Tool Support for Design Methods - The DOCASE Object-Oriented Approach[1]

P. Rösch, W. Gerteis[1], M. Mühlhäuser, T. Leidig
University of Kaiserslautern, Informatics Dept., Telematics Group
Erwin-Schroedinger-Str., W-6750 Kaiserslautern, GY. Email: peter@informatik.uni-kl.de
[1] University of Karlsruhe, Kaiserstr. 12, W-7500 Karlsruhe Email: gerteis@ira.uka.de

1. Motivation

Sophisticated design tools should 'understand' the methods underlying the design process, not only in terms of syntax but in terms of the semantics of design 'elements' and 'steps' and their relations, etc. Despite the need for such *'comprehensive method support'*, at present only few design tools support much more than the syntax of design elements; most graphical tools mingle design methods and graphical representations into one tool, and little emphasis has been put on the use of tool-building-tools.

These problems are becoming aggravate as we move towards large distributed applications. In the *DOCASE* [3] project, coping with object-oriented development of this kind of applications, it has been found essential to

- separate design methods and corresponding 'graphical design methods' (the human interface part)
- provide a formal basis for comprehensive *method support* (as above) for both kinds of methods
- rely on tool-building-tools for reduced development *and* extension efforts

We present a model for *design methods* and *graphical design methods* (2) and corresponding tools as part of the *DOCASE* architecture (3). Based on issues of distribution (4) we present the design and implementation of tool-building-tools and their customization (5). In the remainder please note that our understanding of plain 'design methods' excludes graphical aspects.

2. Set-Theoretic Model of Design Methods and Graphical Design Methods

The definition of the model for design methods is based on the approach of [4], which is an extension of Pott's Model [6]. To be able to identify the elements that differentiate a design method from a graphical design method, the following set-theoretic model is described.

Design Artifacts represent *elements* of the 'system under development' (SUD), *relations* between those, or other characteristics of the SUD. A design artifact is considered as a single item, to be used in the design process. It may represent an aggregation of other design artifacts. Design artifacts are composed of **Design Elements**, whereby a design artifact can be viewed as a set of instances of design elements.

Design Steps: design artifacts can be modified, created or deleted by the executing design steps. Design steps can be viewed as operations, that can be performed on certain types of design artifacts (design elements).

Design Rules specify pre-defined semantic constraints between design elements. These rules are oriented towards checking the results or intermediate results of the design process. They do not specify relations between design steps.

Design Procedure: supports the transformation of artifacts of a problem domain into the solution domain. Within a design procedure the control flow, the process of design reasoning and design decisioning is specified. The control flow is described by a sequence of **issues**. **Positions** are alter-

[1] presented at ECOOP Workshop ''CASE for Object-Oriented Methods'', Utrecht, NL. June 1992.

native solutions to issues that are supported or objected to by **arguments**. A selected position results in certain design steps that modify artifacts and raise new issues to be dealt with.

