KOTEK: Clustering Of The Enterprise Code

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

[1] ComArch SA
ul. Leśna 2
02-844 Warsaw
Poland

[1] Institute of Informatics*
Warsaw University
ul. Banacha 2
02-097 Warsaw
Poland

Abstract Development of large code bases is extremely difficult. The main cause of this situation is that the internal dependencies in a large code body become unwieldy in management. This calls for methods and tools that support software development managers in maintaining properly ordered connections within the source code. We propose a method and a static module system KOTEK to facilitate high and medium level management of such source code dependencies. The system enforces all dependencies to be clearly declared. Since KOTEK is also a build system, it automatically enforces these declarations to be up-to-date. Moreover, KOTEK allows advanced software engineering constructions like parametrisation of large code fragments with respect to some functionality.

1 Introduction

Big source code bases are extremely difficult to develop and maintain. Thus, a proper management of the code is needed [PC90]. There are various ways to organise the code. In object-oriented languages, the most basic ones are objects (or classes). The objects or classes are usually considered as low-level units, though, so they are grouped in components, packages or modules.

The power of the organisation mechanism depends on the way the grouping affects the code and is imposed on the code. For instance the tools which are based on UML or Semantic Web ontologies provide grouping in the design stage of software production but are weakly enforced in the development and maintenance stages. Moreover, they do not encourage comprehensive arrangement of construction blocks and so complicated diagrams are commonly encountered.

Moreover, the flexibility of these design standards and programming language grouping constructs like packages make it easy to build circular dependencies. The experience in software development shows that circular dependencies cause problems [SM03,Fow01] so the DAG-based coding pattern occurs often in project design guidelines [Mar02,Kno01,Com05]. Cyclic dependencies are regarded as a

* This work was partly supported by KBN grant 3 T11C 002 27.

strong factor in measures of code complexity [TT01] especially when maintainability of the code is of the main interest [Jun02]. Moreover, the presentation of code dependencies in form of a DAG has already been used in the context of support for maintainability [BR01].

We propose a system KOTEK which assists in maintaining the code structure and in organising knowledge within a software project. It ensures the match between the description of the organisation and the code since it is a tool that builds the final application. All kinds of dependencies are based on the tree or DAG structure here. Moreover, we impose the rule that each component may consist of at most seven items [Mil56,Dou02].

The basic unit of code organisation in KOTEK is called a module. KOTEK has two perspectives of code organisation. They correspond to a different basic source code organisation activities. The first one, *vertical*, allows to abstract knowledge about the modelled fragment of the real world. The second one, *horizontal*, describes functional dependencies between the abstracted notions. This division separates two basic modes of thinking. The first one focuses on the internal structure of the defined computing notion, and the second one focuses on the relations with other pieces of the software. Other Java module systems did not consider explicitly this code organisation facets [CBGM03,IT02,AZ01].

As KOTEK is a build tool, it is related to `ant`, `make` and `maven`. The main difference is that these tools operate on files only while KOTEK performs semantical checks. Thus, it gives an additional control power for a code manager.

2 Gentle introduction to KOTEK

In this section we describe the most important aspects of using the KOTEK tool through consecutive refactoring of an example of a simple database client application.

KOTEK builds the application making sure that all dependencies are declared and that their structure follows the structure described in Sect. 1. The invocation of KOTEK in the root directory of the project, makes it recursively build all components and combine them (link) into the resulting object file. The order of the building process and dependencies between modules must be described in the file `.kotek`, which is located in the project's root directory.

The first example. In this section we use the simplest version of our sample application. Its main `.kotek` file can be seen in Fig. 1 left.

The first line of this file says that in order to build our project, one needs external libraries `JDBC` and `Swing`. Next, one has to build the module `DataModel` using `JDBC`, the module `UI` using `Swing`, and `Logic` using `Swing` together with just built `DataModel` and `UI`. The final *product* of our code is the module `Logic`, which provides a class with the main method.

Apart from being a building instruction, the `.kotek` file provides an overview of the main dependencies of the project which helps in understanding of the code.

Since `DataModel`, `UI` and `Logic` may be large pieces of code, they can also be divided into submodules and KOTEK can be used to manage the order of

```
uses JDBC Swing

build DataModel: JDBC
build UI: Swing
build Logic: DataModel Swing UI

return Logic
```

```
uses DataModel Swing UI

build DataManip: DataModel
build UILogic:
       DataModel DataManip Swing UI
build App: Swing UILogic

return App
```

Figure 1. Files `Root/.kotek` and `Root/Logic/.kotek` of the sample application.

their building and their dependencies. We assume that larger modules lie in the corresponding subdirectories and each subdirectory contains the local `.kotek` file. For example, in the `Logic` directory, this file may look like in Fig. 1 right. The `Logic` component contains 3 sub-modules: `DataManip`, `UILogic` and `App`.

Note that our modules `DataModel` and `UI` are treated inside `Logic` as external ones and the implementation of `Logic` has no access to their internal details.

The hierarchical structure of `.kotek` files permits a person who wants to learn the code (e.g. a new developer) to read it in a needed level of details and only in the branches that are interesting at the moment.

Abstraction and programming with variants. Sometimes, almost identical code is used in several places of the whole project. This code must be placed in a separate organisational unit. This is done by abstraction. As the way the code is used in different places may differ, it is useful to have more than one run-time component derived from a single piece of the source code (for example, it is the case when one wants to provide several versions of the application, for different graphical environments). In KOTEK, such multiple *products* of a single piece of the source code are called *views*. Each view may have different dependencies, as it is the case in the final version of our example in Fig. 2.

