Systems Development with Java: Experiences from a Practical Project Course in Software Engineering*

Klaus Bergner
Technische Universität München,
Institut für Informatik,
D-80290 München, Germany
bergner@informatik.tu-muenchen.de

Franz Huber
Technische Universität München,
Institut für Informatik,
D-80290 München, Germany
huberf@informatik.tu-muenchen.de

Abstract

This paper describes our experiences in using the Java programming language in a student software engineering project. We focus on the suitability of Java for developing large-scale software systems in teams, and on the tools and techniques used for design and implementation. Furthermore, we comment on the significance of our experiences for future educational software engineering projects as well as for industrial projects.

1. Introduction

Developing large software systems in teams is a skill best learnt by practical experience. In order to make this experience available to our students at the informatics department of Technische Universität München, we have been giving practical project courses in software engineering for three years. Each of these courses comprised a complete software development project and was intended to deliver a usable software product to a customer at the end of the project. The time available for the students to realize the project was one semester (three months).

This paper summarizes the experiences gained in the 1996 summer term, during which we developed a CASE tool for distributed systems engineering with a team of 14 students. Aside from organizational issues we will put special emphasis on how the concepts of our implementation platform, the Java [7] programming language and its libraries, work together with standard object-oriented software engineering techniques.

We think that most of our experiences from this course are of interest not only for other university projects, but for industrial software development as well.

*This work was partly carried out within the Subproject A6 of the "Sonderforschungsbereich 342" and the Project SysLab, sponsored by the German Research Community (DFG) under the Leibniz program and by Siemens-Nixdorf.

2. The Project – Goals, Results, Future Work

2.1. The Goal: A Tool for the FOCUS Method

The goal of the project was to develop AUTOFOCUS, a working prototype of a CASE tool for distributed systems engineering, comprising graphical editors for the different views of a system. The description formalisms for these views are embedded in the semantical framework of FOCUS [5], a formal development method for distributed systems. FOCUS is based on a mathematical foundation using semantical concepts like stream processing functions [4] to describe the components of a system. It provides precise mechanisms for refining system specifications, enabling a developer to prove the correctness of development steps. The concept of AUTOFOCUS is described in detail in [11] and [10]. Its description formalisms, for which appropriate editors had to be implemented, are

- system structure diagrams (SSDs), which describe the static structure of a system by a network of interconnected components exchanging messages over channels,
- state transition diagrams (STDs, or automata) to describe the behaviour of system components, and
- extended event traces (EETs, basically a subset of Message Sequence Charts as standardized in ITU Z.120 [12]) used to specify exemplary system runs.

It should be obvious that a tool that complex can hardly be implemented entirely in an educational course during a period of three months. Therefore, it was the goal of the course to act as a starting point for the whole project and to implement the basic components of the tool.

2.2. Project Results

2.2.1. Tool Prototype

The main result of the project course is a working prototype of AUTOFOCUS. As can be seen from the architectural

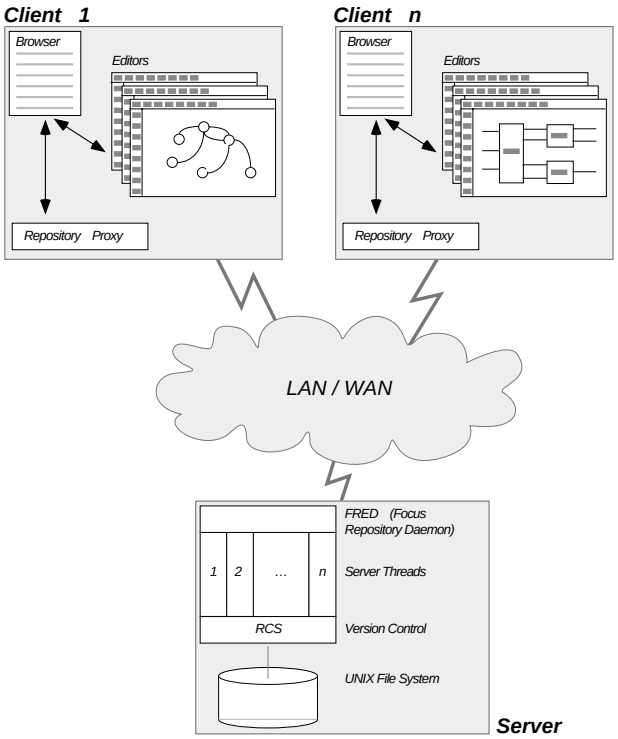

Figure 1. Architectural View of the Current AUTOFOCUS Implementation

view in figure 1, AUTOFOCUS is a typical client/server application with a central repository where all development documents are stored. An arbitrary number of clients can connect to the repository over a network using TCP/IP-based sockets. Within the server process, a dedicated thread is running for each client. The repository is currently document-based, mapping development documents to files on a UNIX file system. Access and version control is provided by the UNIX revision control system RCS [3].

