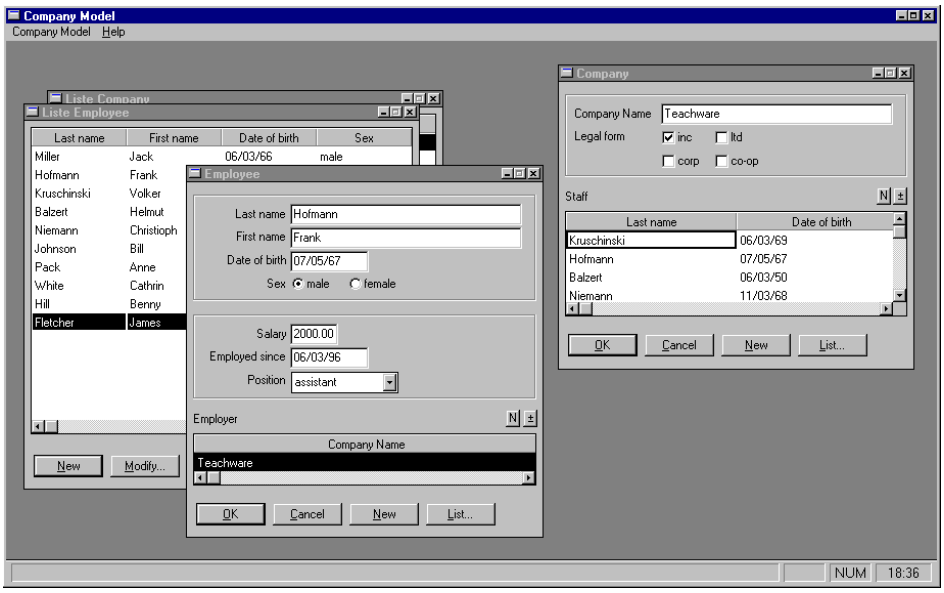

Figure 5. Generated dialog windows for the sample application

The connection GUI-problem domain works even in the other direction. If the user wants to modify the former entered data he has first to select the special employee via a list view. When selected a single employee the employee's entry form opens. The entry form is linked with the corresponding problem domain object. The data are now transferred from this object to the controls of the entry form. The user is able to access the previously entered data for further processing.

4 Technical Solution: Designing the User Interface

Before generating the source code for the application, the data keeping, the interface and the binding, the user interface has to be designed. The starting point is the OOA model. Combining the model's semantics with the selected design strategy will produce all necessary windows including their controls. The standard transformation of a model is described in [Balzert95a, Balzert95b]. The dialog design strategy gives information which additional and standard functionality is taken into consideration, including its appearance.

Additionally the decision is made which possibility is chosen when there are different theoretical possibilities. For example, displaying an association as a table or simply making it a menu entry that calls a connection window when selected. Most

strategies are adjustable, so using a different strategy will lead to several different user interfaces of the same problem domain.

The basic controls are chosen from the attribute specification of the OOA model. Normally each attribute is transformed into an control with a belonging label, exceptions are elements that are marked as non UI relevant. Supported controls are: edit field, text field, combo boxes, drop-down combo boxes, list box, drop-down list box, check boxes, radio buttons and tables. The result of the transformation, is an object network with windows and their elements. They have to be arranged for generating the source of the user interface.

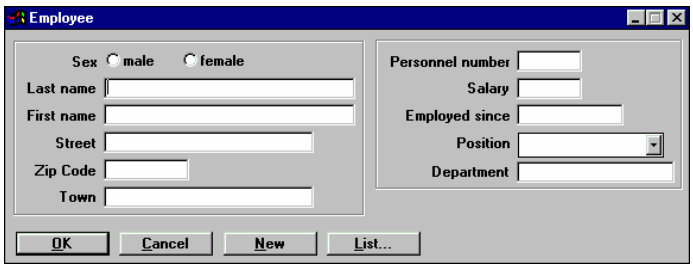

Figure 6a. Different layout by choosing different settings: window A

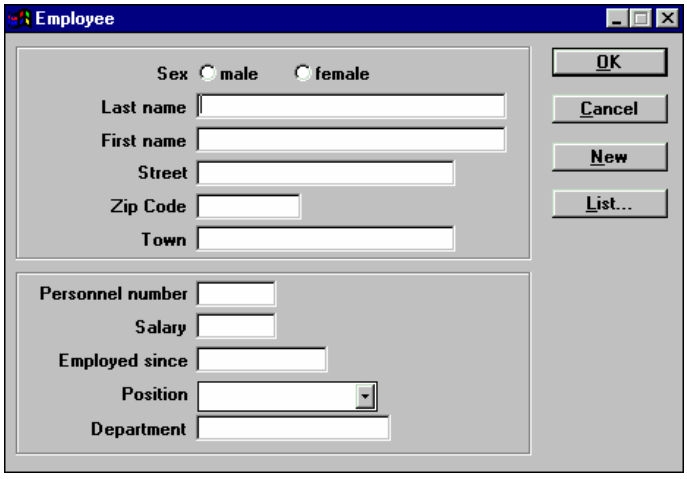

Figure 6b. Different layout by choosing different settings: window B

JANUS uses several placement strategies and has access to ergonomic knowledge. So once again, different static layouts can be produced for a single application, because the user working with JANUS can change aspects of the layout strategy to change the layout. Finally the version fitting the user's needs best can be chosen for the application.

The main goal of the layout component of JANUS is to get a well balanced layout. So there is a special focus on alignment of the elements. To get equilibrium mask the two column placement [Vanderdonckt94d] gives the best results. The different

between the strategy of GENIUS [Janssen93] is the involving of groups to combine equal information. The benefit is a more compact layout.

Groups result from inheritance or embedded types (structs) in the OOA model. So there is the possibility for arranging all elements into two columns or placing the groups into two column. One column layout is also possible, it will depend on the number of interaction elements and the setting of the layout component.