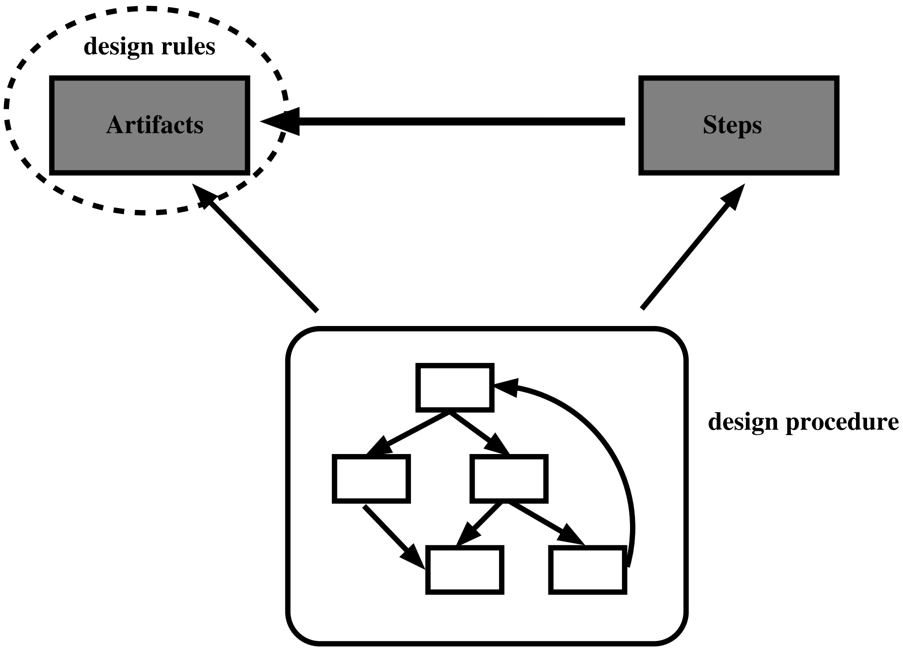

Figure 1   A simplified picture of a design method

For a complete and detailed description of the design method and design procedure see [4].

The design method can now be defined as a set of design elements {**DE**}, design rules {**DR**}, design steps {**DS**} and the design procedure **DP**:

Design Method :== ({DE},{DR},{DS}, DP)

In the context of this article we want to concentrate on the differences between a design method itself and a design method based on graphics. The design procedure includes issues, arguments, positions and the relations between them. Whereby artifacts and steps define *syntactic aspects* of a design method, the design procedure describes the *semantic aspects*. Combining the later aspects into one box, a simpler picture of a design method can be drawn (Figure 1).

Design Method Assistant: In the literature, graphical notations and methods are mostly represented as different 'diagrams' or 'charts' (with an underlying design procedure). The relation between design method, rules and graphical representation is given rather intuitively. Few authors distinguish a design method and graphical notations. The *DOCASE* Design Method Assistant *DODMA*, based on tool-building-tool *DMA*, supports the *DOCASE* design method syntactically and semantically (cf. 5.1).

Graphical Artifact: analogous to design artifacts, *graphical artifacts* are considered as single items. They are visually represented and can be encapsulations (e.g. charts) or *atomic artifacts* (icons, arcs). Like design artifacts, graphical artifacts are composed of **Graphical Elements**.

Graphical Notations denote bijective functions which map design artifacts to graphical artifacts :

$$\textbf{\textit{Graphical Notation gDE :== \{design artifacts\}}} \xleftrightarrow{\textbf{\textit{gDE}}} \textbf{\textit{\{graphical artifacts\}}}$$

The notion *gDE* points out that graphical notations play the same role for graphical design methods as sets of design elements *DE* do for plain design techniques.

Graphical Rules: Sophisticated graphical interaction tools account for 'human interface' aspects which are *not* related to application semantics, such as event-type specific interaction-regions, window placement, graph layout, highlighting of input focus. This leads to *graphical rules* and constraints about the graphical system itself. In some cases graphical rules correspond to design rules (e.g. 'a class hierarchy graph should have as few crossings as possible' is both a graphical and a pure design rule in object-oriented design), but the mapping needs to to be complete. The mapping of those design rules to corresponding graphical rules is denoted as mapping-function ***gDR***. While not all design rules can usually be mapped to graphical design rules, *gDR* should be as complete as possible and a 'design method assistant' should observe the residual rules.

$$gDR :== \{design\ rules\} \xleftrightarrow{gDR} \{graphical\ rules\}$$

Graphical Steps: like design steps, graphical steps can be viewed as operations, that can be performed on graphical artifacts.

$$gDS :== \{design\ steps\} \xleftrightarrow{gDS} \{graphical\ steps\}$$

Graphical Design Method: For an elaborate *graphical design method* it is important to reflect as much as possible of the design method. We define a

Graphical Design Method :== ({GE}, {GR}, {GS}, GDP) where

- *{GE}*, *{GR}, {GS}* are graphical equivalents to design elements, design rules and design steps.
- *GDP* is the Graphical Design Procedure defined as Design Procedure and mapping functions
GDP :== (DP, gDA, gDR, gDS)

It is important to note, that in our understanding, only the syntactic aspects of a design method are mapped to the corresponding graphical design method. If the design procedure and the graphical design procedure cannot be transformed by mapping the corresponding syntactic components (artifacts, steps, rules), then the methods themselves are different. The mapping between these components is easy only for artifacts. The mapping functions *gDR* and *gDS* need not to be bijective. In our approach we do not yet specify a formalism to describe these functions. The current work tends towards using graph grammars and graph transformation rules to describe the mapping functions. A graph grammar module will then provide the interface between design method assistant (corresponding to the design method) and graphical tool (corresponding to the graphical design method).

