JKogge: a Component-Based Approach for Tools in the Internet

Jürgen Ebert, Roger Süttenbach, Ingar Uhe

University of Koblenz-Landau – Institute for Software Technology

Abstract

The huge success of the Internet encourages existing systems to evolve by using this technology. This paper presents a component-based architecture for a system which is especially designed for tools working in the Internet.

Keywords: software architecture, component, CASE, software engineering environments, Internet.

1 Component-Based Architecture

Due to the wide spread of the Internet and one of its standards – Java – it is possible to develop systems with unusual but very flexible architectures. In this paper we would like to describe an architecture which is based on *loadable and cooperative components*. The background of this architecture was the development of a successor for KOGGE [ESU96], a meta-CASE system [E97] which allows the generating of tools for visual languages. The successor of KOGGE is called JKogge, showing that the implementation is made in Java. The project was inspired and was able to profit from the experience gained in building CASE tools with the KOGGE meta-CASE system. We will sketch some of these experiences and their influence to the concepts of JKogge at the end of this paper. And without denying the CASE background of JKogge we think that these concepts can also be used for other applications working in the Internet.

Although the request for component technology is very old [M68], although the idea of modularization is well-known and established (e.g. in Eiffel, ADA, Oberon) and although the existence of component systems (e.g. DCOM, CORBA, JavaBeans) is reality, there is still a lack of a clear definition of components. Many definitions, for example in Booch [B87] or Jacobson [J93], only demand the property of modularity. Industrial standards like DCOM declare any piece of software a component which implements their interfaces. Others like Orfali et al. [OHE96] and Sametinger [Sa97] add the request for a defined environment and standard interfaces for inter-operability. A restrictive definition is made by Szyperski [S97] based on the results of an ECOOP workshop. He states that a component is a unit which can be deployed independently, which is used by third-parties, and which has no persistent state.

The components used in our system have some characteristics which distinguish them from other approaches using component technology. We will state our definition and use of *components* in **Section 3**. The other main parts of the architecture, the *base system* and the *documents*, are described in **Section 2**. An example for concrete components is shown in **Section 4**. Finally, **Section 5** describes the current state of the CASE system and gives an outlook to its further development.

2 JKogge System

The JKogge system consists of three logical elements: *documents*, the *base system*, and *components* called JKogge-plug-ins. Figure 1 shows a schematic view of the three elements.

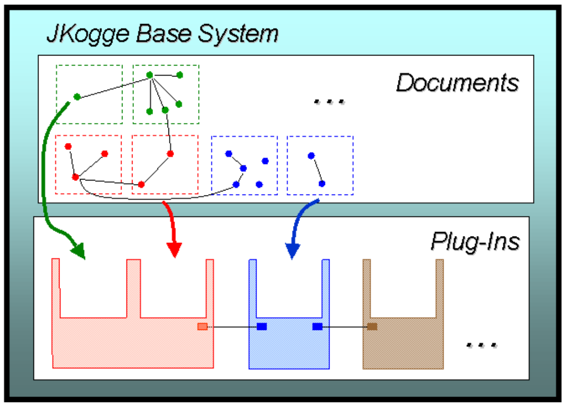


Figure 1: JKogge system, schematic view

Documents

In principle, a document is any identifiable unit of information with clear boundaries and a certain meaning. Because of the background of JKogge, the focus is on a special kind of document: CASE documents. CASE documents are produced in the software development process, examples are class diagrams, state transition diagrams but also source code. In JKogge, graphs ([EWD+96]) are used as internal data structures for these documents because this makes it possible to integrate them into a homogeneous format. Graphs offer an efficient way to store and retrieve information and they are capable to realize distributed documents. Additionally they provide possibilities to prove the correctness of the documents which is very important for CASE documents.

