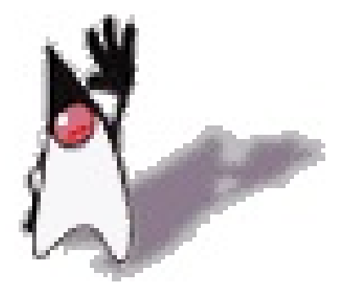
Must Java^{TM} development be so slow?

An "Oberon-inspired" Java^{TM} environment might be the answer to several problems.



AUSTRIA
Albrecht Wöß
Teaching & Research Assistant
Johannes Kepler University Linz
Institute of Practical Computer Science
System SoftWare Group
Altenbergerstrasse 69, 4040 Linz, Austria
Tel (Fax): ++43 / 732 / 2468 - 7134 (7138)
woess@ssw.uni-linz.ac.at
http://www.ssw.uni-linz.ac.at

Abstract

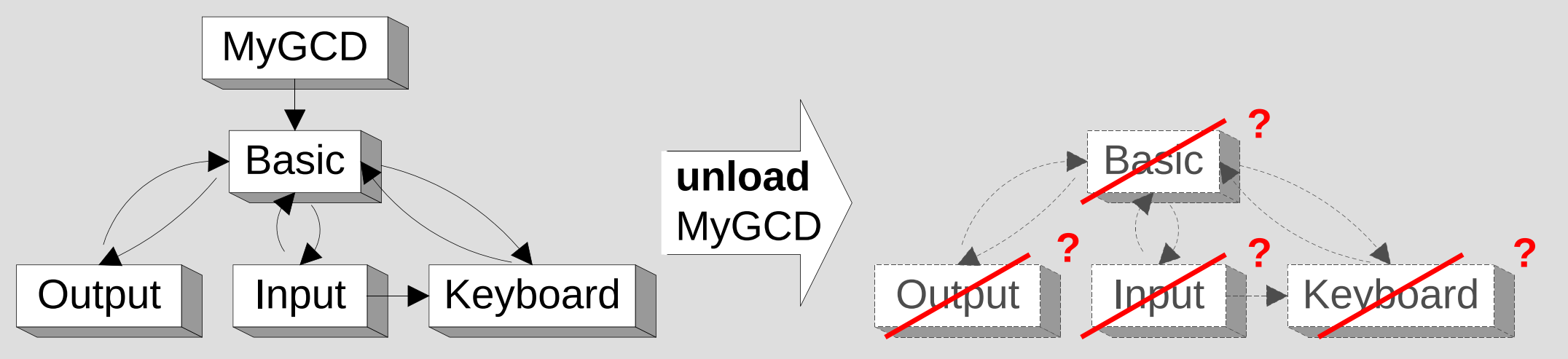
Starting a new VM each time an application is executed, forces numerous classes to be loaded multiple times. This overhead significantly slows down the development of Java^{TM} software.
So don't do it!
We plan to eliminate these bottlenecks with an open Java^{TM} environment where only one VM hosts all applications and classes are loaded only once.

Discussion

Type safety:
Does complete sharing of classes violate type safety?

Versioning:
Is it important to support the use of different versions of the same class, or is this just another hallmark of user-unfriendlyness?

Unloading:
When can a class be unloaded?



Reloading:
Is it possible to reload a new version of a class (whose interface has not changed) just by replacing the old one and copying its static field values?

Commands in Java^{TM}?
Does it make sense for Java^{TM} to add the notion of the command beside the one of a program as the unit of executable code?

Compatibility:
Will employing these ideas in Java^{TM} *require* a whole new way of programming and invalidate all existing Java^{TM} programs OR will it just *add* additional possibilities and still support execution of previously produced software?

Literate Programming:
Does LP actually prolong software development OR do we budget too little time for documentation?
Does LP really lower maintainance effort?

Further ideas, comments, critique, ...:
Any additional input is more than welcome!