The AUTOFOCUS client contains a project browser used to navigate through the development projects and documents in the repository, and three graphical editors necessary to edit the different diagram types. Both the clients and the server, which encapsulates the RCS system, are completely implemented in Java.

To provide an impression of the current status of AUTOFOCUS, a screen hard-copy of some parts of the client application is shown in figure 2.

2.2.2. Documentation

Besides the tool prototype, a set of documents has been produced in the project: the system specification document [20], the system design document [19], and class design documents (the latter suffered from the time pressure in the final stages of the project and must still be completed), as well as a user's guide for the AUTOFOCUS tool, from which an HTML online help was generated.

2.2.3. Statistics

The results just outlined were achieved by a group of 14 students within three months, with one and a half months used for implementation. The whole project thus spans 42 person months with implementation consuming 21 person months. The documents produced in the project comprise roughly 100 pages, not including the partially incomplete class design documents. The current size of the AUTOFOCUS prototype is around 32 kLoC (900 kByte) written entirely in Java.

2.3. Further Development

Based on the current status, we are planning a series of further development activities that will mostly be carried out in student projects.

Currently, AUTOFOCUS is re-designed, mainly because of two reasons: First, all user interface parts will be reworked to fit into the new version 1.1 API [24] of Java, and second, there is some amount of "runaway code" which has been implemented in the final stages of the project under heavy time pressure and is thus not very well structured and documented.

Although AUTOFOCUS can be used for real work in its current state, most of its potential as a tool based on formal foundations has not been realized yet. Extensions of AUTOFOCUS to be implemented are

- consistency checks for all documents in a development project,
- interfaces to external verification tools, like model checkers and theorem provers,
- simulation, and
- code generation.

While the consistency checks are based on the well-defined abstract syntax and context conditions of the AUTOFOCUS description techniques, the latter three extensions build upon the formal semantics defined for them.