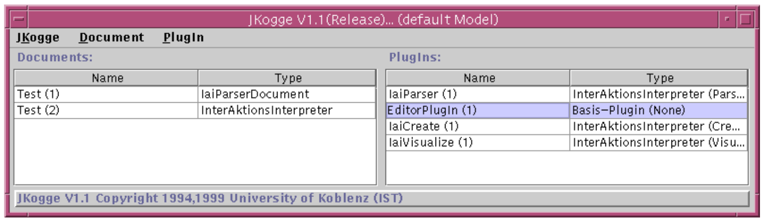
Base System

The base system serves as a *loader* for components and documents. It supports access to Internet addresses, so any document or component can be loaded if it is accessible via an URL (Uniform Resource Locator). But it also supports access to the local file system. The base system does not realize any functionality of a tool, it is only responsible for loading and managing components and documents.

As mentioned before the documents are represented as graphs. Text documents and images are used in addition to this. The base system loads documents on demand. I.e. only if a component explicitly requests a document, this document will be loaded.

The second major task of the base system is to serve as a *mediator* (see [GHJ+95]) between the components. When a component is loaded it automatically gets connected to the base system. Thus, the component is able to get a connection to any other component by the base system without knowing specific details about its location. The connection between different components is used to exchange messages, called JKogge-messages (see section 3.2).

Figure 2 shows a screenshot of the base system: Four components (in the figure called plug-ins) and two documents have been loaded. The next action could be to start one of the components.



Documents:

Name	Type
Test (1)	IaiParserDocument
Test (2)	InterAktionsInterpreter

PlugIns:

Name	Type
IaiParser (1)	InterAktionsInterpreter (Pars...
EditorPlugIn (1)	Basis–Plugin (None)
IaiCreate (1)	InterAktionsInterpreter (Cre...
IaiVisualize (1)	InterAktionsInterpreter (Visu...

JKogge V1.1 Copyright 1994,1999 University of Koblenz (IST)

Figure 2: Base system with loaded documents and plug-ins, screenshot

In addition to its loading and management tasks the base system deals with user information, including authorization, and provides services for customizing the user's environment.

The JKogge base system and the components are implemented in Java. The graphs used throughout the system are stored and manipulated with a derivative (*JGraLab*) of the *GraLab* [DW98] software.

3 Components

Components are realized by *JKogge-plug-ins*. The use and the meaning of the word plug-in in this context is similar to the use of this term in other software systems, like in Eudora® or Netscape®. The latter, for example, defines plug-ins as: "... software programs that extend the capabilities of Netscape in a specific way". Following this definition a JKogge-plug-in extends the capabilities of the JKogge base system.

But there are same additional aspects which distinguish JKogge-plug-ins from these pure plug-ins. The goals of JKogge-plug-ins are

- to run on every computer platform connected with the Internet
- to locate and to access them via URLs
- to load and to instantiate them on demand
- to run independently from other plug-ins
- to communicate with other plug-ins by a determined interface

These goals make it appropriate to choose an own architecture and do not use an existent component architecture like DCOM or CORBA or realize plug-ins as DLLs. And there are some additional reasons (e.g. the costs and the complexity of such an architecture) which lead to this decision.

So, although the term JKogge-plug-in indicates that we are dealing with pure plug-ins we see our JKogge-plug-ins as components which are built for specific document types. A JKogge-plug-in does *process* documents of a certain document type. In principle, this *processing* is arbitrary and up to the component in question. Thus, JKogge-plug-ins meet Szyperski's definition with the additional demand that there has to be a base system loading the plug-ins. Typical examples of JKogge-plug-ins[1] are viewers, editors or interpreters of documents.

[1]In the following we will use the term plug-in instead of JKogge-plug-in.

3.1 Structure

In principle a plug-in is a unit with its own thread of control which is able to process documents. Figure 3 gives a schematic description of a plug-in.