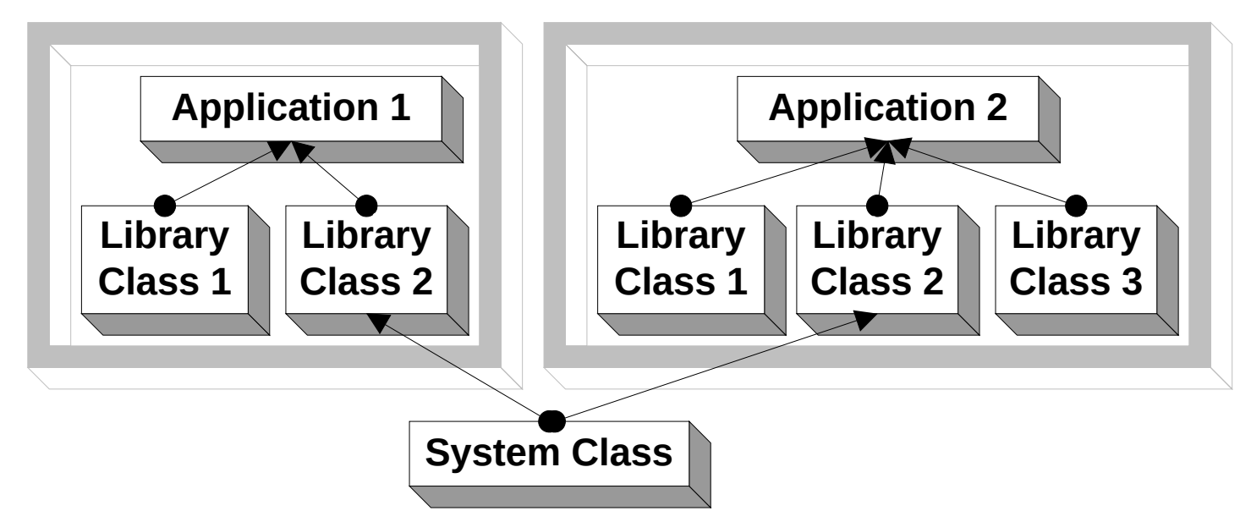
G O A L S

- Faster application start-up
- Better interoperability between applications
- Scripting of applications
- Reduced memory usage
- Literate programming

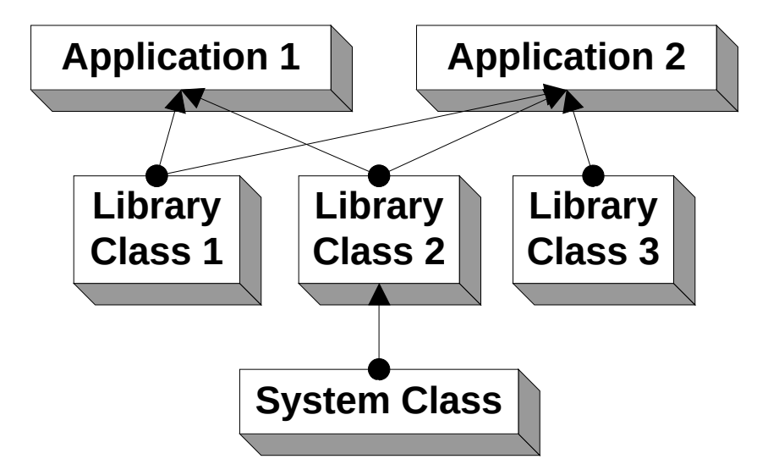
Details of the Approach

Single multi-tasking JVM:
Multiple applications execute on a single JVM.

Applications share classes and state:
Each class is loaded only once, and all applications use the same class object with all its fields when they use the same type (complete class sharing).
This implies that there are no separated memory areas.



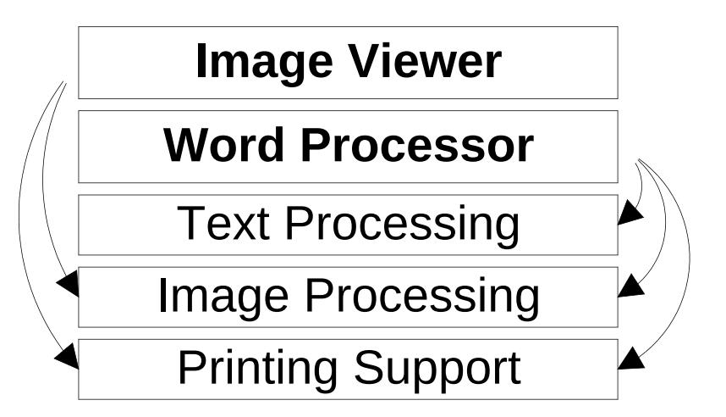
Separation
(=Traditional Java^{TM} Approach)



