Minik: A Tool for Maintaining Proper Java Code Structure[⋆]

Jacek Chrząszcz[2], Tomasz Stachowicz[1],
Andrzej Gąsienica-Samek[1], and Aleksy Schubert[2,3]

[1] Comarch SA, Warsaw, Poland
[2] Institute of Informatics
Warsaw University, Poland
[3]SoS Group NIII,
Faculty of Science, University of Nijmegen, Netherlands

Abstract. Maintaining discipline of code in an evolving software project is known to be difficult. We present Minik, an automatic tool written in Java and for Java, that assists technical managers to enforce high and medium level design decisions on programmers. The tool supports hierarchical encapsulation of software components and helps to maintain order in dependencies between parts of the project's source code and to control calls to external libraries.
Minik was created to support the development of Ocean GenRap Report Generator, a complex Java project of over 350KLOC, developed in Comarch Research Center. With time, it became an invaluable help for technical managers as well as for new programmers who could quickly learn the structure of the code base.

1 Introduction

Development of large software projects often escapes traditional waterfall software creation methodology. This concerns in particular systems of big complexity, such as compilers, database engines or modern spreadsheets. Agile approach to software creation process seems much better suited to this kind of software, which is by nature in constant development, improving (hopefully) with each release in terms of enhanced functionality and stability. However, this kind of iterative development style may easily lead to overly complicated code, where almost every part of code depends on every other. Such structure is very difficult to maintain and develop [21], so it is crucial to create methodologies and tools supporting project managers in their task of limiting the code complexity without precluding integration of new features and improvements.

Mainstream programming languages, such as Java, offer some support for hierarchical code organization, but the support is limited. While encapsulation on the level of one file (class) or one directory (package) is supported by the language, higher level encapsulation is left to the programmer. In particular, even if files are placed in a hierarchical directory structure, from the Java programming language point of view the package structure is flat. As far as e.g. method visibility is concerned, two classes

[⋆] This work was partly supported by KBN grant 3 T11C 002 27 and Sixth Framework Programme MEIF-CT-2005-024306 SOJOURN.

may either be from the same package or from different ones, regardless if they are just in sibling directories or far away apart in the directory structure. Consequently, a modification of a method annotated as *public*, may potentially require the knowledge needed to modify any part of the program.

Moreover, the flexibility of these design standards and programming language grouping constructs like packages make it easy to introduce circular dependencies. The experience in software development shows that circular dependencies cause problems [9, 17, 23, 26] so the acyclic coding pattern occurs often in project design guidelines [7, 16, 18]. Cyclic dependencies are regarded as a strong factor in measures of code complexity [27] especially when maintainability of code is of the main interest [15]. Moreover, the presentation of code dependencies in form of a DAG (directed acyclic graph) has already been used in the context of support for maintainability [2] and easy extensibility [19].

In this paper we present Minik, the code management tool, primarily created to support orderly development of the Ocean GenRap Report Generator [10] by the Comarch Research Center. Minik supports a true hierarchical encapsulation and enforces a transparent and simple acyclic dependency structure. In a project managed using Minik, dependencies on distant packages are declared in a separate .minik management file of each component. The tool makes sure that all dependencies are declared and that they form an acyclic structure. Programmers are required to run Minik at every build, so the dependency descriptions are always up-to-date. Moreover, a modification of the .minik file, requires the consent of the technical manager. Experience shows that these changes tend to happen less and less frequently. Minik is also useful in enforcing certain design patterns, e.g. Facade or Bridge [11], which can be recorded in the .minik files and hence become harder to break by programmers.

Another important aspect of Minik is the possibility to constrain the usage of external dependencies inside the project. Minik can immediately enforce the manager's decisions that e.g. JDBC classes can only be used in the DAO implementation or that the class java.lang.Thread can only be used in the main package of the application and not inside components. Any programmer trying to break this policy, either by haste or unawareness, will immediately be warned by Minik and forced to correct the mistake.

The adoption of Minik by GenRap programmers turned out to be very smooth. After an initial reluctance, a natural people's reaction to any limitation, the programmers started treating the Minik discipline as part of the limitations of the programming environment, like e.g. Java type system. Moreover, since .minik files constitute only a fraction of the whole code (less than 1%), they proved to form a very good guide for programmers which were new to the GenRap project.

Minik implements the core functionality of the Kotek methodology. The latter, presented in [12], is an advanced module system combined with a build tool, that extends Minik with precise inter-component contract specifications, parametrisation of large code fragments with respect to some interface (e.g. widget library) and conditional compilation, depending e.g. on a hardware platform.