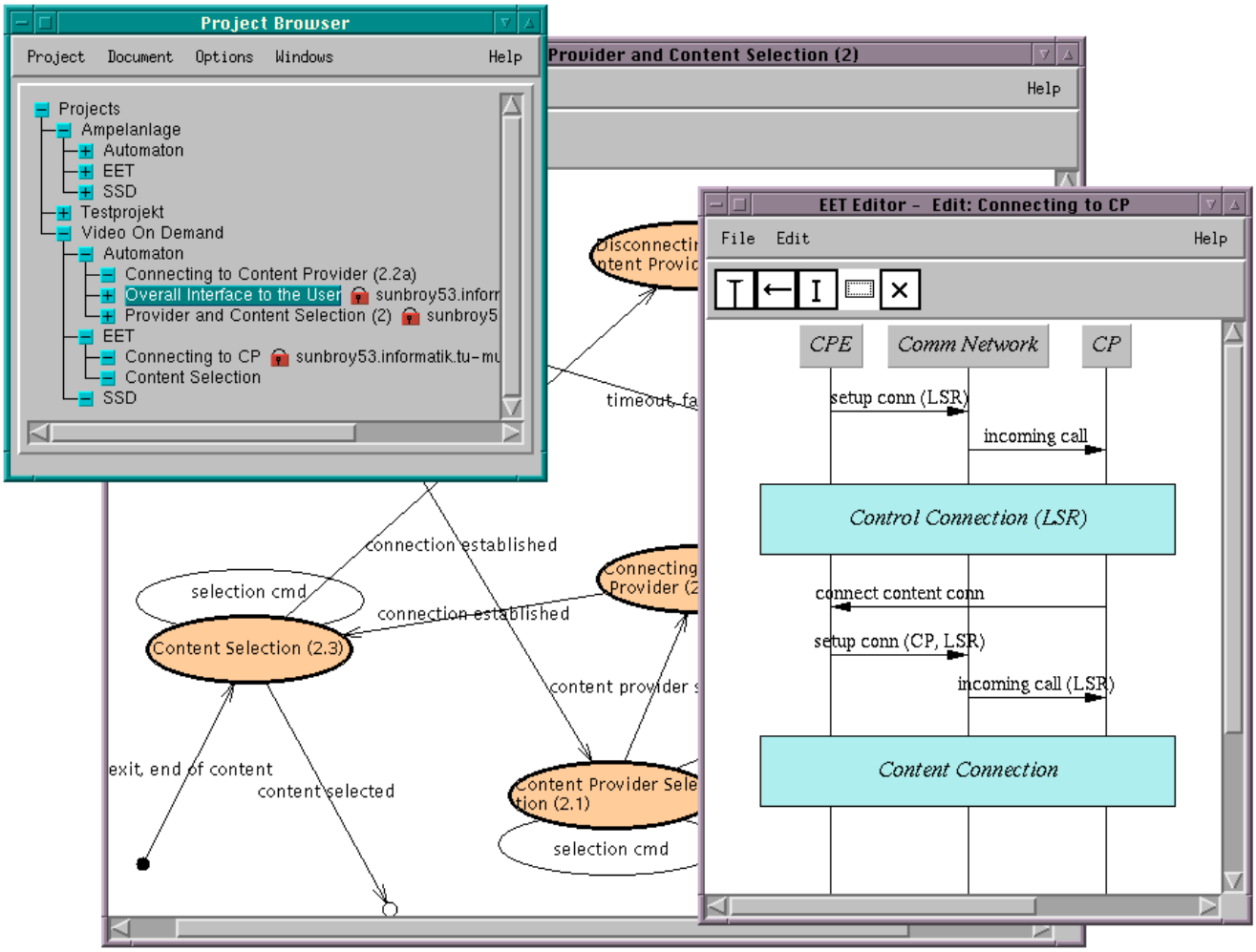

Figure 2. Project Browser, STD Editor, and EET Editor of the AUTOFOCUS Client

3. Project Organization

As in the former runs, we tried to keep the course as close as possible to a real industry project [1, 2]. The announcement of the course, published in our local university news groups and web pages, was for instance formulated like typical job offerings found in newspapers. After the registration we ran application interviews with the students in order to assign them to developer teams according to their interests and their knowledge in specific areas.

However, a number of circumstances usually found in industry projects could not completely be reproduced in our course. In contrast to "reality", where requirements of a new system are often unclear and keep changing even during the actual development process, the requirements for our tool were clearly stated at the beginning of the course, following a few months of conceptual work by the tutors. Furthermore, our student developer team was mostly unexperienced in software engineering issues, and many of them as well in object-orientation. We tried to alleviate this problem by extensive training during the first weeks of the course (two sessions per week, which were later used for project meetings). Probably the biggest difference to industry projects was our decision to base the development of a new software product on a technology, namely the Java platform, that was completely new for us and not yet evaluated.

The latter two issues were not felt as drawbacks compared to real-world development because they made the project situation even more challenging for our participants.

3.1. Developer Teams

The architectural view of the AUTOFOCUS tool depicted in figure 1 already implies a coarse modularization into the

following subsystems:

- a central repository,
- a set of graphical editors for
  - extended event traces and
  - system structure diagrams and state transition diagrams, both of which are basically editors for binary graphs and thus similar to implement,
- a central component on the client side coordinating the editors and providing a graphical interface to organize the documents in the repository.

In order to achieve a consistent internal data representation for all development documents, the actual data in the documents were kept separate from the editors and grouped together in an own module, following the Model-View-Controller paradigm [13]. The resulting five subsystems were each implemented by a team of two to three students.

3.2. Development Process

The development process we followed in the course was influenced by Denert [6] and had furthermore grown out of the experiences of the former two runs of the course. Our recommendations in that respect are summarized in [2].

We spent two weeks on system specification, three weeks on system design, six weeks on implementation, and two weeks on test and integration. Each phase was concluded by a milestone with a set of deliverables. Following each milestone, reviews were held for quality assurance.