Graphical Design Tool: In order to build a *graphical tool* for a given design method, one has to

- create/define all graphical elements used,
- map the individual design techniques and the encompassing design steps and rules into the tool
- add specific graphical rules to both the individual techniques and the overall design steps and rules
- provide for integration with the residual software engineering environment.

The above formalism shows some of the lacks of common graphical design tools: they often consist of a set of graphical notations gDA_i for several design methods; typically few graphical design rules and graphical rules (gDR_i, GR_i) are implemented; the same holds for gDS. Design assistants hardly exist, many rules are only verbally described. The integration with a software engineering environment normally consists (at most) of a 'one way' code generation. The *DOCASE* Object-Oriented Design Editor (*DODE*) supports our specific graphical design method. It is based on tool-building-tool *ODE* (cf. 5.2), provides for rich semantics and tight environment integration.

3. The *DOCASE* Generic Design Tool Approach

We propose an architecture in which *design method support* and *graphical tool* are clearly separated and both based on tool-building-tools, cf. fig. 2. Customization of the tool-building-tools leads to specific tools for design method and graphical design method support, respectively, and to the integration functions for integration with the residual software engineering environment.

As has been pointed out, we call for sophistication of both methods and tools. As to the methods, we require ample semantical support, expressed in the sets DR_i and DS. As to the tools, the design rules DR_i and design steps and rules DS have to be mapped completely to either the design method assistant or the graphical rules (gDR_i gDS); additional human interaction support has to be given.

In addition, we propose the tool-building-tool approach for reduced construction effort and higher flexibility (in *DOCASE*, we customize our tools to even reflect specifics of the application domain). We also propose a tight integration between graphical design tool, design method assistant, and other software engineering tools, based on the compiler parse tree *(not* described in more detail below).