All interaction elements will be left justified. There will be an overall alignment of interaction elements, whether they are in groups or not. The push button for navigation or for the operations can be placed left or at the bottom. It is adjustable if they are centered, set in a block or if they are set from left/top with equal distances.

In addition to the placement strategy the over grid, and all distances between interaction elements are to be set. All settings are saved in *initialization files,* so a simple reuse is possible. It is also possible to define parts of style guide in the settings. Figures 6a and 6b show a result of choosing different settings. Further examples are shown in figures 4, 10 and 11.

5 Technical Solution: Connecting the GUI with the Problem Domain Component

In all of the later mentioned approaches, the automated linkage of the generated user interface to the core of the application is not taken into consideration. This problem is solved by the improved JANUS system envisaged here with the help of the *JANUS Application Framework* (*JAF*). This is a highly specialized class library which serves the preparation of the basic functionality in the considered environment.

The environment is characterized by an ODMG conforming, object oriented database, a GUI development system, and an object oriented programming language (C++). The research prototypes of PICASSO [Rowe91] and ACE [Johnson93, Zarmer92] as well as the commercial products like zApp [Inmark94] contributed to the draft of the *JANUS Application Framework*.

The user interface constitutes a limited number of various elementary interface objects or GUI widgets which differ substantially by their position, appearance, as well as relationship to one another. The GUI generator has to first select the appropriate interface objects and see to the technically correct placement and appearance from the objects' parameterization.

Conventional GUI development systems require a programmer to couple the final user interface with the technically specific portion of the application using *Callback* mechanisms. The tasks of Callback procedures may be assigned to various categories [Myers91].

- The interaction object's internal representation often differs from the expected representation the application object's attribute. The type conversions have to be expressly carried out by a programmer.
- It must be checked whether the application satisfies the restrictions specified in the problem domain component before an application further processes data entered from an end user.
- The links among the interface objects have to be established to let the user navigate through the application

To solve this problem, JAF provides special operations at higher abstraction levels. Access operations automatically created by the App-generator are used on the attributes of the OOA objects in these operations. This results in a significant reduction of the complexity of the user interface specific code.

These facts are reflected in the generated source code. The code fragments for the implementation of the same interface objects in miscellaneous contexts vary substantially by a different parameterization in their dynamic creation. The generated code is significantly shorter than previously since a large portion of the aforementioned tasks are encapsulated in the classes of the application framework.

In most cases, controls function on a string basis. To keep the costs on the GUI side low and to ease the adaptation to various GUIs, the linkage of the interaction objects to an OOA model's attribute of a random type was solved with the parameterization.

An interaction object only has to know the name of the appropriate attribute and a reference to the respective object. The type of the attribute bound to the interaction object does not need to be known although the type, of course, has an impact on the parameterization of the interaction object. For example Boolean types are represented by two radio buttons etc.

Therefore, each class of the OOA model has to provide a single operation which allows read and/or write access to all attributes. Since a strict type concept is to be at the target language's disposal but the interaction objects do not have the type information, a data type has to be found upon which random types can be reproduced. Therefore, the use of strings and/or lists of strings is ideal for an exchange format between the GUI and the OOA model. An operation's declaration in C++ for reading a random attribute of a class can be expressed as follows:

```
bool GetAll(const char *name, String &val) const;
```

The first parameter serves to select the attribute. The second parameter is a place holder which receives the value of the attribute converted to a string. Whether an attribute with the desired name exists or not is shown by the Boolean return value of the operation. If the attribute does not exist in the class requested, all of the base classes must naturally be consulted first. The failure of the operation will be reported only if the attribute searched for is not present here.

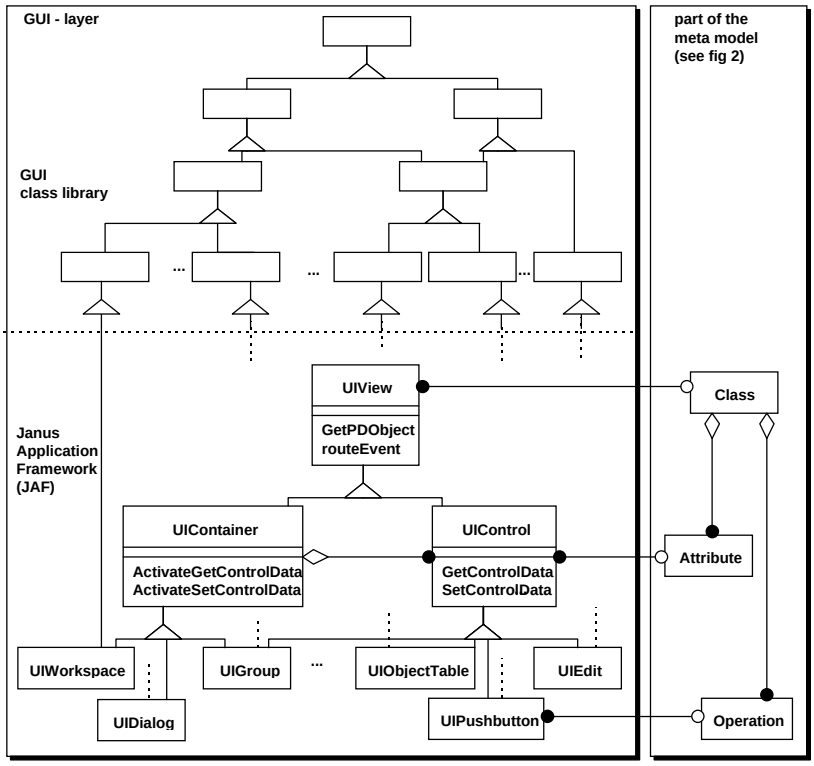

Figure 7. Communication between GUI - and application layer

Another (overloaded) version of these operations supports the reading and writing of an attribute with the help of a list of strings where such a list makes sense. Lists can be used, for example, for accessing an enumeration type which allows several simultaneous selections.