The present paper is organized as follows. In the next section we describe basic features of Minik and limitations that it puts on the project structure in order to maintain clarity. In the presentation we use a simple example of a toy project MiniEdit, whose structure is a (substantial) simplification of the structure of GenRap. Then we present and explain the syntax of the .minik files and describe the impact of Minik on the development of the GenRap project.

2 The structure of software projects

The complexity of contemporary software structure has led to several approaches aiming to conceptually simplify dependency diagrams of software. Most approaches rely on providing the developers means to present the code interdependencies and coupling them with source code quality metrics. Examples of systems in this category are [5,6,13,22,24]. The systems give guidance to good source code structure, but do not enforce or enforce in a weak manner the structural rules which are appropriate for the project at hand.

Another approach is represented by MJ [3], a rich system of modules for Java. In this case, the software modules are the grouping entities which forbid accesses which are not explicitly declared in module descriptions. This mechanism, however, does not impose any structural restrictions on the way the dependencies are organized.

Yet another approach to structuring the source code consists in the use of a type system which controls the read or write access to particular pieces of the code. In this case, a separation between different code pieces is governed by local annotations in the source code (or in the comments in the source code) that specify which classes are intended to be used as a single module. Examples of systems in this group are [4,8,28]. These systems allow to describe detailed data dependencies up to the level of fields in objects.

2.1 Software project structure enforced by Minik

We describe here the structure of projects that is enforced by our tool and methodology. We focus on greater units of source code, called modules or components. Conceptually, the basic ones should contain several classes or packages, the complex ones consist of several sub-components (and possibly a few additional classes). To introduce the notions addressed by Minik, we use a simple example of a hypothetical editor MiniEdit, whose structure is a considerable simplification of the structure of the GenRap project.

The strength of Minik results from the structure of possible interdependencies between components that describe the way the source code is organised. We consider two perspectives of code organisation. The first one, vertical, corresponds to the hierarchical division of the project into components, components into sub-components and so forth. The second one, horizontal, describes functional dependencies between fragments of code. Other Java module systems did not consider explicitly these code organisation facets [1,3,14].

2.2 Vertical structure

The hierarchical structure of the project that is enforced by Minik corresponds well to good organisational patterns in which hierarchical connections allow to avoid communication blow-up between different organisational units. This kind of code

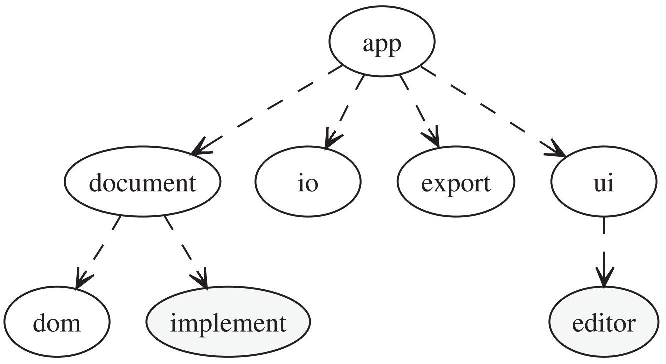

Fig. 1. The vertical structure of the MiniEdit project.

management support is often present in programming languages. The Java package system, in which packages correspond literally to the directory tree of the development site, is the most notable example of it. In our example (Fig. 1) the MiniEdit application (represented by the topmost component app) is divided into four components of which two have sub-components. Minik strengthens the Java package system by enforcing a true hierarchical encapsulation: it is forbidden to refer to the insides of a component without its permission. We discuss it further at the end of the next section.

2.3 Horizontal structure

By horizontal structure of the project we mean functional dependencies between components of our project. We say that an entity (class/package/component) M depends on another entity N when the source code in M refers a class, a method or a value in N. Figure 2 presents the graph of dependencies between main components of the MiniEdit application. In order to obtain a system with low maintenance cost, we impose several restrictions on the structure of possible references between components.

The first restriction is based on the assumption that functional dependencies between components should form a DAG. The experience in software development shows that circular dependencies cause problems, especially when maintainability of the code is of the main interest.

Of course, cyclic dependencies are not problematic when they occur in a fragment of code whose size is small enough to be easily grasped by a programmer. Therefore Minik does not prevent dependency cycles between classes belonging to one component, but only the *big* cycles, i.e. involving classes in several components.