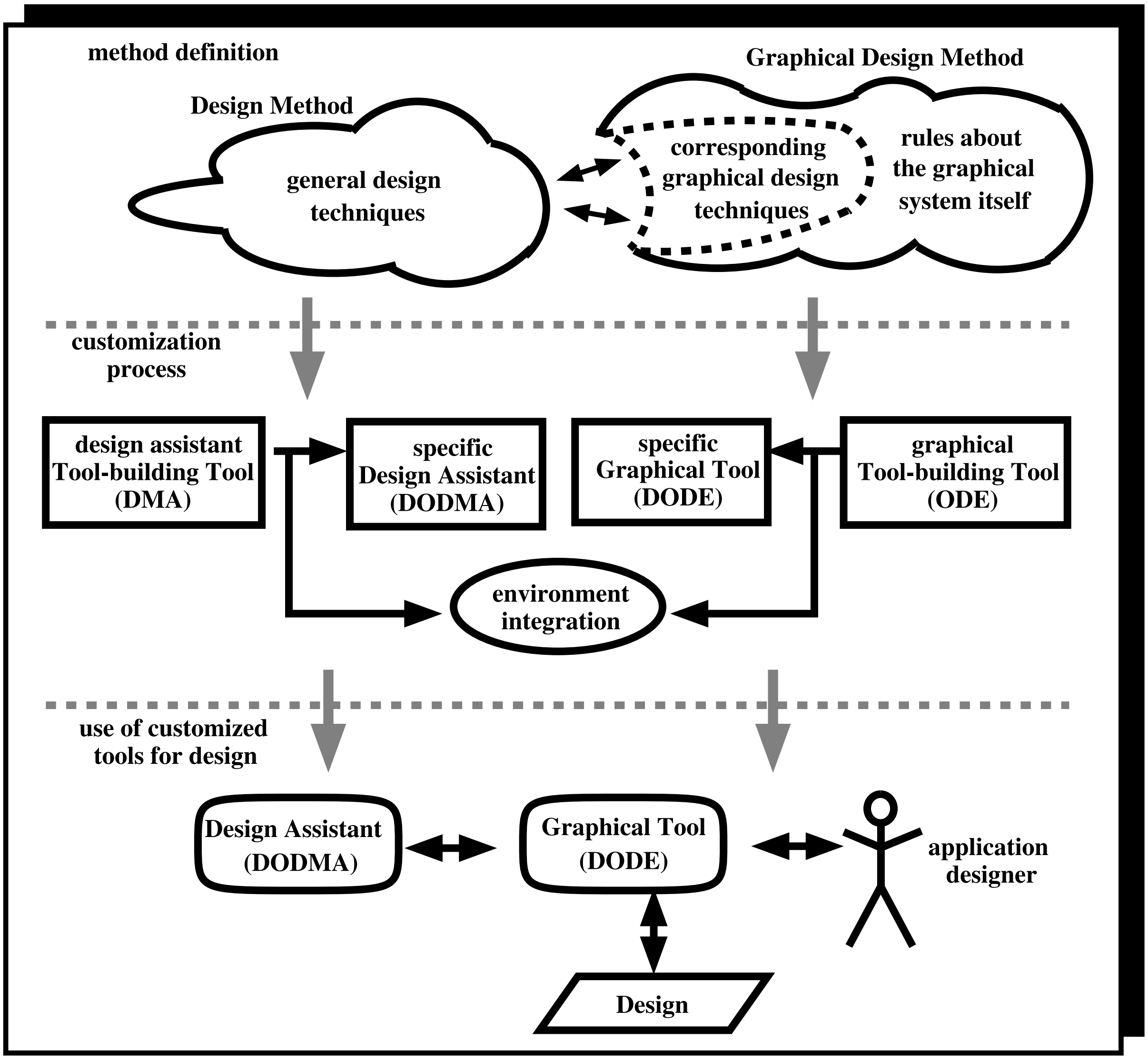

Figure 2: Development procedure for a graphical design tool

4. Issues in 'Distribution and Objects' and their Reflection in *DOCASE*

Modularization and structuring concepts are among the most important in a design method. With respect to distribution, units of modularization are expected to be units of distribution, too. Here, existing distributed o-o systems prove objects to have ideal properties (abstraction, unit independence, arbitrary granularity). However, many o-o design methods, e.g. [1,2], follow the rule 'small is beautiful'. As a consequence, they offer a small set of design elements only. This is adequate for the design of small and medium scale applications; for large distributed applications however, a larger set of 'distinguished design elements' is required, as mediators between real-world and software modeling. The distinction of these elements must not fade away during the various transformations of the SUD, even when the target is a 'conventional' o-o programming language where 'everything is an object'. Instead, it must be visible to all tools during the whole lifecycle (e.g. for the animation of 'thousands of objects'). Therefore, the additional semantics expressed by distinguished design elements have to be accounted for by the design method.

With respect to distribution, we found it crucial to distinguish e.g., *configured, generated,* and *relation* objects. *Configured* objects specify parts of the application configuration (intuitively, 'long-'first-order' objects). *Generated objects* are created (intuitively, 'in masses') and manipulated by configured objects. *Relation objects* specify complex semantic relations between configured objects, e.g. cooperation or interaction patterns, object migration constraints etc. (all of which are usually mingled into 'method call' relations). We also added to the (semantically poor) 'aggregation' concept that of *subsystems* which denote a set of configured objects, their relations and management.

A 'good' design method must not only provide 'creation and naming', but support the developer in choosing the right 'distinguished design elements', object structure, etc. for the SUD elements. The same holds for references, which are all treated equal in o-o design despite their different semantics. This knowledge is again helpful throughout the software lifecycle.

Since structural (coarse-grained) parallelism is a 'natural part' of distributed applications (modeling parallel real-world activities), a design method has to offer adequate design elements and strategies for these problems, too.

DOCASE offers the design language *DODL* [3] which provides a large amount of 'distinguished elements', reflecting the above distribution issues, and an elaborate corresponding design method.