Figure 7 demonstrates the effects of these technical solutions on the architecture of the *Janus Application Framework*. Conventional GUI class libraries are expanded by subclasses. The new classes are attached as leafs to the inheritance hierarchy of the class library. By multiple inheritance from the abstract classes of *UIView*, *UIContainer*, and *UIControl*, they obtain operations which produce a connection to the attribute (and/or the operation) of the OOA class and/or of the object.

The cost of expanding a GUI class library with the desired functionality depends upon its structure. A UIMS with a C based API (Open Interface [Neuron91, Neuron93]) as well as the purely object oriented implemented UI builder (zApp) [Inmark94] and StarView [Star93] were expanded to meet the additional requirements. *UIView* presents an abstract base class for all visible elements of the user interface. It expands its derived classes to concepts for the connection of the OOA model's objects.

All GUI classes that inherit characteristics from *UIView* have the possibility to communicate with the OOA model objects by the polymorph (virtual) operation *GetPDObject()*. Every *UIView* class can be directly linked to an object of an OOA class by subclassing. The newly formed classes are enhanced with the reference to the OOA object which can be accessed by the redefined operation *GetPDObject()*. However, only container objects or complex controls are generally directly linked to an OOA object. Simple controls can access an OOA object through the overarching container object.

To make these mechanisms available, the *UIControl* and *UIContainer* classes form specializations of *UIView* with which the characteristics of atomic interaction objects and their groupings can be described.

Interaction objects are always positioned in an object of the *UIContainer* class that administrates it. Container objects can, however, also contain subordinate containers. All messages, which a container receives, are automatically forwarded to its sub-containers. By activating the operation *GetControlData()*, the *UIContainer* object can command its interaction object to register its contents in the corresponding attributes of the OOA object.

Conversely, the *UIContainer* object with help from *GetControlData()* sees to the transfer of the attribute values from the OOA object to the representation of the respective interaction objects. Groups and windows are the various variants of the *UIContainer* class.

Contrary to conventional GUI class libraries, the *UIGroup* class does not only serve the visual arrangement of multiple interaction objects. The described mechanisms create a simplistic means to access the attributes of several OOA objects in one window. An object of the *UIWorkspace* class functions as an application's main window. It administrates its sub-windows and offers the functionality to arrange these windows or to switch between them. If the application window is closed, the application will be quit. Apart from that, a Workspace object has to establish a link to the object oriented database which is used for the application's data keeping.

The end user can access data from the OOA model using dialogue windows. One or several OOA objects can be displayed or modified by the dialogue window. At the beginning of a dialogue, an OOA object has to be assigned to the dialogue window and to each of the groups or complex interaction objects which might be present inside of this window. The attribute values of the OOA objects are transformed in the internal representation of the interaction objects and presented to the user.

If the end user has changed the content of the interaction objects and chooses to save them, a message is sent all interaction objects subordinate to the dialogue commanding them to remit their contents to the attributes of the corresponding OOA objects.

UIControl is the baseclass for all of the controls supported by the corresponding GUI class library. Each control is linked to an attribute of a specified OOA model class. This link can be removed and changed during runtime. Each control not directly linked to an OOA object knows its *UIContainer* object and thereby has access to the specific OOA object which is supposed to represent it. Complex controls can also be directly linked to an OOA object by forming sub-classes. The moment for transferring the attribute values to and from the OOA object can be controlled by sending messages to the control.

Every control provides two polymorph operations, *SetControlData()* and *GetControlData()* for this, which are redefined in each of the classes derived from *UIControl*.

SetControlData() copies the data from the OOA object to the internal representation of the control. This occurs in the following steps:

- With help from the operation *GetPDObject()*, the control receives a reference to the assigned OOA object over a direct link (C++ pointer or ODBMS intelligent pointer) or its *UIContainer* object.
- Since each control is assigned to an attribute, the OOA object's *GetAll()* operation can be called to get the current attribute value.
- The attribute value is transformed to the internal representation of the control and presented to the user. This transformation is actually done by the generated *GetAll()* operation of the OOA object.

The *GetControlData()* operation does the transformation in the other direction in a similar way. Additional actions can be taken in it if the internal value of the control cannot be filed as an attribute value. The reason for this may be a limit error or a violation of a restriction which was defined in the OOA model for that particular attribute. In the current version of the JADE system, the *PutAll()* operations return an error code in such cases. The user interface component uses this error code to display a dialog box which tells the users which attribute has an illegal value and why the value is wrong.

The table has a special role under the interaction objects. One or more attributes from a set of OOA objects can be reported with the assistance of the *UIObjectTable* class. The OOA objects to be displayed in the table are generally instances of the same class. At least they have to inherit from the same base class.

Access to the string based *GetAll()* and *SetAll()* operations separates the complicated internal design of a table object from its mere interface. Aside from that, this program allows the modification of the display shape of a table during the application's running time.

In this way, for example, the order of the attributes to be displayed, as well as the selection of the attributes themselves, can be interactively adjusted by the end user. The effort required to create appropriate dialogues to activate these preferences is

drastically reduced if the meta data are a permanent part of the problem domain component.

It is also easy to achieve direct editing of the attribute values in the table by exchanging data with the OOA object using strings and the above mentioned mechanism.

6 Technical Solution: the Generator Systems

The target language which is to be used also plays a role in the generation process. As a rule, code for multiple target languages must be created depending on the application environment. If the application is to be client-server capable using CORBA standards [OMG91], the interface of all classes has to be generated in the declarative language IDL. A similar situation is presented by the use of an ODMG conforming, object oriented database. The code here has to be created in the language ODL.

The App-generator system as well as the GUI generator system builds on assumptions about the target language. The following concepts are to be supported:

- the module concept;
- the ability to define additional data types;
- the pointer concept (or a similar concept that allows *smart pointers*);
- the ability to define "free functions" (functions not belonging to any class).