We intended to make extensive use of prototyping, especially for the user interface parts of AUTOFOCUS, where we planned to use an evolutionary GUI prototype as a foundation for the final user interface of the system. Due to the poor quality of the user interface builders available for Java at that time (see also section 5), this was not possible; instead we started coding the user interface classes directly in Java very early.

4. Experiences

The reasons we chose Java as development language for AUTOFOCUS arose from former experiences with C++ and `itcl` [14], which were used in the first two runs of the course. Each of these languages had its deficiencies: C++ suffered from long turnaround times, the complexity of the language, and the necessity to manage storage by hand. Furthermore, the C++ implementations available to us suffered from the absence of easily usable, stable, and portable libraries and did not support important features of the language, like exception handling. `itcl`, an object-oriented variant of the `tcl` scripting language, did not have most of these drawbacks, but the language is, especially for teaching purposes, not very elegant, has no strong typing, and the programs run very slowly.

Java seemed to incorporate the virtues of both of these languages, avoiding their disadvantages. Another reason to choose Java was that the novelty, and the attention it was gaining in the media made it an interesting and appealing language for attracting interested students for the practical course.

In the following, we will examine whether Java could fulfill our expectations and also how well it supported the design process and the transition from analysis to design.

4.1. The Java Platform

In the whole, Java could be easily learnt by the students of our course, most of which had experience with C, but were not familiar with object-oriented languages. This experience differs from the one with C++ in the first course, where it took a long time before the students could master more than a basic set of features. Surprisingly, with Java the transition seemed easier because here it is hard to write "non-object-oriented" programs and developers are forced to change their habits quite radically.

4.1.1. Safe Execution Environment

We think that Java's safe execution environment and automatic garbage collection saved a lot of time because it considerably reduced the number of errors compared to our first project. In addition, most of the bugs were relatively harmless, and finding and fixing them did not need much time: Many bugs in our bug list were fixed by the responsible developers the day they were reported. The only "hard" bug during the course, taking nearly one week to find and fix, was caused by a bug in the standard `InputStream` class, whose objects were sometimes corrupted when the server's load was high. In contrast, the C++ project in the first course suffered from sporadic memory corruption errors which in rare cases needed weeks to hunt down.

4.1.2. Platform Independence

The platform independence of Java could only partly fulfill its expectations: In principle, the code (which we developed under Solaris) could be run on other platforms. However, there were some serious deficiencies and numerous small inconsistencies, mainly concerning the user interface area. Currently, we have verified the versions for Solaris, Linux and HP-UX 10, whereas it is not possible to run the AUTOFOCUS client on HP-UX 9, Microsoft Windows, or MacOS due to bugs of the AWT implementation. In the

case of Windows, there exist some "standard" workarounds [26, 21] to fix the problem (modal dialogs do not suspend the execution of the calling method), but since they all deteriorate the structure of the code, we chose not to implement them, but to wait for SUN fixing the problem instead.

4.1.3. Performance

In spite of these negative experiences, we were satisfied with the speed of the resulting code. Although the tool in its present form has serious drawbacks with respect to its performance, they are mostly due to the use of RCS as repository: For most actions involving the repository, a new process has to be scheduled on the server and a file has to be read, parsed and transmitted over the network. Actions concerning only the client side are surprisingly fast, even when they impose interactive repositioning of some graphical elements and redrawing the screen of an editor. The precaution measure to provide two modes of moving graphical objects with the mouse, one with instant diagram update during the move, one with update only at the end of the move, is hardly necessary even for larger documents.

4.1.4. Single Inheritance

Compared to C++, Java supports only single inheritance, but additionally offers so-called interfaces. This was not felt as a deficiency: The syntactical elements of FOCUS' description techniques could be (meta-)modeled as Java classes in a natural way. Single inheritance was used only for factoring out graphical information common to some elements. For example, the `Edge` class is used as a base class both for channels in SSDs and for transitions in STDs. Interfaces were rarely used: Only three were introduced overall, and two of these are only repositories for system constants and have no methods. The remaining "real" interface `TreeNode` captures the common functionality of the different entries (like projects or documents) in the project browser.

In some cases, the standard interfaces provided by the Java framework were implemented: `Runnable` was used to create runnable menu commands, and `Observer` was used for realizing the update mechanism between the clients and the server. We suspect that during the redesign of AUTOFOCUS, interfaces will be used more often because we expect that more features common to some classes can be factored out.