Dependencies and encapsulation The second restriction concerns the vertical structure that we introduced earlier. It is based on a natural principle that one should not manipulate the internals of another component, unless explicitly authorised. In our example the document.minik file declares the dom sub-component as exported (see Fig. 5 and its description in Sect. 3), but not the implement sub-component (marked gray in Fig. 1). Consequently, the code in e.g. ui may use the (public) classes defined in app.document.dom, but not those defined in app.document.implement.

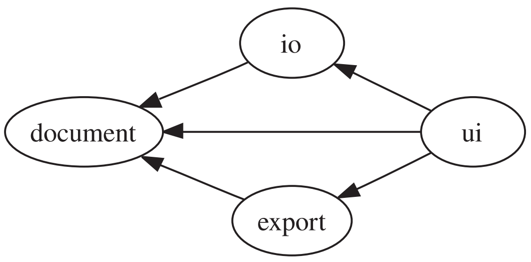

Fig. 2. The horizontal structure of the middle layer of MiniEdit.

3 Syntax and example

The usage of Minik is directed by .minik files, which are placed in most directories of the project source tree. The syntax of those files is very straightforward (see Fig. 3). We explain it here, using our MiniEdit example, whose structure is depicted in Fig. 1 and 2.

The contents of a .minik file consists of four parts. The first one, starting with the keyword **use**, specifies the dependencies of the the given component. In Fig. 4, one can see the contents of the main .minik file of the MiniEdit application, app.minik, so its first line specifies external dependencies of the whole project: the Java standard library and an (imaginary) library to produce PDF documents. For internal components, such as ui (Fig. 6), one specifies dependencies on (parts of) external components of the whole projects (like java_lang, java_ui) and other components of the project (document, io, export).

The second section contains definitions of restricted components. They are used to control which parts of external dependencies are used where in the project source. In general, from the dependency control point of view, a component is simply a name attached to a list of class names. Consequently, a definition of a restricted component consists just in creating a new name for the list of classes obtained by restricting the list attached to the original component. One can use the following optional restriction operations:

– positive restriction – keyword **allow** – from the list of class names of the original component, we select only those which match at least one of the given class patterns,
– negative restriction – keyword **deny** – from the list of class names, we subtract those which match one of the given class patterns.

Of course if both restrictions are omitted the new component is just a renaming of the original one.

In case of app.minik, we name various parts of standard library and the PDF writer in order to precisely say, in the next section of app.minik, which classes can be used in which components. The third section describes how sub-components of the given component depend on one another, on external components and on restricted components. For example, it is easy to see that the ui component depends on all other sub-components of app and that it is the only one allowed to manipulate reflection, threads and other java_lang classes not included in java_core. Besides, only export can access classes of pdfwriter, and only those defined in the topmost package, not internal ones. A special name **this** can be used as the target of the last build clause (see e.g. Fig 5) to indicate what dependencies are allowed in classes from the current directory.

Since the name of a sub-component c_1 can only be used as a dependency of the another sub-component c_2 after the corresponding **build** c_1 clause, the structure of dependencies cannot contain cycles.

The last part of the .minik file lists the names of the sub-components to be exported. This is where hierarchical encapsulation is implemented: other components can only refer to those parts of the current component which it explicitly lists as exported.

In case of the topmost .minik file of the application, the **return** clause only indicates the component containing the class with the main method.

```
   <minik>  ::=  <use><define> .. <define><build> .. <build><return>
     <use>  ::=  use <ident> .. <ident>
  <define>  ::=  define <ident> = <ident>  [ allow {<package>, .., <package>} ]
                                           [ deny {<package>, .., <package>} ]
 <package>  ::=  <ident>.<package> | <ident> | * | **
   <build>  ::=  build <thident> : <ident> .. <ident>
  <return>  ::=  return <thident> .. <thident>
 <thident>  ::=  this | <ident>
```

Fig. 3. *Syntax of* .minik *files.*

```
use java pdfwriter

define java_core =
  java allow { java.lang.* }
       deny { java.lang.Class, java.lang.ClassLoader, java.lang.Compiler,
              java.lang.Process, java.lang.Runtime, java.lang.Thread }
define java_lang = java allow { java.lang.* }
define java_io = java allow { java.io.*, java.nio.** }
define java_xml = java allow { javax.xml.**, org.xml.**, org.w3c.dom.** }
define java_ui = java allow { java.awt.**, javax.swing.**, javax.print.** }

define pdf = pdfwriter allow { com.pdfwriter.* }

build document : java_core
build io : document java_core java_io java_xml
build export : document pdf java_core java_io
build ui : document io export java_lang java_ui

return ui
```