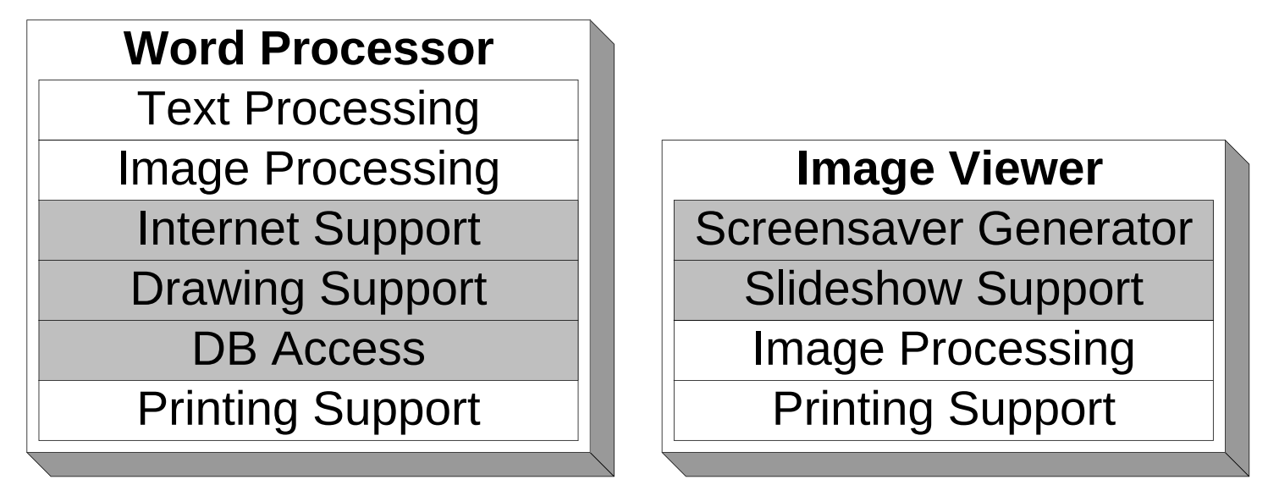
Complete Sharing

Classes remain loaded:
After an application terminates or a command execution ends all loaded classes remain in memory and retain their state.
Subsequent applications can communicate via shared state.
Unloading shall only happen when explicitly desired.

Minimal modules working together



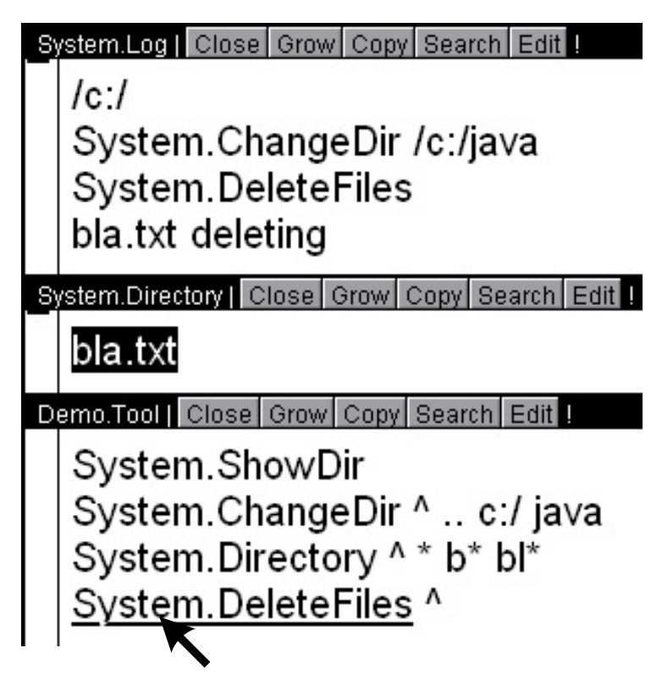
instead of



Monolithic programs in isolation

Commands:
Not only the *main*-method can serve as an entry point into a Java^{TM} program, but any other method, as well. These methods are called commands and specified like method calls (= *Classname*.*Methodsname*).