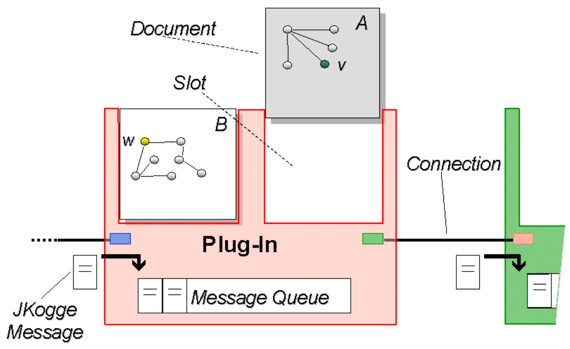


Figure 3: Structure and communication of plug-ins

The number and the types of documents, which a plug-in can process, is defined by its slots. A *slot* is a named and typed field into which a document can be inserted. The type of the slot describes the type of the document which can be contained. Documents can be added to or removed from the slots at run-time. Each plug-in has its own message queue which can take in JKogge-messages. Plug-ins can have *connections* to other plug-ins. I.e. there exist references from one plug-in to other plug-ins. These references are created on demand, i.e. the plug-ins request particular plug-ins from the base system. Plug-ins all share a common interface and are packed into Java archive files using some of the bean features. The base system is able to load `jar` files[2] and unpack them. The size of the plug-ins and the fact that only the plug-ins needed are loaded makes them especially suited for the use in the Internet.

3.2 Communication

The communication among the plug-ins in the base system is an asynchronous message-passing. There are message objects – called *JKogge-messages* – which are exchanged between plug-ins. Some messages are requests issued to one plug-in by another, other messages deliver data or notifications.

[2]To illustrate the size of the files some statistics:
- base system: appr. 70 classes/interfaces and 90 kbyte
- plug-ins: appr. 2-25 classes and 5-70 kbyte
- JGraLab: 100 classes/interfaces and 80 kbyte

A JKogge-message is an object sent from one plug-in to another. It contains information about the sender, the specific command to be executed, a document and specific vertices in the document. The messages are processed in the order in which they are received. At one point of time a plug-in can only process one message. Because each plug-in has its own thread of control, other messages can be put in a *message queue* – concurrently – by other plug-ins but processing is delayed until the processing of the message currently worked on is finished.

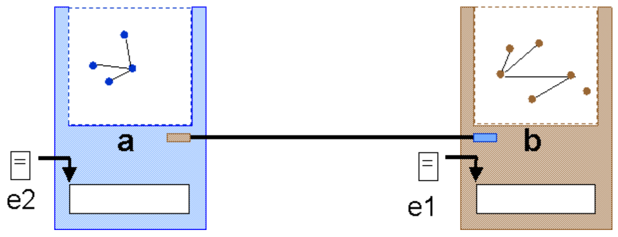


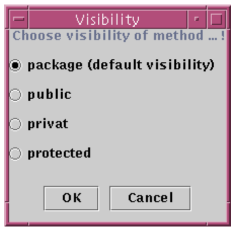
Figure 4: Communication of plug-ins

Figure 4 shows the communication between plug-in **a** and **b**. **b** receives a JKogge-message **e1** which contains the request for the job **b** has to carry out. **b** generates a return message **e2** which is sent to plug-in **a**.

4 Example

This section gives an example of how some plug-ins work together and how they communicate with each other. The plug-ins presented take care of the visualization of the windows and dialogs used in a concrete tool.

A dialog used in a CASE tool could look like this:


Visibility

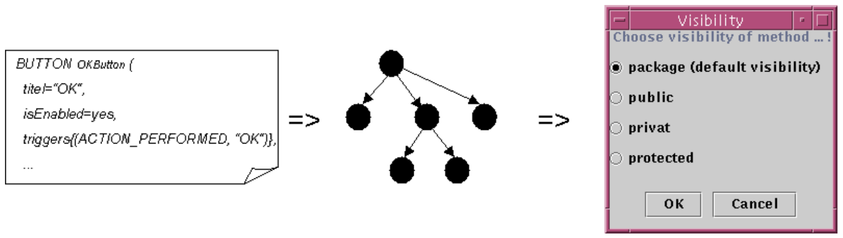
Choose visibility of method ...