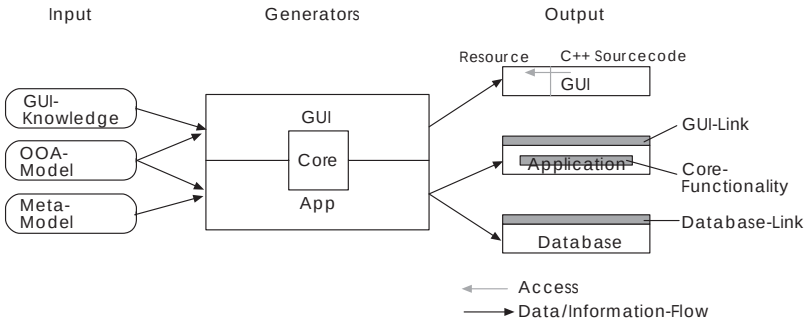

Figure 8. The generator systems

All other information is encapsulated in a general generator core in a meta model of the utilized programming language. Figure 8 depicts the generator system with the corresponding inputs and outputs. It is possible to change the target language within the scope of the displayed limits by changing the meta model. The individual generators are linked to the respective language by a temporary association. In this way it is possible to generate parts of the application in different languages without altering the generators' interfaces.

It is to be shown by means of example how the declaration of a specific class's operation can be described with the assistance of a generator core. The result is the

generation of an operation for reading any attribute of the class the operation belongs to.

This operation contains the name of the attribute to be read in the first parameter. It is a string constant. The temporary association for the generator core is represented by *L*. It is a pointer to a C++ object.

```
JParameter p1(L->AttribName(), L->String(),
        NULL, Param_In);
```

The second parameter will be filled with the value of the attribute. This is a reference to a string functioning as an output parameter.

```
JParameter p2(L->AttribVal(), L->String(), NULL,
        Param_Out | Param_Ref);
```

This example shows that the purpose of *JParameter* is to describe Parameters of an Operation to be generated. Each Parameter has a mandatory name an type and optionally a default value and some flags that describe special characteristics of the parameter.

The operation's actual declaration is generated by the following command:

```
L->Method(decl, L->Bool(), GetName(),
        L->GetAll(), p1, p2,
        Meth_Poly | Meth_Declare | Meth_Const);
```

By calling this Operation of the Language class, an Operation with the Name given by *L->GetAll()*, belonging to a class that is named by *GetName()*, is declared. The return type is boolean *(L->Bool())*, and the parameter list is *p1*, *p2*. The generated operation is to be constant (only read access to the attributes) and polymorph (*late binding*). This is specified by the flags Meth_Poly and Meth_Const. The declaration will be written to the ostream object decl.

If one observes the generation of the resource files for which the aforementioned generator core cannot be used, one discovers basically comparable situation. The syntax for defining resources in different window systems or UI builders is different for the same interface objects, although the semantics basically remain the same.

Each generator system is specialized for a particular area of tasks. Therefore, the mechanisms for code generating describe above have to be individually controlled. In the case of the App-generator, the controlling mechanism is already given by the application's meta model. The generator system's task is to read information from the model and transform it to the generating commands. Dependencies between individual classes, which can affect the distribution of the created code in different files, have to be taken into consideration.

The GUI generator system consists of three levels. The lowest level provides the functionality described above for simplified source generating as well as for creating resource files. The middle level comprises a meta model of the JANUS Application Framework.

A class is implemented in the generator for each corresponding class in the JANUS Application Framework. This middle level can independently create the resource files and the accompanying source code in the target language by using the lowest level. The main problem is the parameterization of the middle level.

A third level serves to solve this problem. It controls the course of events in the generation and establishes a link to the application's OOA model. Preferences are made in it that affect the application's screen display.

Dialogue information, layout information, conventions, and the application's self-portrait flow in here [Balzert94]. In accordance with these defaults and the transformational rules, an object network of the interface object-meta model, which is defined in the second level, is created. The completed user interface results from this.

7 A More Complex Example

To evaluate whether JANUS can handle more complex OOA models and to see the advantages of JANUS, we have tested our system with a model of a seminar organization (see figure 9). The model describes the problem domain of a company that organizes seminars. Data of customers, lectures, companies (in the role of customers), booking and the seminaries themselves can be created, stored or changed.

Also information about connections can be handled, e.g., which lecture can lecture on which kind of seminar. For further information, see figure 9. The model includes 15 classes, 67 attributes, 17 operations and 8 connections (associations or aggregations, not counting inherited connections).

The model was made with the OOA tool Paradigm Plus. All necessary specifications were made with this tool. By using a self written script the Paradigm Plus generates a JDL file. This file is the input of the JANUS system. The generation process results in four C++ files consisting of 22566 LOC that can be compiled to a running application. About 50% of the generated code implement the GUI. The rest implements the problem domain with the implicit operations and the connection to the GUI and database. The GUI code implements 826 GUI widgets.

The resulting application—remember: finished without writing a single line of code—was tested with some demonstration data. The application JANUS has produced includes all the described basic functionality, including persistence and an acceptable graphical user interface.

Figure 10 shows how to handle single objects by using the list view. For this example we have chosen the lectures class. Figure 11 shows all established connections (upper list in the front dialog) and how to modify the connections between the classes lecture and type of seminar.

