# Generation of Interactive Visual Environments for Direct Manipulation of Database Content

Andreas Dangberg, Wolfgang Mueller
*C–LAB, Paderborn, Germany*
*sprudel@c-lab.de, wolfgang@acm.org*

## 1 Motivation

Though database applications are evolving there are presently only very few system for rapid development of more general interactive visual front-ends. Tailoring non-standard graphical user interfaces to different views of a database is always an error prone and time consuming activity.

Database frameworks like Microsoft Access, Borland Paradox, or Visual dBase come with basic graphical user interface toolkits for rapid development of limited client applications. A couple of systems provide so-called reports for data representation and additional forms with a limited set of tool-specific user interface controls for data manipulation. Data aware controls in form of text fields or combo boxes support additional, mostly text-oriented, data input and manipulation.

On the other hand there are systems for generation of visual interactive systems. They can be distinguished in

- systems for the generation of visual language environments like VLCC [3, 5], and
- approaches to the specification and generation of interactive visual interfaces by Dialogue-Nets, Petri-Nets, UAN (User Action Notation), and ODSN (Object-oriented Dialogue Specification Notation), etc. [4, 6].

Considering all the above approaches we can generally observe that user interface builders for databases are limited to form-based visual interfaces or to very application-specific visualization. Approaches to user interaction specification mainly focus on the user dialogue rather than on the specification of a visual language. Environments for generation of visual language environments, in contrast, focus on the visual language with standard commands for the editing environment. Those environments additionally have no support for database integration and are typically limited to a set of predefined graphical symbols and widgtes. They cannot be easily tailored to different sorts of layout requirements and are often limited to one class of visual languages like diagramatic languages.

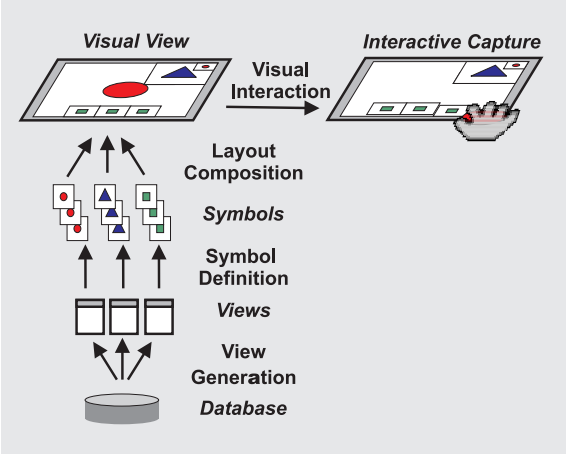


Figure 1: Automatic Generation of Interactive Environments

## 2 VIVID

In this article we present the VIVID (Visual Interactive VIew Development) framework which support the rapid development of interactive views for direct manipulation of database objects through arbitrary combinations of diagrammatic and icon–based visual languages. A VIVID view visualizes a database configuration by filtering and transforming data to graphical objects in two-dimensional space. Objects are represented by arbitrary graphical symbols and their relationships are given as spatial relationships or connections between them.

Visual tokens directly correspond to objects or object groups and their manipulation by, e.g., drag&drop, directly correspond to database operations like creation, deletion, or manipulation of attribute values. The interactive visual view is automatically generated from the definition of

1. symbols and their relation to data source,
2. their layout composition, and
3. their visual interaction

as it is sketched in Figure 1.

Given a relational database we first have to define composition of symbol classes and their association to entity types in the database. By the use of a symbol editor we define the visual representation of those symbol classes and their spatial hierarchy. Figure 2 gives an example with tree-structured symbol hierarchy w.r.t. symbol containment on the left and the corresponding visual representation on the right. In the tree structure presentation, inner nodes define container symbols with an associated layout manager from the VIVID layout library. Leafs are given either as container symbols or as primitive symbols, e.g., rectangles, lines, labels, buttons. In Figure 2 the example specifies $G$ as a symbol with two embedded containers with a graphical primitive for a *GroupId* on top of the innermost container.