[✓ package (default visibility)] public privat protected

Figure 5: Dialog used in a CASE tool

There are JKogge plug-ins which allow to specify the windows and dialogs needed and to create their visualization on screen at run-time. This task is realized by three plug-ins: a *parser-*, a *create-*, and a *visualize-plug-in*.

At present the dialogs needed are specified by a simple textual language. The description is parsed into a graph which serves as a template for the windows and dialogs which can be used in the tool (see Figure 6).



```
BUTTON OKButton (
  titel="OK",
  isEnabled=yes,
  triggers{(ACTION_PERFORMED, "OK")},
  ...
```

=>
=>
Visibility
Choose visibility of method ... !
package (default visibility)
public
privat
protected
OK
Cancel

Figure 6: Specification of windows and dialogs

Parser-Plug-In

As stated above windows are described by a textual language. The lines below are part of the description of the dialog shown in Figure 5.

```
FRAME Frame ( title="Visibility",
resizable=yes,
isEnabled=yes,
contentIsPlacedBy Border );

LABEL Label ( title="Choose visibility of method ... ",
alignment=center,
isChildOf Frame,
isEnabled=yes,
placedByBorderLayout Border ( position=North ) );

FLOWLAYOUT Flow ( alignment=center,
...
```

The result of the parsing process is a graph of dialog templates.

Create-Plug-In

These template graphs are instantiated by the create plug-in which means that it generates a second graph which can be customized and then be visualized. An example for such a customizing process could be to fill a text field with a certain string before the corresponding window is visualized. There is a distinction between the template and the visualization graph to make it possible to have one template for a dialog and various instances of it at the same time which only differ in the labels of the buttons for example.

Visualize-Plug-In

The concrete dialogs are described in the visualization graph. This visualization graph can be visualized by the visualize plug-in. Furthermore the

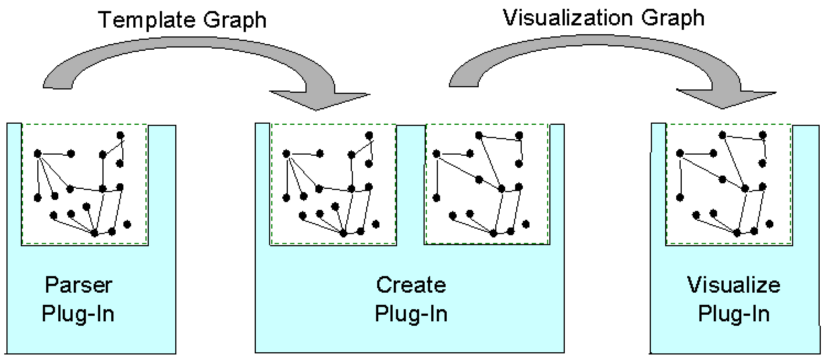


Figure 7: Plug-ins for the visualization of dialogs and windows

visualize plug-in can hide and destroy dialogs or windows and update the visualization according to changed values in the visualization graph.