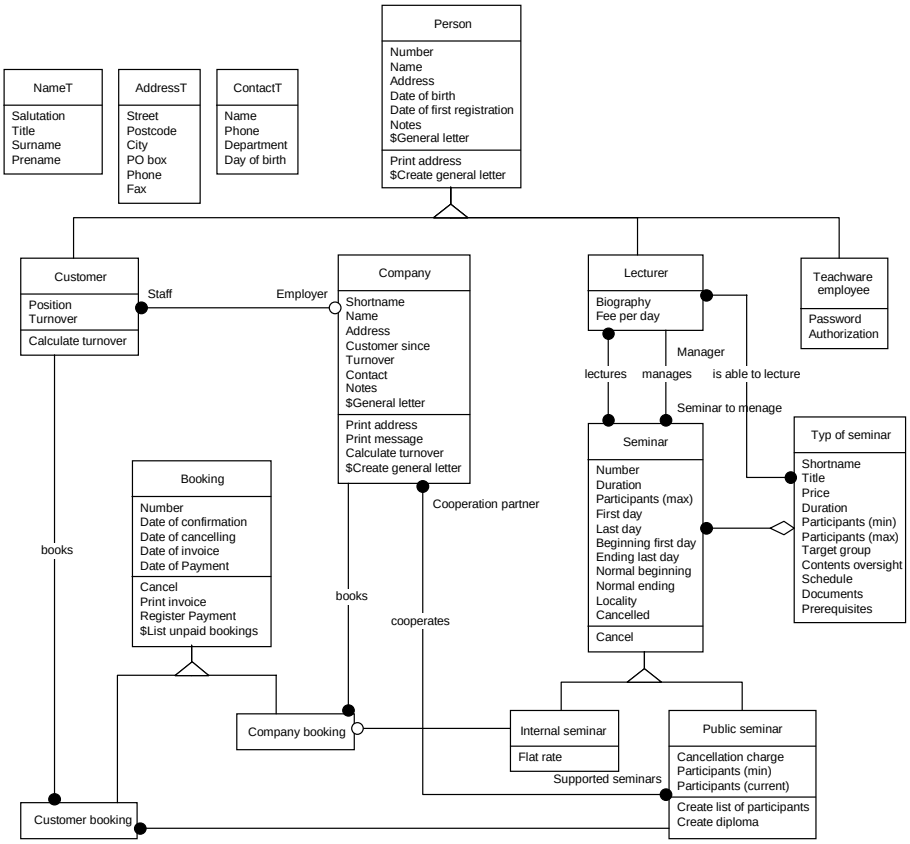

Figure 9. The OOA model of the seminar organisation

8 Related Research Approaches

MB-IDEs [Foley91, Szekely93, Szekely96] offer the possibility to describe user interfaces at a higher level of abstraction. They demonstrate a beginning basis for an automated GUI development from the data model of the actual application. The UIDE environment was expanded from a data model by de Baar et al. [de Baar92] to tools for generating the static layout.

Even other approaches deal with knowledge-based selection of interaction objects corresponding to a data model [Johnson92a, Vanderdonckt93] and their layout in dialogue windows [Kim93]. In an additional step, parts of an application's dynamic behavior will be included in GUI generating. Gieskens and Foley [Gieskens92] enhance the interface objects used by UIDE with pre- and post-conditions to thereby describe their relationships.