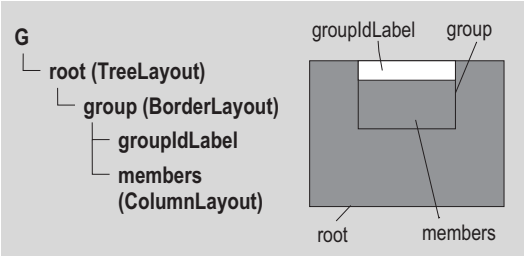


Figure 2: Hierarchical and Topological Structure of Symbol Classes

Once the symbol classes are defined we associate each symbol class to a data source which can be given by an SQL statement. The example in Table 1, for instance, defines that *Group*s returned by the SQL statements are associated to symbol class $G$. When executing the table one symbol is instantiated for each object returned by the SQL statement. Additional specification gives the mapping between symbol properties and database attributes. That means, for instance, that attribute value "male" of attribute type *sex* is displayed as *blue* and "female" as *red*.

| Symbol Class | SQL Statement |
|---|---|
| $G$ | $SELECT * FROM\ Group$ |

Table 1: Association of Symbol Classes to Sources

The next step defines the composition of the layout by the specification of constraints. which basically refer to conditional embeddings of symbol containers. An example is given in Table 2. The first row, for instance, defines the case when the source attribute *SuperGroup* is not defined, i.e., a group has no supergroup. In that case the *Child* symbol of class $G$ is placed in the container *scrolledArea* of *Parent* $V$. When parent class equals child class then both are distinguished by index

| Parent | Child | Condition | Symbol Container |
|---|---|---|---|
| $V$ | $G$ | $G \rightarrow SuperGroup == NULL$ | $V \rightarrow root \rightarrow scrolledArea$ |
| $G$ | $G$ | $G[1] \rightarrow GroupId == G[2] \rightarrow SuperGroup$ | $G[1] \rightarrow root$ |

Table 2: Example for Layout Constraints

1 and 2. The exact placement of symbols is determined by the layout manager of $G \rightarrow root$ which was defined in the tree structure (cf. Figure 2).

In a final step, the visual interaction is defined by a set of operations which are assigned to user interface events for each symbol like drag & drop. For drag & drop the interaction is specified through a table. The table defines the database operation which is executed when dragging a specific object type and dropping in on top of a second object type.

# 3 Implementation

The VIVID framework is implemented as a set of specification tools and libraries which are controlled by the user through a view management tool. The editor for symbol specification is derived from a GUI builder extended by graphical object primitives like boxes, lines, circles. The constraints editor for layout composition and the interaction editor for drag&drop specifications are implemented as form-based wizards with source code generation. VIVID is implemented in Java with JDK 1.2, JDBC, the Java Foundation Classes (Swing), and the Drag & Drop API which is released with the JDK 1.2.

# References


[1] S.K. Chang, et.al., A Visual Language Compiler, IEEE Transactions on Software Engineering, vol. 15, no. 5, pp. 506-525, May 1989.

[2] A.L. Chow, et.al., Topological Composition of Systems: Specifications for Lexical Elements of Visual Languages, Proceedings of the IEEE Symp. on Visual Languages, Kobe, Japan, Oct. 1991.

[3] G. Costagliola, et.al., Automatic generation of visual programming environments, IEEE Computer, vol 28, no. 3, pp. 56-66, March 1995.

[4] H.R. Hartson, et.al., UAN: A User-Oriented Representation for Direct Manipulation Interface Designs, ACM Transactions on Information Systems, vol.8, no. 3, pp. 181-203, July 1990.

[5] J. Rekers, A. Schuerr. A Graph Based Framework for the Implementation of Visual Environments. In Proc. 1996 IEEE Symp. on Visual Languages, Boulder, Colorado, Sept. 1996.

[6] G. Szwillus, Object Oriented Dialogue Specification with ODSN, Proceedings of Software-Ergonomie '97, (In German), Teubner, Stuttgart, 1997.