4.1.5. Lack of Parametric Types

Java's lack of parametric types was first felt as a serious drawback by us because it requires the use of dynamic casts at runtime, especially when using the powerful container classes of the `util` package, and therefore adversely affects the type safety of the system. At some time during the course we thought about solving this problem via a simple, macro-based template mechanism based on the substitution of a type parameter with an actual type, similar to the heterogeneous translation scheme proposed in the Pizza approach [15]. In the end, we did not choose this solution because it would have required the reworking of the already implemented uses of container classes. Furthermore, the students did not have any difficulties with it and there were hardly any `ClassCastException` runtime exceptions. One explanation for this is that most such casts follow a certain pattern and concern closely related methods in the same class: For example, the "standard" implementation of a 1-to-many association between classes `Foo` and `Bar` uses the `Vector` class and involves a `private Vector barElems` attribute in class `Foo` and `Foo` methods `void addBarElem(Bar)`, `void removeBarElem(Bar)`, and `Enumeration barElems()`. This way, care has to be taken only when iterating over the `Bar` elements because the enumeration provides no explicit type information about its elements.

4.1.6. Exception Handling

The exception handling concept of Java proved to be valuable, especially with respect to evolutionary prototyping. We were able to implement a first prototype that lacked most of the exception handling and could handle only the "easy" standard cases. Later on, when the design was stabilized, the code for exception handling could be added with relatively few problems.

4.1.7. Package Mechanism

The package mechanism was very useful because it made it possible to clearly structure the code following the responsibilities of the developer teams. We provided one package per team (and therefore also per subsystem, see section 2.2) and an additional utility package for code usable for all teams. The package mechanism and the visibility modifiers were quickly understood by the students and they used them as intended to hide their internal classes from the other teams to provide shallow and comprehensible interfaces. This approach may, however, lead to multiple implementations of similar functionality, worsened by the lack of some basic functionality in the standard libraries: At one time in the project, two teams each built their own simple dialog class, thus performing unnecessary work and compromising the uniformity of user interaction. Later on, this functionality was integrated into our `util` package. We recommend the installation of a person or team responsible for identifying, factoring out, and implementing possible `util` classes.

Another difficulty with packages is the lack of a tool for re-working their structure: If it is necessary to move classes between packages (or even to move whole packages), much manual editing is needed because all classes importing the class (or package) have to be changed.

4.1.8. Libraries

As mentioned above, one of the main reasons to use Java was the presence of a library of standardized and easy to use classes. To build a distributed CASE tool like AUTOFOCUS, libraries for concurrency, network, GUI, and graphics programming are indispensable. Apart from minor flaws, the functionality provided by the Java libraries was sufficient for our project in most areas. However, we were missing some basic functionalities in the area of GUI programming. Programming simple dialogs, toolbars, scrollable panels, and print support did consume much time during the course, and the migration to standard classes in new releases of the Java libraries will also consume much time in the future.

Two very useful facilities were not available in the standard libraries at the time of the course: For the communication and the transmission of commands between the clients and the server one team had to invent special, simple versions of features that will be contained in the "Remote Method Invocation" and "Object Serialization" standards of the next release of the JDK [25]. Although that was not particularly difficult given the standard libraries, it did cost quite some effort.

Another feature we were missing was the lack of powerful mechanisms for managing persistent storages. Our decision to use RCS with a flat file representation of the respective documents was very adequate for the repository of the prototype, since we got the sophisticated locking and revision control facilities of RCS "for free". The granularity level of a repository document, however, is too coarse for some of the planned extensions of AUTOFOCUS. Given our present design, complex queries like, e.g., "return all channels on which a message with a certain datatype can be transferred" involve the transfer of all documents from the repository server to the client. This is not acceptable, especially given the poor performance of the Java-RCS-connection, which requires the creation of a new server process for each repository action. We think that upcoming approaches like Java APIs for object-oriented databases [16] or orthogonal persistency [27] will fulfill our needs in the future.