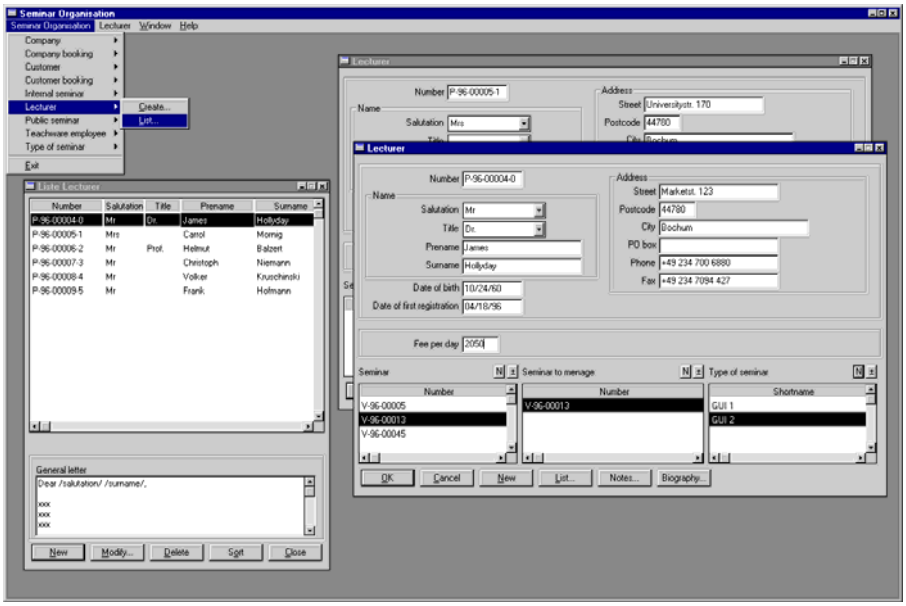

Figure 10. Navigating from list to single objects

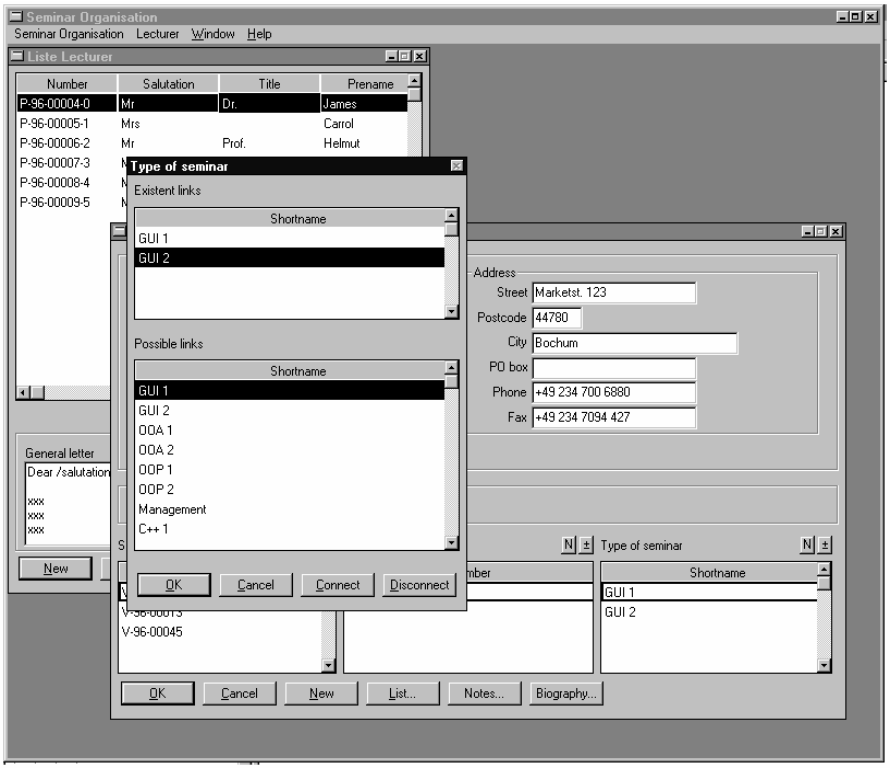

Figure 11. Establishing connections between objects

Janssen et al. [Janssen93] use a graphical editor in GENIUS in addition to an entity-relationship-data model in order to input special dialogue networks for each application. These approaches have the disadvantage of necessitating exceedingly high costs on the part of the developer particularly for complex applications, and therefore neutralizes the advantages of automated generation.

Both JANUS [Balzert93, Balzert94, Balzert95a, Balzert95b] and MECANO [Puerta-94b, Puerta96] use the relationship between the objects identified in the problem domain component and a generation of the dynamic behavior of an application. While a language of its own similar to LISP was drafted for MECANO to describe the problem domain component, the modeling of the problem domain component for the JANUS system was based on the object oriented analysis (OOA).

Conclusion: Actual Status of Development

The system for automated application development described has been essentially completed. Beginning with an OOA model, the application is completely generated with the standard functionality (new, change, delete, find) including a user interface and linkage to an ODMG compatible (Poet 4.0 or O_2), object oriented data base management system, and subsequently can be put into used. The expansion of the App-generator system is in the works in order to also be able to generate client/server capable applications. The client/server part will be based on an object request broker that conforms to the OMG standard.

The generator itself is written in C++ and consists of about 130,000 lines of code. It runs on systems with an AT&T 3.0 compatible C++ compiler and was successfully tested using various compilers and operating systems. The generated applications need the zApp or StarView class library to run. These libraries are available for various GUIs so that the generated application runs at least under Windows, Motif and OS/2. When generating for the Windows environment, an online help system using the windows help format is also generated. For the Unix platform we generate the same information as HTML files. These files can be browsed by any HTML viewer such as Netscape or Mosaic.

A versioning system that allows to preserve the user added code for implementing the problem specific functions of each application is also part of the Janus system. This system is very robust against modifications of the OOA model. For example it is possible to change an operation's name in the OOA model without losing the hand made implementation of it.

In the future we will try to connect more CASE tools to our system for entering the model data. We also plan to support different user interface management systems and more database systems including relational DBMS.

For more up to date information you might want to visit our web server at http://www.swt.ruhr-uni-bochum.de.

Appendix A. Exported JDL file from Paradigm Plus.

```
// JANUS Definition Language
//
// Generated by Paradigm Plus on: Tue Apr 09 17:44:40 1996
//
// Paradigm: RUMBAUGH
// Project:  company
// Diagramm: Company Model

MODULE Company_Model
(
   ERGNAME "Company Model"
   DESCRIPTION "A simple example"
)
{
  // interface forward reference(s)
  INTERFACE Company;
  INTERFACE Employee;
  INTERFACE Person;

  INTERFACE Company
  (
    EXTENT CompanyList
    KEY Name
    ERGNAME "Company"
    DESCRIPTION "represents a company with its name its legal form"
    ABSTRACT false
    UIRELEVANT true
  ):PERSISTENT
  {
    // Attribute(s)
    ATTRIBUTE STRING<30> Name
    (
      ERGNAME "Company Name"
      DESCRIPTION "Full name of the company"
      UIRELEVANT true
      CLASSGLOBAL false
      MANDATORY true
      DEFAULTVALUE ""
    );
    ATTRIBUTE ENUM Legal_formT {inc,ltd,corp,co_op} Legal_form
    (
      ERGNAME "Legal form"
      DESCRIPTION "legal form(s) of the company"
      UIRELEVANT true
      CLASSGLOBAL false
      MANDATORY false
      ITEMS "inc","ltd","corp","co-op"
      SELECTMIN 0
      SELECTMAX 4
      EXTENSIBLE false
    );
    // Relation(s)
    RELATIONSHIP LIST<Employee> Staff inverse Employee::Employer
    (
      PATHTYPE Part
      ERGNAME "Staff"
      DESCRIPTION ""
      CARDINALITY [0,N]
      UIRELEVANT true
      UIINFORM true
    );
  };
  INTERFACE Employee  : Person
  (
    EXTENT EmployeeList
    ERGNAME "Employee"
    DESCRIPTION "represents an Employee (person with special properties)"
    ABSTRACT false
    UIRELEVANT true
  ):PERSISTENT
  {
    // Attribute(s)
    ATTRIBUTE FLOAT Salary
    (
      ERGNAME "Salary"
      DESCRIPTION "The monthly salary of an employee (in Dollars)"
      UIRELEVANT true
      CLASSGLOBAL false
      MANDATORY false
      DEFAULTVALUE 2000
      LOWERBOUND 0
      UPPERBOUND 100000
    );
```

```
    ATTRIBUTE DATE Employed_since
    (
      ERGNAME "Employed since"
      DESCRIPTION "Date of employment in the associated company "
      UIRELEVANT true
      CLASSGLOBAL false
      MANDATORY false
      DEFAULTVALUE "current"
    );
    ATTRIBUTE ENUM PositionT {clerk,manager,developer,consultant} Positon
    (
      ERGNAME "Position"
      DESCRIPTION "The employee's position in the company "
      UIRELEVANT true
      CLASSGLOBAL false
      MANDATORY false
      ITEMS "clerk","manager","developer","consultant"
      DEFAULTSELECTED "clerc"
      SELECTMIN 1
      SELECTMAX 1
      EXTENSIBLE true
    );
    // Relation(s)
    RELATIONSHIP Company Employer inverse Company::Staff
    (
      PATHTYPE Whole
      ERGNAME "Employer"
      DESCRIPTION ""
      CARDINALITY [1,1]
      UIRELEVANT true
      UIINFORM true
    );
  };
  INTERFACE Person
  (
    EXTENT PersonList
    KEYS Last_name, Date_of_birth
    ERGNAME "Person"
    DESCRIPTION "abstract class which holds a person's commonly used attributes"
    ABSTRACT true
    UIRELEVANT true
  ):PERSISTENT
  {
    // Attribute(s)
    ATTRIBUTE STRING<30> Last_name
    (
      ERGNAME "Last name"
      DESCRIPTION "Surname(s) of a person"
      UIRELEVANT true
      CLASSGLOBAL false
      MANDATORY true
      DEFAULTVALUE ""
    );
    ATTRIBUTE STRING<30> First_name
    (
      ERGNAME "First name"
      DESCRIPTION "Christian name(s) of a person"
      UIRELEVANT true
      CLASSGLOBAL false
      MANDATORY false
      DEFAULTVALUE ""
    );
    ATTRIBUTE DATE Date_of_birth
    (
      ERGNAME "Date of birth"
      DESCRIPTION "the person's birthday"
      UIRELEVANT true
      CLASSGLOBAL false
      MANDATORY false
      DEFAULTVALUE ""
    );
    ATTRIBUTE ENUM SexT {male,female} Sex
    (
      ERGNAME "Sex"
      DESCRIPTION "the sex of a person can be a male or female"
      UIRELEVANT true
      CLASSGLOBAL false
      MANDATORY false
      ITEMS "male","female"
      DEFAULTSELECTED "male"
      SELECTMIN 1
      SELECTMAX 1
      EXTENSIBLE false
    );
  };
};
```