Fig. 4. *The file* app.minik *of MiniEdit.*

```
use java_core

build dom : java_core
build implement : dom java_core
build this : dom implement java_core

return dom this
```

Fig. 5. *The file* document.minik *of MiniEdit.*

Every directory of the project can have its own .minik file. If it is missing, all classes in the directory and its subdirectories are treated as one basic component.

use *document io export java_lang java_ui*

build *editor* : *document java_lang java_ui*
build this : *editor document io export java_lang java_ui*

return this

Fig. 6. *The file* ui.minik *of MiniEdit.*

Its exported classes are those declared as public. In MiniEdit the document and ui components have their own .minik files. The document.minik file is given in Fig. 5. It specifies the Facade design pattern: the dom sub-component defines the interface, i.e. the data object model together with the names of the operations that can be performed on the document. Next, the implement sub-component contains the actual implementation of the data structure representing the edited document. It may use the dom sub-component, for example to say that some classes of implement are instances of interfaces defined in dom. The classes in the document directory relate the specification and implementation, for example by providing factory functions returning an object created by a class from implement, satisfying an interface specified in dom. The last line says that only the classes exported by the dom component and classes in the document directory can be used outside document.

The file ui.minik, presented in Fig 6 is similar to document.minik, but the interface part is not placed in a separate sub-component.

It is worth noting how the use of the external component pdfwriter can be traced in the project using the .minik files. Indeed, app.minik tells us to look only in the export component and nowhere else.

How Minik *works.* The most important part of Minik is the recursive function *minik_fun* operating on an environment which maps component names to sets of Java class names.

The initial environment describes the external dependencies of the project and is created from the arguments supplied by the user in the invocation of Minik. One of the arguments is the directory containing .jar files of the dependencies. By default, for each dependency M, the file M.jar should be placed in this directory, apart from the java component, which is found in the standard location of the Java installation. For our example, the only external dependency file is pdfwriter.jar.

The initial environment passed to the first invocation of minik_fun is created by scanning the .jar files of dependencies.

The function *minik_fun* takes an environment E and a directory name D, and returns the set of names of classes exported by the component located in the directory D.

If the .minik file is missing in the directory D, the function just checks the legality of the dependencies of all classes in D and its sub-directories: it is verified that all referenced class names are in the range of the environment E. The returned set of classes includes all public classes of D and its sub-directories.

If .minik is present in D, minik_fun operates in four steps, corresponding to sections of .minik. The first step consists in checking that dependency names listed in the **use** clause are valid, i.e. they are in the domain of the environment. In the second step the **define** clauses are processed: the environment E is extended with restricted components. In the third step, for each clause **build** $c : d_1 \dots d_n$, the function *minik_fun* is called recursively with the environment $E|_{d_1 \ldots d_n}$ (i.e. E restricted to the dependencies allowed for the sub-component c) and directory name D/c, which checks correctness of the dependency structure of the sub-component c and returns the set of classes C_c exported by c. The mapping $c \mapsto C_c$ is then added to E for the processing of subsequent sub-components. If the last build clause is of the form **build this** : $d_1 \ldots d_m$, the legality of dependencies of classes in D is checked: all referenced class names must be members of the components $d_1 \ldots d_m$. The last step is the processing of the **return** $c_1 \ldots c_k$ clause. It is checked that $c_1 \ldots c_k$ are components built in step 3 (and not external or restricted components) and the set of all classes exported by components $c_1 \ldots c_k$ is returned as the result of *minik_fun*.

4 GenRap: The Minik experience

Ocean GenRap [10] is a complex application of over 350 thousand lines of code, written mostly in Java. It is a report generator for database applications, allowing for intuitive and easy edition of reports with constant data view, enabling data analysis directly in the edited document. It has a graphical user interface, similar to modern text editors or spreadsheets, and a novel live context association mechanism, allowing the user to move fragments of reports between documents. GenRap has the possibility to connect to a number of database engines and export the generated report to popular formats, including pdf and html. It is available as part of the CDN OPT!MA system [20] since mid 2005 and as a standalone application since January 2006.

The development of GenRap started in 2003. Since then, it has been actively developed by a dozen of enthusiastic programmers, following an agile development methodology. There is no precise long term development plan, only the product vision from which the detailed plan for a following couple of months is derived. The vision itself is modified as new features are implemented and users give their feedback. Such cycles usually take two to three months. During that time two processes are done in parallel: implementation of new features and maintenance, consisting in bug-fixing and code refactoring.