4.2. Analysis and Design Techniques and Java

4.2.1. Analysis and Design Techniques

Due to the strict schedule of the course, the most important requirement for analysis and design techniques was ease of use. Some techniques turned out to be very useful and were well accepted by the participants: Among the notations provided by the Object Modeling Technique (OMT, [18]), we mainly used class diagrams to develop a first meta-model of the AUTOFOCUS description formalisms. This supported the participants in gaining insight into how the concepts of these formalisms work together and thus served to establish a level of common understanding among the participants. The meta-model was subsequently refined and could be translated almost without structural changes into Java (see also sections 4.1.4 and 4.1.5).

The lack of useful GUI prototyping tools for Java (see section 5) forced us to apply "handmade" rapid prototyping: To describe user interface behaviour at an abstract level, we used automata similar to the interaction diagrams from [6] together with mock-up "screen shots" that were produced using standard drawing software. Data and control flow between the different subsystems of AUTOFOCUS were specified at a high level using a notation similar to OMT's data flow diagrams.

4.2.2. Design Patterns

Design patterns played an important role during the development of AUTOFOCUS. Apart from their advantages for educational purpose [2], their knowledge (especially of those from [9]) is very helpful in understanding the architecture of the Java libraries because they were designed using a pattern approach (an overview over some of the patterns built into the AWT framework is given in [8]). In some cases, the Java libraries go one step further and provide explicit support for special patterns, thus turning an "immaterial" pattern into a standard framework mechanism. This is, e.g., the case for the `Observer` pattern from [9].

For our system, the `State` pattern from [9] turned out to be the most important one. We used it to create a `Behavior` class representing the actual mode of a graphical editor, and containing methods to handle user events like `singleClick` and `mouseDrag`. Special subclasses — like `ViewOnlyBehavior` and `SetChannelBehavior` — override them to provide a behavior according to the current mode of the editor, which can be selected via the toolbar. During the run of the AUTOFOCUS client, all reactions on events arising inside the canvas of an editor are performed by its actual `Behavior` object.

The `State` pattern was particularly useful because it allowed to clearly separate GUI-related code from the code responsible for handling user events, a concept not directly supported by the standard event handling mechanisms of the first version of AWT [22]. The new version 1.1 [24] provides an improved mechanism based on the delegation of events to special `EventListener` objects. This fits very well to our approach (a possibility is to make `Behavior` an implementation of the `EventListener` interface) and

will make the transition to the new API smooth. Furthermore, the separation of the code for the different modes of the editors made it easy to develop the editors step by step, using evolutionary prototyping.

5. Tools

Although the tools provided by Sun's Java Developer's Kit [23] make up just a very basic set of development tools, they proved to be sufficient for the development of AUTOFOCUS. Among the tools we found missing, the most important ones were a graphical, windowed debugger, a tool for managing and visualizing the inheritance and association structures of the classes and packages, and a decent GUI development tool. We hope that the upcoming integrated development environments will fulfill our needs in this respect.

Our experience with the JFactory GUI Builder V1.0 for Solaris [17], which we wanted to use for building an evolutionary prototype of the user interface, was less satisfactory: We could not use the generated code because it did not fit very well into the code structure of our project, the screen layout during the design sometimes did not match the screen layout of the generated programs, there were some serious flaws with respect to multi-user capability, and the tool crashed frequently. Therefore, we stopped using it for evolutionary as well as for explorative prototyping (see also section 4.2.1) very early in the development process. In the current JFactory version V1.1, however, some of these flaws were fixed.

Fortunately, the lack of a decent GUI builder was not a critical point because only a relatively small fraction of the program would have benefited from such a tool: Most of the code concerning the user interface went into the implementation of the editor behaviors and the custom components (like the toolbar and the tree browser). A GUI builder is not useful at all here. However, some of the AUTOFOCUS dialogs are not very appealing because optimizing the look of a "hand-crafted" GUI is very tedious and consumes too much time, especially given the poor infrastructure of the current AWT version.

6. Conclusion

To summarize our experiences gained in the project course, we can clearly state that Java is indeed more than an applet-only programming language. Despite its current deficiencies, which are, in our view, mostly problems of the current implementations, Java turned out to be a viable platform for implementing large and complex real-world applications. Compared to C++, it offers essentially greater ease of use, making it especially suitable for educational programming projects.

References

[1] W. Bartsch, K. Bergner, R. Hettler, and B. Paech. **S**tudenten **E**ntwickeln **U**niverselles **H**ochschulinformationssystem: Erfahrungen aus einem Softwaretechnik-Praktikum. In *SEUH '95, Report 44 of the German Chapter of the ACM*. Teubner-Verlag, 1995.