5. *DOCASE* Tool-Building-Tools and Their Customization

5.1 DMA (Design Method Assistant) and its customization into *DODMA*

DMA is based on Potts' generic model of design methods describing the relations between design elements, design steps and rules, and design reasoning. Design reasoning is described in an issue - argument - position cycle: design elements are reviewed because of a certain issue (questions about the design element/s); positions respond to these questions; depending on a selected position, one or more steps have to be executed; arguments support or object to positions. For formal specification of design methods, a *design method description language* is developed, comprising three major parts:

- the element representation definition used to define the design elements
- the issue specification, describing the issue/argument/position components. The sequence of issues describes the major strategy used to elaborate a SUD; arguments may be based on the design elements (syntax or semantics). A good design method has to provide intensive support here. Additional arguments may be specific to the application domain (subject to domain customization).
- the user interface specification, mapping the human interaction of the design method assistant to the graphical design tool, either according to the mapping described in (3) or using the graphical tool as i/o interface (for rules not supported within the graphical tool).

The customization of *DMA* into *DODMA* represents work which is still in progress. E.g., we help the user in identifying the correct *DODL* 'distinguished design elements' (e.g., finding 'important' real world parts as candidates for subsystems and configured objects), support various kinds of built-in and semantic relations (cf. above), and organize parallel global control flows on the 'basic' objects.

5.2 ODE (Object Design Editor) and its customization into *DODE*

ODE is an extensible, customizable tool-building tool intended for generation of graphical editors and graphical design tools. It includes an interpreter for its built-in o-o (Lisp-like) customization language. This language was designed to be small, easy-to-learn, but powerful. Its 'object' concept, also very simple, can be customized to reflect any object model. Its 'relation type' is supported by a number of relation functions like 'build transitive closure' and 'traverse graph'. The language also serves for adding method-specific types and relations, functions and methods, and even language extensions.

ODE enforces an explicit separation between models (e.g., design elements handled in a design technique/method) and *view*s (e.g., graphical design techniques), similar to the *model/view/controller* paradigm of Smalltalk-80. Thereby, *ODE* supports a *constraint evaluation* mechanism based on *active values*, providing, e.g., means for handling global dependencies. In terms of the formalism described in (2), this means that *ODE* is well suited for modeling different graphical notations, gDA_i, which provide different aspects of the same set of design elements. It also means that *ODE* is an ideal means for describing the different rule/constraint sets gDR, GR, gDS.

ODE supports simultaneous views and has several built-in (customizable) views such as a Graph-View supporting several sophisticated automatic graph layout mechanisms (which are designed 'snap-in', i.e. subject to enhancement/replacement). Based upon *OSF/Motif*, *ODE* offers all *OSF/Motif* widgets as view types.

The customization of *ODE* into *DODE* required the representation of, e.g. the above 'distinguished design elements' and according graphical artifacts, the implementation of the method-specific rule/constraint-sets (for which the *ODE* customization languages proved to be very efficient), and of the 'plain graphical' rules. In reality, *ODE* holds only 'proxies' of the design elements, mapped to the 'shared' design elements of the integration facility via *ODE*'s rewriting rule system.

6. Conclusion

We presented a model and architecture for extensive *design support*, a very urgent problem in the development of large distributed applications. We described the corresponding tools and their application to a design/implementation language and method.

Major requirements stated where: separation of method and graphics; extensive method support in the tools; methods based on 'distinguished object categories' instead of a 'universal object notion'; use of tool-building-tools; tight integration with residual tools.

7. Bibliography

[1] G. Booch: Object Oriented Design with Applications. The Benjamin/Cummings Publishing Comp., 1990

[2] P. Coad, E. Yourdon: Object-Oriented Analysis. Yourdon Press 1990, Prentice Hall, Englewood Cliffs, NJ

[3] W. Gerteis, Ch. Zeidler, L. Heuser, M. Mühlhäuser: DOCASE: A Development Environment and a Design Language for Distributed Object-Oriented Applications. Proc. TOOLS Pacific '90, Sydney, Australia, Nov. 1990.

[4] W. Gerteis: An Approach Towards Guided Interactive Design. Proc. 2nd GLCSC, Kalamazoo, Mi, Oct 1991.

[5] M. Page-Jones et al.: Modeling Object-Oriented Systems: The Uniform Object Notation. Computer Language, Oct. 1990.

[6] C. Potts: A Generic Model for Representing Design Methods. Proc. 11th Int. Conf. on SW Eng., May 1989