From the historical perspective, the need for a tool helping to manage the code became clear after a few months of intensive coding, when the code reached 40 thousand lines. In order to be maintainable the project needed a strict regime in encapsulating and separating components. A simple bash script to separately compile components in restricted environments was used at first. If the code contained a disallowed dependency the compilation just stopped with an error.

Later, it was decided that this policy was too strict. For productivity reasons, a developer should be able to build the project with bad dependencies, but a patch supplied to the central repository should always ensure a correct dependency structure.

Minik was implemented with this idea in mind. The tool automatically checks the structure of the code without completely preventing defective builds. Apart from that, other correctness tests were incorporated to Minik, which are beyond the scope of this paper.

As experience shows, the .minik files constitute between 0.5% and 1% of the whole code (see Fig. 7). It turns out that they are modified more-less in one out of 10

	Patches				Source Code					
Month	P	PM	PM/P	PLOC	NM	LM	NJ	LJ	NM/NJ	LM/LJ
2004-10	67	2	2.98%	5 970	65	1065	1106	124 283	5.87%	0.85%
2004-11	136	15	11.02%	35 871	66	1158	1120	146 076	5.89%	0.79%
2004-12	78	8	10.25%	57 856	70	1216	1219	159 238	5.74%	0.76%
2005-01	53	14	26.41%	70 518	75	1282	1246	178 340	6.01%	0.71%
2005-02	65	17	26.15%	73 442	79	1388	1330	187 130	5.93%	0.74%
2005-03	89	13	14.60%	38 898	100	1734	1372	199 035	7.28%	0.87%
2005-04	33	6	18.18%	15 013	106	1871	1459	220 518	7.26%	0.84%
2005-05	66	13	19.69%	43 454	113	1960	1512	227 104	7.47%	0.86%
2005-06	105	12	11.42%	22 368	125	2200	1612	244 235	7.75%	0.90%
2005-07	85	5	5.88%	89 641	126	2245	1638	249 815	7.69%	0.89%
2005-08	114	5	4.38%	10 940	127	2260	1653	251 459	7.68%	0.89%
2005-09	89	1	1.12%	7 699	129	2292	1662	254 513	7.76%	0.90%
2005-10	130	10	7.69%	51 236	129	2292	1676	257 662	7.69%	0.88%
2005-11	191	19	9.94%	42 754	137	2431	1746	278 380	7.84%	0.87%
2005-12	190	12	6.31%	44 669	158	2683	1864	311 194	8.47%	0.86%
2006-01	130	2	1.53%	14 965	164	2758	1921	327 098	8.53%	0.84%
2006-02	119	9	7.56%	44 640	164	2761	1939	332 677	8.45%	0.82%
2006-03	115	12	10.43%	49 366	164	2768	1923	326 754	8.52%	0.84%
2006-04	20	4	20.00%	38 420	168	2864	2002	345 471	8.39%	0.82%
Total	1875	179	9.54%	757 720						

P = Total no of patches PM = No of patches touching .minik
PLOC = Total no of lines of code patched
NM = No of .minik files LM = LOC of .minik
NJ = No of .java files LJ = LOC of .java

Fig. 7. GenRap development statistics.

commits. Thanks to the good structure, the project enjoys a stable growth in lines of code per month and the project managers are not afraid to improve any of its components. Indeed, since it is easy to see what depends on a given fragment of code, it is possible to foresee the impact of a planned refactoring on the rest of the code base.

Currently the GenRap code is divided into around 170 hierarchic components, described by as little as 2800 lines of .minik files. Almost all of these files are smaller than 50 lines, the average being about 17. Their structure is also very simple so they are very easy to understand.

Using Minik in the project has also a positive psychological impact on the programming team's integrity. The programmers do not feel intimidated by a manager pointing out their structure errors. Instead, they just treat limitations imposed by Minik as part of the limitations of the working environment: the language, the compiler, design patterns and Minik.

The tool itself is written in Java, it has about 3000 lines and uses a custom class file parser. To increase its integration with the working environment an Eclipse plugin for Minik has been developed. It is rather basic, but nevertheless it is possible to automatically start the verification process and easily access the files with bad dependencies.