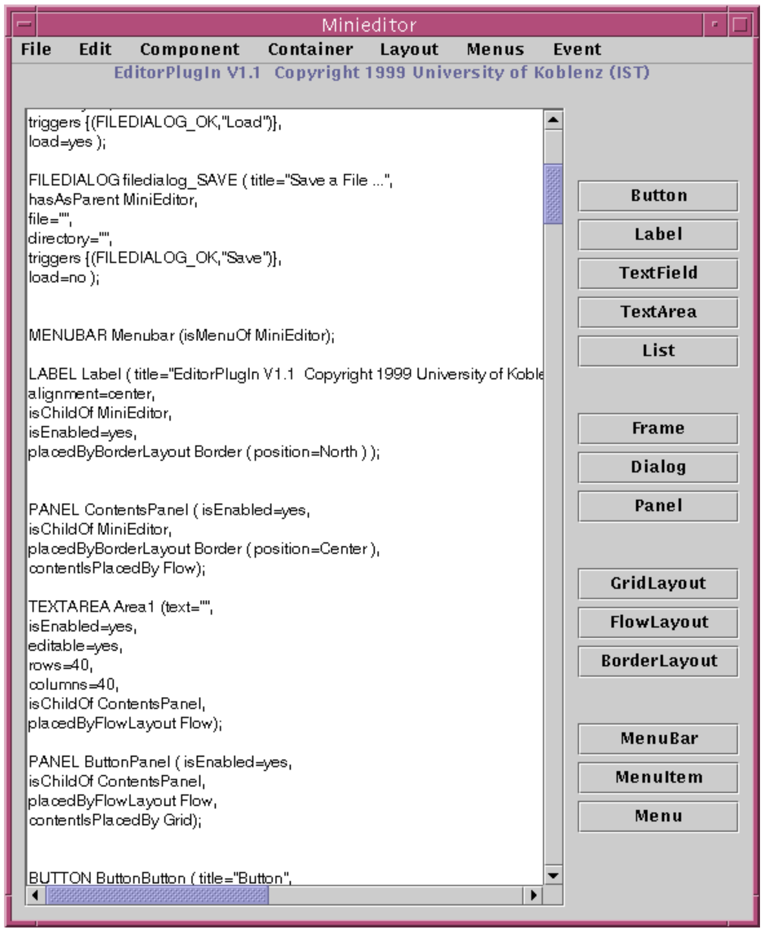
JKogge-Messages

As described above some plug-ins (like the visualize plug-in) can fulfill various tasks. This implies that there has to be a means of communication between plug-ins. They communicate via JKogge-messages as described in Section 3.2. The visualize plug-in accepts the commands (which are part of the JKogge-message): *show, hide, destroy* and *update*. But what is even more important is the communication back from dialogs and windows visualized on screen to the visualize plug-in. The visualize plug-in works as listener for the events emerging which means that it receives the events from the Java runtime system. It passes them on to the other plug-ins of the system. For example, if the user clicks the `Ok-button` the rest of the system has to be notified to be able to react to the user interaction. In JKogge the specifier of a tool interface decides which events can be delivered. This is done like in the following example:

```
BUTTON OKButton ( title="Ok",
isEnabled=yes,
isChildOf ButtonPanel,
triggers {(ACTION_PERFORMED,"Ok")},
placedByFlowLayout Flow);
```

In this example the visualize-plug in creates a JKogge-message which tells the component which uses the dialog that the specified `Ok-Selected` event occurred. The reaction to this is up to the plug-in which receives the JKogge-message.

The screenshot in Figure 8 shows a small tool realized using the three plug-ins described above (parser, create and visualize plug-in) and an additional one (editor plug-in). The plug-ins used can be seen in the screenshot in figure 2. This small tool simplifies the creation of the dialog and window specifications.



```
triggers {(FILEDIALOG_OK,"Load")},
load=yes );

FILEDIALOG filedialog_SAVE ( title="Save a File ...",
hasAsParent MiniEditor,
file="",
directory="",
triggers {(FILEDIALOG_OK,"Save")},
load=no );


MENUBAR Menubar (isMenuOf MiniEditor);

LABEL Label ( title="EditorPlugIn V1.1  Copyright 1999 University of Koble
alignment=center,
isChildOf MiniEditor,
isEnabled=yes,
placedByBorderLayout Border ( position=North ) );


PANEL ContentsPanel ( isEnabled=yes,
isChildOf MiniEditor,
placedByBorderLayout Border ( position=Center ),
contentIsPlacedBy Flow);

TEXTAREA Area1 (text="",
isEnabled=yes,
editable=yes,
rows=40,
columns=40,
isChildOf ContentsPanel,
placedByFlowLayout Flow);

PANEL ButtonPanel ( isEnabled=yes,
isChildOf ContentsPanel,
placedByFlowLayout Flow,
contentIsPlacedBy Grid);


BUTTON ButtonButton ( title="Button",
```

Figure 8: Editor Plug-In

It provides editing capabilities for the textual specifications of the windows. There is a menu item for each textual element which can be used. Some of them are also available as buttons on the right hand side. The editor creates textual templates, the only thing the user has to change are the names. Additionally the small editor provides the functionality to load and store the specifications.