Point & Click commands in "*tool*" texts

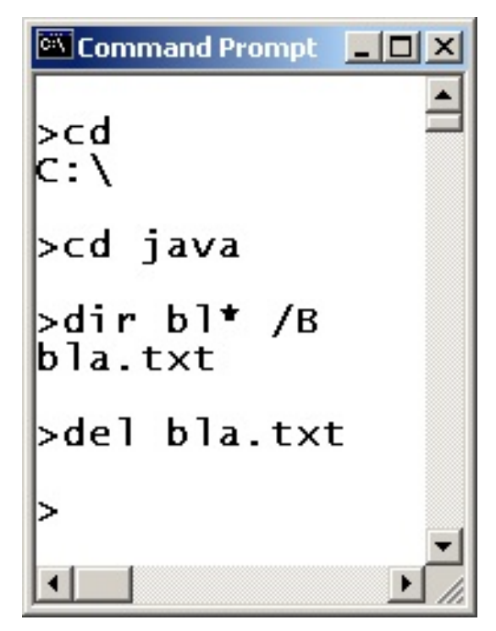

System.Log | Close Grow Copy Search Edit !
System.Directory | Close Grow Copy Search Edit !
Demo.Tool | Close Grow Copy Search Edit !

```
/c:/
System.ChangeDir /c:/java
System.DeleteFiles
bla.txt deleting
```

```
bla.txt
```

```
System.ShowDir
System.ChangeDir ^ .. c:/ java
System.Directory ^ * b* bl*
System.DeleteFiles ^
```

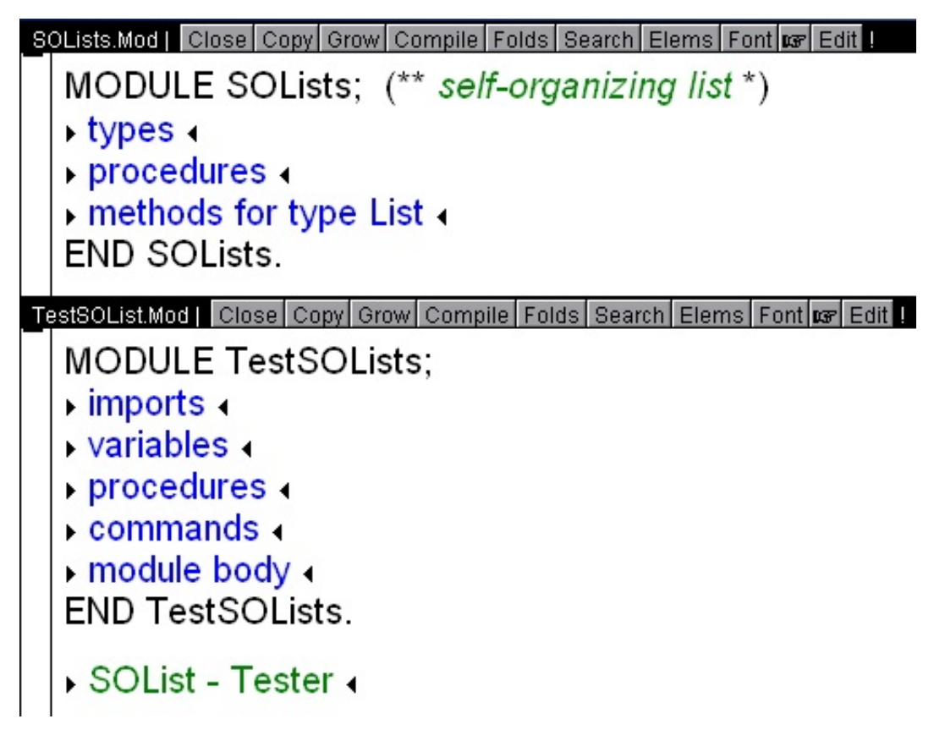
instead of



```
>cd
C:\

>cd java

>dir bl* /B
bla.txt

>del bla.txt

>
```