In the example, we replace a single `UI` module from Fig. 1 by two modules: `UICommon` and `UIJ2SE`. The first one provides only the abstract window interface used in our application. The interface can be understood as a Java package containing only class interfaces. The second module, `UIJ2SE`, provides the implementation of the abstract interface, based on `Swing`. Both modules are then passed on to the `Logic` component.

Moreover, we add a `.NET` frontend based on `Forms` to our application. Multiple views are used in two components in this version of our application. First, the two related modules, `UICommon` and `UIJ2SE`, have been joined into a bigger component `UI`, which got the third sub-module `Dotnet`, implementing the `Common` contract using `Forms`. The code that is the same in `J2SE` and `Dotnet` has been extracted to the module `Utils`. Apart from the latter, the other three modules are exported as three products of the `UI` components.

The `Logic` component changed accordingly: the `UILogic` is based on the common interface as before, and there are now two final modules `AppJ2SE` and `AppDotnet`, depending on suitable graphic toolkits and instantiated `UILogic`.

In the main `.kotek` file the dependencies of the modules `UI` and `Logic` are listed twice. The first time, in the `absbuild` command (a shorthand for *abstract* build), which causes a recursive build of the component but without the final linking phase. The second time, they are used as arguments of the `create` command which performs the linking. Note that by analysing the dependencies in the `.kotek` file alone it can be seen that the `J2SE` version of the application does not depend on `Forms` and that the `Dotnet` version does not depend on `Swing`.

3 Technical overview

The KOTEK tool is not limited to a particular programming language, even though we specifically thought of needs of large Java projects while designing it. It consists of four language layers, two of which are intermediate and hence practically invisible for users. These are (N) native (object files) e.g. Java, (M) low-level abstract (`.ms` files) invisible, (L) linker instructions (`.cc` and `.ld` files) invisible, (K) `.kotek` files.

The (M) level provides the detailed description of the interfaces of modules that define dependencies outlined in Sect. 1. The description language is independent from the source language. The (L) level is the list of instructions for the linker, connecting formal parameters of modules to their actual dependencies. Formally, it just binds to new names the applications of functions to arguments.

The input for KOTEK is the list of object files of the module's dependencies, together with their contracts (`.ms` files) and type sharing information between the dependencies (in the `.cc` files). The output is the final object code and a pertinent `.ms` file with interface specification. The information is processed between the four layers as follows:

native reader compiles source code only modules (without `.kotek` files) and derives `.ms` files for them,

native linker links the object code of the submodules into the object code of the module, according to the linker instructions in the `.ld` file,

linker links the `.ms` files of the submodules into the `.ms` file of the module, according to the linker instructions in the `.ld` file,

KOTEK main transforms the input `.cc` file and `.kotek` into `.cc` of each submodule, runs KOTEK recursively, then builds the `.ld` file and calls the linkers to build the resulting `.ms` and object files.

Note that only the two first transformations are language specific and hence in order to use the KOTEK method for other programming languages, one only has to provide these two. Note also that it is possible to implement only part of the functionality for a given programming language (for example without abstract modules and views) and still benefit from other advantages of KOTEK.

4 Conclusions

Prototype. The KOTEK method is actively used in ComArch research laboratory to manage an actively developed Ocean GenRap business intelligence platform

File: Root/.kotek

```
uses JDBC Swing Forms

build DataModel: JDBC

absbuild UI: Swing Forms

let UICommon=UI create Common ()
let UIJ2SE=UI create J2SE (Swing)
let UIDotnet=UI create Dotnet (Forms)

absbuild Logic: DataModel UICommon Swing UIJ2SE Forms UIDotnet

let AppJ2SE=Logic create AppJ2SE(DataModel UICommon Swing UIJ2SE)
let AppDotnet=Logic create AppDotnet(DataModel UICommon Forms UIDotnet)

return AppJ2SE AppDotnet
```

File: Root/UI/.kotek

```
uses Swing Forms

build Common:
build Utils:
build J2SE: Utils Swing
build Dotnet: Utils Forms

return Common J2SE Dotnet
```

File: Root/Logic/.kotek

```
uses DataModel UICommon Swing UIJ2SE Forms UIDotnet

build DataManip: DataModel
build UILogic: DataModel DataManip param(UICommon contr Common) as UI
let UILogicJ2SE=UILogic(UIJ2SE as UI)
let UILogicDotnet=UILogic(UIDotnet as UI)
build AppJ2SE: Swing UILogicJ2SE
build AppDotnet: Forms UILogicDotnet

return AppJ2SE AppDotnet
```

Figure 2. Two versions of UI.

prototype, the project of about 210 000 lines of mostly Java code. It is managed using about a 100 `.kotek` files of total length of 1700 lines, so less than 1%. Even though not all features presented in the paper are implemented in the KOTEK prototype, it already proves to be a great help in learning the code by new developers and in managing the code by component owners.

References

[AZ01] 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.

[BR01] 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.

[CBGM03] 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.

[Com05] Compuware. Optimaladvisor supersedes the Package Structure Analysis Tool. Technical report, JavaCentral, 2005.

[Dou02] Jean-Luc Doumont. Magical Numbers: The Seven-Plus-or-Minus-Two Myth. *IEEE Transactions on Professional Communication*, 45(2), June 2002.

[Fow01] Martin Fowler. Reducing Coupling. *IEEE Software*, July/August 2001.

[IT02] 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.

[Jun02] 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.

[Kno01] Kirk Knoernschild. Acyclic Dependencies Principle. Technical report, Object Mentor, Inc., 2001.

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[Mil56] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psycological Review*, 63:81–97, 1956.

[PC90] 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.

[SM03] Barry Searle and Ellen McKay. Circular Project Dependencies in WebSphere Studio. *developerWorks, IBM*, 2003.

[TT01] 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.