5 Outlook

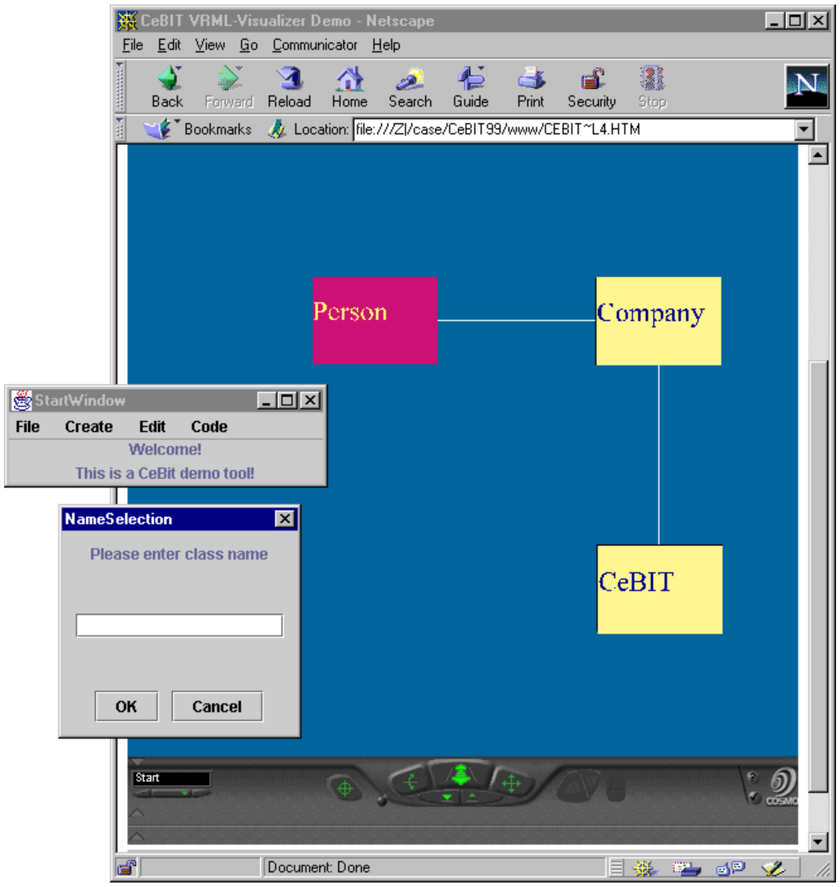
The previous sections showed that a great variety of tools can be realized with our component-oriented architecture. As mentioned in the introduction the architecture of JKogge is influenced by the experiences of former projects (e.g. [KU96], [PL97], [P98]). As a result the main demands on a system are:

- adaptability – the system has to adapt in an easy and flexible way to the individual situation of a project
- re-usage – in contrast to the individual situation there are always parts of the system which are needed in all projects (maybe slightly different)
- non-monolithic – only those parts of the system should be delivered which are really needed by the project
- platform-independence – there is more and more mixture of computer platforms and the system should be able to run on each of them
- networking – it gets more and more important that the system is able to communicate to other computers and systems via a network
- open architecture – the system should be able to communicate or it should be extendable with other programs

Of course, the current implementation does not cover all these demands. For example, at the moment we are able to communicate with other programs by wrapping plug-ins. But this solution is quite expensive because we have to build a wrapping plug-in for every program. Another open point concerns such problems as configuration management or versioning which is needed for adaptability and re-usage. We here think of special configuration and versioning plug-ins which manage all the plug-ins needed for a certain project. So the strategy is to build complex plug-ins maybe out of simple plug-ins which control other plug-ins.

Our goal is now to design a number of plug-ins for various contexts and diagrams, respectively. This will include visual editors, parsers, analyzers and simulators, graph browsers, viewers for many kinds of text and HTML text generators. Figure 9 shows a screenshot of a prototype which was presented at CeBIT'99 using VRML for the visualization of – simplified – class diagrams. In this prototype five plug-ins are working together to build this tool. Four of these plug-ins can be used without modification for other CASE tools, for example to support an editor for state charts.