References

[Balzert93] Balzert, H., *Der JANUS-Dialogexperte: Vom Fachkonzept zur Dialogstruktur*, in Softwaretechnik Trends, Band 13, Heft 3, Proceedings der GI-Fachtagung Softwaretechnik, Dortmund (8-10 November 1993), pp. 62-72.

[Balzert94] Balzert, H., *Das JANUS-System: Automatisierte, wissensbasierte Generierung von Mensch-Computer-Schnittstellen,* in Informatik-Forschung Entwicklung, Vol. 9, Springer-Verlag, Heidelberg, 1994, pp. 22-35.

[Balzert95a] Balzert, H., *From OOA to GUI - The JANUS-System*, in Proceedings of the 5th IFIP TC13 Conference on Human-Computer Interaction INTERACT'95, Lillehammer, 25-29 June 1995, K. Nordbyn, P.H. Helmersen, D.J. Gilmore and S.A. Arnesen (Eds.), Chapman & Hall, London, 1995, pp. 319-324. http://www.swt.ruhr-uni-bochum.de/forschung/janus/lillehammer. html

[Balzert95b] Balzert, H., Hofmann, F., Niemann, C., *Vom Programmieren zum Generieren - Auf dem Weg zur automatischen Anwendungsentwicklung*, in Proceedings of GI-Fachtagung Software-technik'95 (Braunschweig, October 1995), 1995, pp. 126-136. http://www.swt.ruhr-uni-bochum.de/forschung/swt95/artikel.htm

[Blaha94] Blaha, M., Premerlani, W., Shen, H., *Converting OO Models into RDBMS Schema*, IEEE Software, May 1994, pp. 28-39.

[Bodart94c] Bodart, F., Vanderdonckt, J., *On the Problem of Selecting Interaction Objects*, in Proceedings of British Conference on Human-Computer Interaction HCI'94 « People and Computers IX » (Glasgow, 23-26 August 1994), G. Cockton, S.W. Draper, G.R.S. Weir (Eds.), Cambridge University Press, Cambridge, 1994, pp. 163-178. http://www.info.fundp.ac.be/cgi-bin/pub-spec-paper? RP-94-018

[Booch94] Booch, G., *Object-Oriented Analysis and Design with Applications*, The Benjamin/Commings Publishing Company, 1994.

[Coad91a] Coad, P., Yourdon, E., *Object-Oriented Analysis*, Prentice-Hall, 1991.

[de Baar92] de Baar, D.J.M.J., Foley, J., Mullet, K.E., *Coupling Application Design and User Interface Design*, in Proceedings of the Conference on Human Factors in Computing Systems CHI'92 « Striking a balance » (Monterey, 3-7 May 1992), P. Bauersfeld, J. Bennett, G. Lynch (Eds.), ACM Press, New York, 1992, pp. 259-266. ftp://ftp.gvu.gatech.edu/pub/gvu/ tech-reports/91-10.ps.Z.

[Foley91] Foley, J.D., Kim, W.C., Kovacevic, S., Murray, K., *UIDE - An Intelligent User Interface Design Environment*, in « Intelligent User Interfaces », J.W. Sullivan, S.W. Tyler (Eds.), Addison Wesley, ACM Press, 1991, pp. 339-384.

[Gieskens92] Gieskens, D.F., Foley J.D., *Controlling User Interface Objects through Pre- and Postconditions* , in Proceedings of the Conference on Human Factors in Computing Systems CHI'92 « Striking a balance » (Monterey, 3-7 May 1992), P. Bauersfeld, J. Bennett, G. Lynch (Eds.), ACM Press, New York, 1992, pp. 189-194.

[Heintzen95] Heintzen, P., Kruschinski, V., Balzert, H., *Ein wissensbasiertes System zur Unterstützung des Benutzers bei der ergonomischen Farbzusammenstellung für Dialogmasken*, Tagung Software-Ergonomie'95 (Darmstadt, 1995). *zApp Application Framework V2.2*, Programmers Guide, Inmark Development Corporation, , Mountain View, 1994.