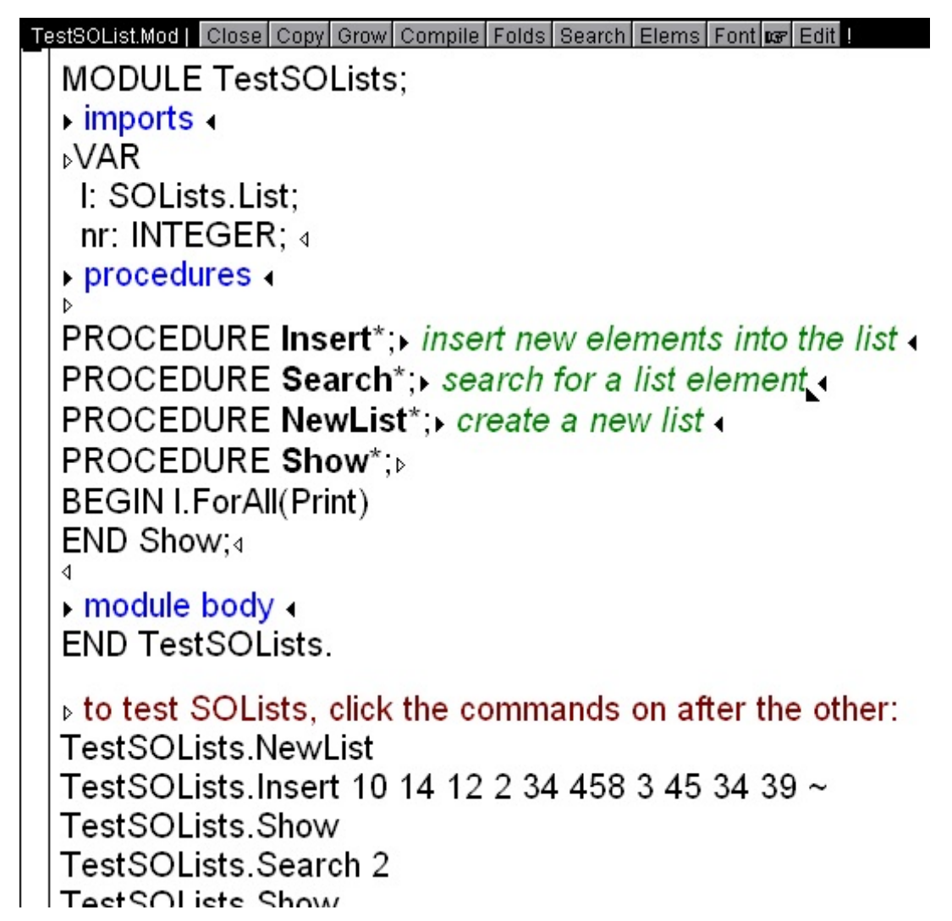
Remember & Type cryptic abbreviations at command prompts

Text Elements:
... are arbitrary objects - like graphics, tables, *Folding Elements*, ... - floating in texts. They will be ignored by the compiler and can be used to highly enhance program structuring and documentation directly in the source code.
This will lead to a new form of literate programming.


SOLists.Mod | Close Copy Grow Compile Folds Search Elems Font Edit !
TestSOList.Mod | Close Copy Grow Compile Folds Search Elems Font Edit !

```
MODULE SOLists;  (** self-organizing list *)
▸ types ◂
▸ procedures ◂
▸ methods for type List ◂
END SOLists.
```

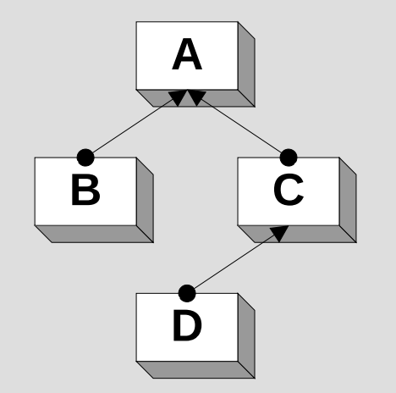
```
MODULE TestSOLists;
▸ imports ◂
▸ variables ◂
▸ procedures ◂
▸ commands ◂
▸ module body ◂
END TestSOLists.

▸ SOList - Tester ◂
```


TestSOList.Mod | Close Copy Grow Compile Folds Search Elems Font Edit !