5 Conclusions

The need to synchronize architecture documents with the actual source code is a very important aspect of modeling. Many tools supporting UML technology, e.g IBM Rational Software Architect [25] or Microsoft Visual Studio [29], have included utilities to synchronize source code changes with the evolution of the visual model, which is called round-trip engineering. However, the tools based on UML emphasise early project development stages. In particular, they provide clustering and encapsulation mechanisms in the design stage of software production but these architectural decisions are weakly enforced in the coding and maintenance stages. Moreover, they do not encourage comprehensive arrangement of construction blocks and so complicated diagrams are commonly encountered.

In this paper, we have presented Minik, a light-weight tool to maintain proper structure of Java projects, realized in Comarch Research Center as a development utility for the Ocean GenRap Report Generator. Minik supports the technical managers in enforcing acyclic structure of inter-component dependencies and helps programmers understand and maintain the structure of the project. It supports true hierarchical encapsulation of software components, helps tracking where external dependencies are used in the code and permits to foresee the impact of planned refactoring.

Thanks to Minik and its consistent use in project management, the development pace of GenRap is steady for over two years without increasing the programmers team. It turns out that the structure of the code grows as fast as its size and therefore the development does not lead to bloated code which is often a nightmare in large software projects.

References

1. Davide Ancona and Elena Zucca. True Modules for Java-like Languages. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 354–380, London, UK, 2001. Springer-Verlag.
2. Liz Burd and Stephen Rank. Using Automated Source Code Analysis for Software Evolution. In *1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), 10 November 2001, Florence, Italy*, pages 206–212, 2001.
3. John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: A Rational Module System for Java and its Applications. In *Object-Oriented Programming, Systems, Langauges & Applications*, 2003.
4. Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–310, New York, NY, USA, 2002. ACM Press.
5. Mike Clark. Jdepend. http://www.clarkware.com/software/JDepend.html.
6. Compuware. JavaCentral. http://frontline.compuware.com/javacentral/tools/26222.asp.
7. Compuware. Optimaladvisor supersedes the Package Structure Analysis Tool. Technical report, JavaCentral, 2005.
8. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.
9. Martin Fowler. Reducing Coupling. *IEEE Software*, July/August 2001.
10. Ocean GenRap report generator. Comarch Research Center. http://ocean.comarch.pl/genrap/.

11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, New York, NY, 1995.
12. Andrzej Gąsienica-Samek, Tomasz Stachowicz, Jacek Chrząszcz, and Aleksy Schubert. KOTEK: Clustering of The Enterprise Code. In Krzysztof Zieliński and Tomasz Szmuc, editors, *Software Engineering: Evolution and Emerging Technologies*, volume 130, pages 412–417. IOS Press, 2005.
13. Alex Iskold, Daniel Kogan, and Goran Begic. Structural analysis for java. http://www.alphaworks.ibm.com/tech/sa4j.
14. Yuuji Ichisugi and Akira Tanaka. Difference-Based Modules: A Class-Independent Module Mechanism. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 62–88, London, UK, 2002. Springer-Verlag.
15. Stefan Jungmayr. Testability Measurment and Software Dependencies. In *Software Measurement and Estimation, Proceedings of the 12th International Workshop on Software Measurement (IWSM2002)*. Shaker Verlag, 2002. ISBN 3-8322-0765-1.
16. Kirk Knoernschild. Acyclic Dependencies Principle. Technical report, Object Mentor, Inc., 2001.
17. J. Lakos. *Large-scale C++ software design*. Addison-Wesley, 1996.
18. Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.
19. D. Notkin and W. G. Griswold. Extension and software development. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 274–283, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
20. CDN OPT!MA. Comarch. http://www.comarch.pl/cdn/Products/.
21. A. Podgurski and L. A. Clarke. A Formal Model of Program Dependencies and Its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.
22. Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *OOPSLA '05: Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 167–176, New York, NY, USA, 2005. ACM Press.
23. Barry Searle and Ellen McKay. Circular Project Dependencies in WebSphere Studio. *developerWorks, IBM*, 2003.
24. Chris Smith. Japan. http://japan.sourceforge.net/.
25. Ibm Rational Software Architect. http://www-306.ibm.com/software/awdtools/architect/swarchitect/.
26. J. Soukup. *Taming C++*. Addison-Wesley, 1994.
27. Lassi A. Tuura and Lucas Taylor. Ignominy: a tool for software dependency and metricanalysis with examples from large HEP packages. In *Proceedings of Computing in High Energy and Nuclear Physics, 2001*, 2001.
28. Jan Vitek and Boris Bokowski. Confined types. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–96, New York, NY, USA, 1999. ACM Press.
29. Microsoft Visual Studio 2005. http://msdn.microsoft.com/vstudio/.