[2] K. Bergner. Under pressure – recommendations for managing a practical course in software engineering. In *Proceedings of Software Engineering: Education & Practice (SE:E&P'96)*. ACM Press, 1996.

[3] D. Bolinger and T. Bronson. *Applying RCS and SCCS: From Source Control to Project Control*. O'Reilly & Associates, Inc., 1995.

[4] M. Broy. Functional specification of time sensitive communicating systems. In M. Broy, editor, *Programming and Mathematical Models*. Springer, 1992. NATO ASI Series F: Computer and Systems Science, Vol.88.

[5] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The design of distributed systems - an introduction to FOCUS. TUM-I 9202-2, Technische Universität München, 1993.

[6] E. Denert. *Software-Engeneering*. Springer, 1991.

[7] D. Flanagan. *Java in a Nutshell: A Desktop Quick Reference for Java Programmers*. O'Reilly & Associates, Inc., 1996.

[8] E. Gamma. Java und Design Patterns – Eine vielversprechende Kombination. *Java Spektrum*, pages 18–24, September/October 1996.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[10] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus – A Tool for Distributed Systems Specification . In J. P. Bengt Jonsson, editor, *Proceedings FTRTFT'96 – Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470. LNCS 1135, Springer Verlag, 1996.

[11] F. Huber, B. Schätz, and K. Spies. AutoFocus – Ein Werkzeugkonzept zur Beschreibung verteilter Systeme . In U. H. H. Hermanns, editor, *Formale Beschreibungstechniken für verteilte Systeme*, pages 165–174. Universität Erlangen-Nürnberg, 1996. Erschienen in: Arbeitsberichte des Insituts für mathematische Maschinen und Datenverarbeitung, Bd.29, Nr. 9.

[12] International Telecommunication Union, Geneva. *Message Sequence Charts*, 1996. ITU-T Recommendation Z.120.

[13] G. E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[14] M. J. McLennan. [incr Tcl] — *Object-Oriented Programming in Tcl*. AT&T Bell Laboratories, 1247 Cedar Crest Boulevard, Allentown, PA 18103 (USA), 1993.

[15] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[16] Poet Software. Poet Homepage, 1996. `http://www.poet.com`.

[17] Rogue Wave Software, Inc., 850 SW 35th St., Corvallis, OR 97333 USA. *JFactory – The Visual Interface Builder for Java*, 1996. http://www.roguewave.com/products/jfactory/jfactory.html.

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International Inc., 1991.

[19] STP-96 Team. AutoFocus Systementwurf Version 2.1, 1996. http://autofocus.informatik.tu-muenchen.de/Results/systementwurf21.ps.gz.

[20] STP-96 Team. AutoFocus Systemspezifikation Version 2.0, 1996. http://autofocus.informatik.tu-muenchen.de/Results/systemspez20.ps.gz.

[21] SUN Microsystems, 2550 Garcia Ave., Mountain View, CA 94043-1100 USA. *AWT Modal Dialog Blocking Bug & Workaround*, 1996. http://java.sun.com/products/JDK/1.0.2/AWTModalBug.html.

[22] SUN Microsystems, 2550 Garcia Ave., Mountain View, CA 94043-1100 USA. *Java AWT: Delegation Event Model*, 1996. http://java.sun.com/products/JDK/1.1/designspecs/awt/events.html.

[23] SUN Microsystems, 2550 Garcia Ave., Mountain View, CA 94043-1100 USA. *The Java Developer's Kit, Version 1.0.2*, 1996. http://java.sun.com/products/JDK/1.0.2/.

[24] SUN Microsystems, 2550 Garcia Ave., Mountain View, CA 94043-1100 USA. *JDK 1.1 Preview*, 1996. http://java.sun.com/products/JDK/1.1/designspecs/index.html.

[25] SUN Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA 94043-1100 USA. *Java Distributed Systems*, 1996. http://chatsubo.javasoft.com/current/.

[26] T. Tempelmann. Modaldialogs.java, 1996. http://www.muc.de/~tt/java/ModalDialogs.java.

[27] University of Glasgow, Department of Computer Science. *PJava – The Persistent Java Project*, 1996. http://www.dcs.gla.ac.uk/pjava/.