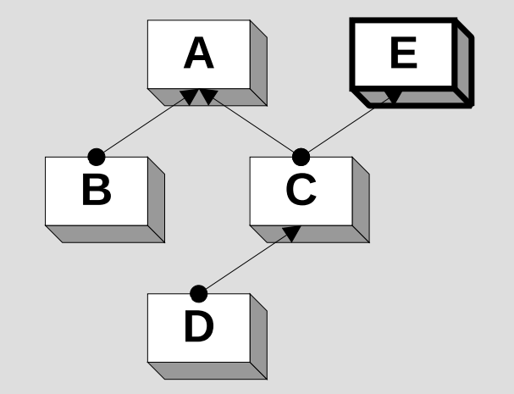
```
MODULE TestSOLists;
▸ imports ◂
▹VAR
  l: SOLists.List;
  nr: INTEGER; ◃
▸ procedures ◂
▹
PROCEDURE Insert*;▸ insert new elements into the list ◂
PROCEDURE Search*;▸ search for a list element ◂
PROCEDURE NewList*;▸ create a new list ◂
PROCEDURE Show*;▹
BEGIN l.ForAll(Print)
END Show;◃
◃
▸ module body ◂
END TestSOLists.

▹ to test SOLists, click the commands on after the other:
TestSOLists.NewList
TestSOLists.Insert 10 14 12 2 34 458 3 45 34 39 ~
TestSOLists.Show
TestSOLists.Search 2
TestSOLists.Show
```

How it works

User executes a command by clicking on it in a text:
A.foo
This results in:
- Loading of class *A*
- Loading of all classes used during execution (*B*, *C*, *D*)
- Execution of method *foo* of class *A*
- All classes remain loaded when *foo* terminates

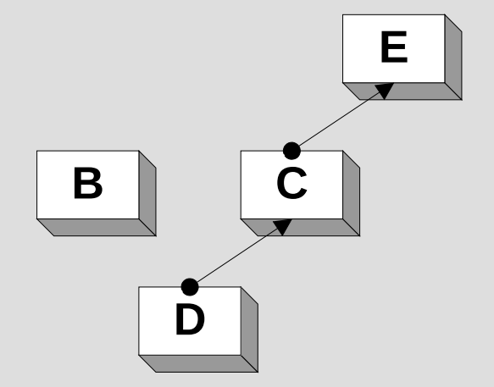


User executes another command:
E.bar
This results in:
- Loading of class *E* (if not already in memory)
- Loading of all classes used during execution that are not already loaded (-)
- Execution of method *bar* of class *E*
- All classes remain loaded when *bar* terminates



↳ Only *E* had to be loaded!
↳ *A* and *E* **share** the **state** of class *C*!

User **unloads A**:



↳ Only *A* is unloaded!
↳ All other classes stay in memory, and can be reused by other applications.

References:

(1) G.Czajkowski, ***Application Isolation in the Java™ Virtual Machine***, in OOPSLA '00 Conf.Proc., pp. 354-366, ACM Press, 2000.
(2) A. Fischer, H. Marais, ***The Oberon Companion - a Guide to Using and Programming Oberon System 3***, vdf Hochschulverlag AG an der ETH Zürich, 1998, ISBN 3-7281-2493-1.
(3) L. Gorrie, ***Echidna - a Free Multitask System in Java***, 2001, http://www.javagroup.org/echidna.
(4) D. Knuth, S. Levy, ***The CWEB System of Structured Documentation***, Addison-Wesley, 1993, ISBN 0-201-57569-8.
(5) S. Liang, G. Bracha, ***Dynamic Class Loading in the Java™ Virtual Machine***, in OOPSLA '98 Conf.Proc., pp. 36-44, ACM Press, 1998.
(6) H. Mössenböck, K. Koskimies, ***Active Text for Structuring and Understanding Source Code***, SOFTWARE - Practice and Experience, 26(7), 1996, pp. 833-850.
(7) ***ETH Oberon home page***, http://www.oberon.ethz.ch .
(8) M. Reiser, ***The Oberon System - User Guide and Programmer's Manual***, ACM Press, Addison-Wesley, 1991, ISBN 0-201-54422-9.
(9) N. Wirth, J. Gutknecht, ***Project Oberon - The Design of an Operating System and Compiler***, ACM Press, Addison-Wesley, 1992, ISBN 0-201-54428-8.