[Janssen93] Janssen, C., Weisbecker, A., Ziegler, J. , *Generating User Interfaces from Data Models and Dialogue Net Specifications*, in Proceedings of the Conference on Human Factors in Computing Systems INTERCHI'93 « Bridges Between Worlds » (Amsterdam, 24-29 April 1993), S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel, T. White (Eds.), ACM Press, New York, 1993, pp. 418-423.

[Johnson92a] Johnson, J.A., *Selectors: Going Beyond User Interface Widgets*, in Proceedings of the Conference on Human Factors in Computing Systems CHI'92 « Striking a balance » (Monterey, 3-7 May 1992), P. Bauersfeld, J. Bennett, G. Lynch (Eds.), ACM Press, New York, 1992, pp. 273-279.

[Johnson93] Johnson, J.A., Nardi, B.A., Zarmer, C.L., Miller, J.R., *ACE Building Interactive Graphical Applications*, Communications of the ACM, Vol. 36, No. 4, April 1993, pp. 41-55.

[Kim93] Kim, W.C., Foley, J.D., *Providing High-level Control and Expert Assistance in the User Interface Presentation Design*, in Proceedings of the Conference on Human Factors in Computing Systems INTERCHI'93 « Bridges Between Worlds » (Amsterdam, 24-29 April 1993), S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel, T. White (Eds.), ACM Press, New York, 1993, pp. 430-437.

[Loomis93] Loomis, M.E., *The ODMG Object Model*, Journal of Object-Oriented Programming, Vol.6, No.3, June 1993, pp. 64-69.

[Myers91] Myers, B.A., *Graphical Techniques in a Spreedsheet for Specifying User Interfaces*, in Proceedings of the Conference on Human Factors in Computing Systems CHI'91 « Reaching through technology » (New Orleans, 27 April-2 May 1991), S.P. Robertson, G.M. Olson, J.S. Olson (Eds.), ACM Press, New York, 1991, pp. 243-256.

[Neuron91] Open Interface V3.0, *Open Interface Toolkit*, Neuron Data, Inc., Palo Alto, 1991.

[Neuron93] Open Interface V3.0, *Development Guide*, Neuron Data, Inc., Palo Alto, 1993.

[OMG91] *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1, December 1991.

[Puerta94b] Puerta, A.R., Eriksson, H., Gennari, J.H., Musen, M.A., *Beyond Data Models for Automated User Interface Generation*, in Proceedings of British Conference on Human-Computer Interaction HCI'94 « People and Computers IX » (Glasgow, 23-26 August 1994), G. Cockton, S.W. Draper, G.R.S. Weir (Eds.), Cambridge

University Press, Cambridge, 1994, pp. 353-366. http://www-ksl.stanford.edu/KSL_Abstracts/KSL-93-62.html

[Puerta96] Puerta, A.R., *The MECANO Project: Enabling User-Task Automation During Interface Development*, in Proceedings of AAAI'96 Spring Symposium on Acquisition, Learning & Demonstration: Automating Tasks for Users (Stanford, March 1996), AAAI Press, pp. 117-121.

[Rowe91] Rowe, L.A., Konstan, J.A., Smith, B.C., Seitz, S., Liu, C., *The PICASSO Application Framework*, in Proceedings of the 4th Annual Symposium on User Interface Software and Technology UIST'91 (Hilton Head, 11-13 November 1991), ACM Press, New York, 1991, pp. 95-105.

[Rumbaugh91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, 1991.

[Star93] Star Division, *StarView programmer's guide*, 1993.

[Szekely93] Szekely, P., Luo, P., Neches, R., *Beyond Interface Builders: Model-Based Interface Tools*, in Proceedings of the Conference on Human Factors in Computing Systems INTERCHI'93 « Bridges Between Worlds » (Amsterdam, 24-29 April 1993), S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel, T. White (Eds.), ACM Press, New York, 1993, pp. 383-390. http://www.isi.edu/isd/Interchi-be-yond.ps

[Szekely96] Szekely, P., *Retrospective and Challenges for Model-Based Interface Development*, in this volume, pp. xxi-xliv. http://www.isi.edu/isd/Mastermind/Internal/Files/DSVIS96/paper.ps.Z

[Vanderdonckt93] Vanderdonckt, J., Bodart, F., *Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection*, in Proceedings of the Conference on Human Factors in Computing Systems INTERCHI'93 « Bridges Between Worlds » (Amsterdam, 24-29 April 1993), S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel, T. White (Eds.), ACM Press, New York, 1993], pp. 424-429. http://www.info.fundp.ac.be/cgi-bin/pub-spec-paper?RP-93-005

[Vanderdonckt94d] Vanderdonckt, J., Ouedraogo, M., Yguietengar, B., *A Comparison of Placement Strategies for Effective Visual Design*, in Proceedings of British Conference on Human-Computer Interaction HCI'94 « People and Computers IX » (Glasgow, 23-26 August 1994), G. Cockton, S.W. Draper, G.R.S. Weir (Eds.), Cambridge University Press, Cambridge, 1994, pp. 125-143. http://www.info.fundp.ac.be/cgi-bin/pub-spec-paper?RP-94-019

[vander Zanden90] vander Zanden, B., Myers, B.A., *Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces,* in Proceedings of the Conference on Human Factors in Computing Systems CHI'90 « Empowering People » (Seattle, 1-5 April 1990), J. Carrasco, J. Whiteside (Eds.), ACM Press, New York, 1990, pp. 27-34.

[Zarmer92] Zarmer, C.L., Cew, C., *Frameworks for Interactive, Extensible, Information-Intensive Applications*, in Proceedings of the 5th Annual Symposium on User Interface

Software and Technology UIST'92 (Monterey, 15-18 November 1992), ACM Press, New York, 1992, pp. 33-41.