Finally, we think that our architecture is a good base for developing systems which consist of a number of independent but cooperative components.


CeBIT VRML-Visualizer Demo - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: file:///Z|/case/CeBIT99/www/CEBIT~L4.HTM
Person
Company
CeBIT
StartWindow
File Create Edit Code
NameSelection
OK Cancel
Start
Document: Done

Welcome!
This is a CeBit demo tool!

Please enter class name

Figure 9: CeBIT-prototype supporting class diagrams

This is especially true for a CASE system like JKogge. The concept of a small base system which loads plug-ins that can be built to provide almost any functionality needed gives the system the flexibility to realize tools for a wide variety of purposes.

Acknowledgement: The authors thank Martin Schulze, Prof. Dr. Manfred Rosendahl, and all the students working on the projects KOGGE and JKogge.

References

[B87] Booch, G.; *Software Components with Ada: Structures, Tools, and Subsystems.* Redwood City: Cummings, 1994^2.

[DW98] Dahm, P.; Widmann, F.; *Das Graphenlabor.* Koblenz: Universität Koblenz-

Landau, Fachberichte Informatik 12/98, 1998.

[EWD+96] Ebert, Jürgen; Winter, Andreas; Dahm, Peter; Franzke, Angelika; Süttenbach, Roger; *Graph Based Modeling and Implementation with EER/GRAL.* In: B. Thalheim [Ed.]; 15th International Conference on Conceptual Modeling (ER'96), Lecture Notes In Computer Science, Proceedings, LNCS 1157. Berlin: Springer, 1996.

[ESU96] Ebert, J.; Süttenbach, R.; Uhe, I.; *Meta-CASE in Practice: A Case for KOGGE.* In: A. Olivé and J.A. Pastor [Eds.]; 9th International Conference, CAiSE '96, Lecture Notes In Computer Science, 1250. Berlin: Springer, 1997.

[E97] Ebert, J.; *MetaCASE: Generierung und Anpassung von CASE-Werkzeugen.* In: W. Gens (Hrsg.) 3. Fachkongress Smalltalk und Java in Industrie und Ausbildung (STJA'97) Technische Universitt Ilmenau, 1997, p. 191-196.

[GHJ+95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; *Design Patterns: Elements of Reusable Object-oriented Software.* Reading: Addison-Wesley, 1995.

[G98] Griffel, F.; *Componentware - Konzepte und Techniken eines Softwareparadiagmas.* Heidelberg: dpunkt.verlag, 1998.

[J93] Jacobson, I.; *Object-Oriented Software Engineering.* Reading: Addison-Wesley, 1993.

[KU96] Kölzer, A.; Uhe, I.; *Benutzerhandbuch für das KOGGE-Tool BONsai III.* Koblenz: Universität Koblenz-Landau, Projektbericht, 1996.

[M68] McIlroy, M.D.; *Mass Produced Software Components.* In: Naur. P and Randell, B. [Eds.]; NATO Conference on Software Engineering, Proccedings. Brüssel: NATO Science Committee, 1968.

[OHE96] Orfali, R.; Harkey, D; Edwards, J.; *The Essential Distributed Objects Survival Guide.* New York: Wiley & Sons, 1996.

[P98] Polock, D.; *Benutzerhandbuch für die KOGGE-Instanz FAKT.* Koblenz: Universität Koblenz-Landau, Projektbericht, 1998.

[PL97] Pühler, T.; Löcher, M.: *Entwicklung eines Softwareevaluationstools* Koblenz: Universität Koblenz-Landau, Studienarbeit, 1997.

[V97] (`http: http://www.vrml.org/Specifications/VRML97/`)

[Sa97] Sametinger, J.; *Software Engineering with Reusable Components.* Berlin: Springer, 1997.

[S97] Szyperski, C.; *Component Software – Beyond Object-Oriented Programming.* Reading: Addison-Wesley, 1997.