Reengineering towards Distributed Applications

Ansgar Radermacher

Siemens Corporate Technology
Software & Engineering (ZT SE 2),
Munich

ansgar.radermacher@mchp.siemens.de

Andy Schürr

Institute for Software Technology
University of the Federal Armed Forces,
Munich

andy.schuerr@unibw-muenchen.de

July 3, 2000

Abstract

Consider the task of distributing an existing monolithic application: its code and architecture have to be changed. This task is usually supported by re-engineering tools. However, these tools do not support the specific transformations required during the transition towards a distributed application. In this paper, we will present such a methodology and a tool that performs this transition by a sequence of interactive and automatic transformations.

1 Introduction

The work sketched in the sequel originates from a project of the University of Aachen and the Aachener and Münchener insurance group and the GEZ. Both companies have large, monolithic programs written in Cobol that should be reorganized. A part of the application logic should move to the client machines or additional server machines.

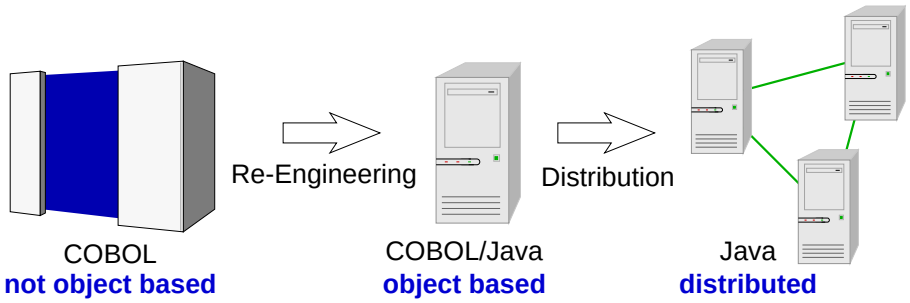

Figure 1: Towards Distributed Applications

As shown in Fig. 1, the underlying task is split into two subtasks. The first comprises the transformation of existing code towards object based code. This is chiefly captured by the work of Cremer [Cre99]. Here, we focus on the second subtask, the transition towards

distributed applications. Therefore, we first sketch the problem of creating distributed applications in general.

1.1 Distributed Applications

The development of distributed applications is alleviated by tools that offer a suitable *abstraction* of the communication between two disjoint program parts. This abstraction is close to the typical means of exchanging information in an imperative programming language: a procedure call or a method invocation. These techniques are called *middleware* because they bridge the abstractions of the operating system's API with that of the programming language. Well-known middleware techniques comprise the OMG standard CORBA [OMG98], Microsoft's DCOM [Ses97] and Java's RMI [Dow98].

The use of such techniques eases the development of a distributed application considerably. But there are deficiencies:

- Middleware-specific code can not be isolated in a single module, it is spread throughout the application code. This makes it difficult to adapt the program to another middleware or a new distribution structure. Thus, programs developed in this way might become future *legacy* systems.
- Middleware has certain restrictions; it is for example not possible to instantiate an object in a remote address space. The developer has to ensure that an object is created in the intended address space.
- There is no visual specification of the distribution structure.

In order to overcome these problems there are additional tools on top of the middleware that allow for the specification of the distribution structure and the components inside a distributed systems. But almost all of these approaches use a particular language for the specification of the components that form the distributed application and can not deal with existing application code –a major requirement of the project with the AM group and the GEZ. The *a-posteriori* distribution of an existing program is not captured with the exception of general case tools offering minimal distribution support (for example the generation of a CORBA interface definition).

From a re-engineering point of view, the distribution of an existing application program, comprises a sequence of suitable transformation steps, in order to achieve conformance with middleware prerequisites. Because the application should not contain middleware specific code (in order to be able to exchange the middleware), it is desirable that a part of the necessary transformation steps are performed automatically and *generate* the distributed application. The starting point for such a generation is a version that is already middleware-aware (i.e. conforms to its restrictions) but does not contain the middleware specific code.

We will now discuss the implementation of the transformation steps.

2 Tool Support

A major part of the implementation of our approach employs the **pro**grammed **g**raph **rew**riting **s**ystem PROGRES [SWZ99]. It is used to store and manipulate UML [RO$^+$99]

like class diagrams enriched by distribution information. The main data structure in PROGRES is a (typed) *graph*. The structure of such a graph is defined in the PROGRES graph schema.

Our specification represents model elements of the class diagram –for instance classes, interfaces and methods– as nodes, relationships as an edge-node-edge combination. Thus, the schema closely resembles the meta model of UML's class diagrams.

Graph tests find places in a graph that conform to an explicitly specified subgraph pattern. *Graph rewrite rules* transform a graph by replacing the subgraph given in the left-hand side of the rule with the subgraph specified in the right-hand side. Path expressions[1] ease the navigation through the graph. In the class diagram specification, path expressions are defined for common relationships, e.g. inheritance and dependencies, and attachment information. The latter is related to distribution extensions (see below).

If a graph rewrite rule manipulates the graph and thus the class diagram of the underlying application, the application's source code is transformed *simultaneously* by a tool implemented in Java. This tool[2] also analyzes source code and derives the necessary information to build up a class diagram (reverse engineering [CC90]).

Fig. 2 shows a screen dump of a prototype we have developed in this project. Diamonds denote containment relationships, solid arrows denote invocations, and dashed arrows denote instantiations. Some nodes carry information from an automatic analysis implemented via graph tests (for example a problematic remote invocation or remote instantiation).

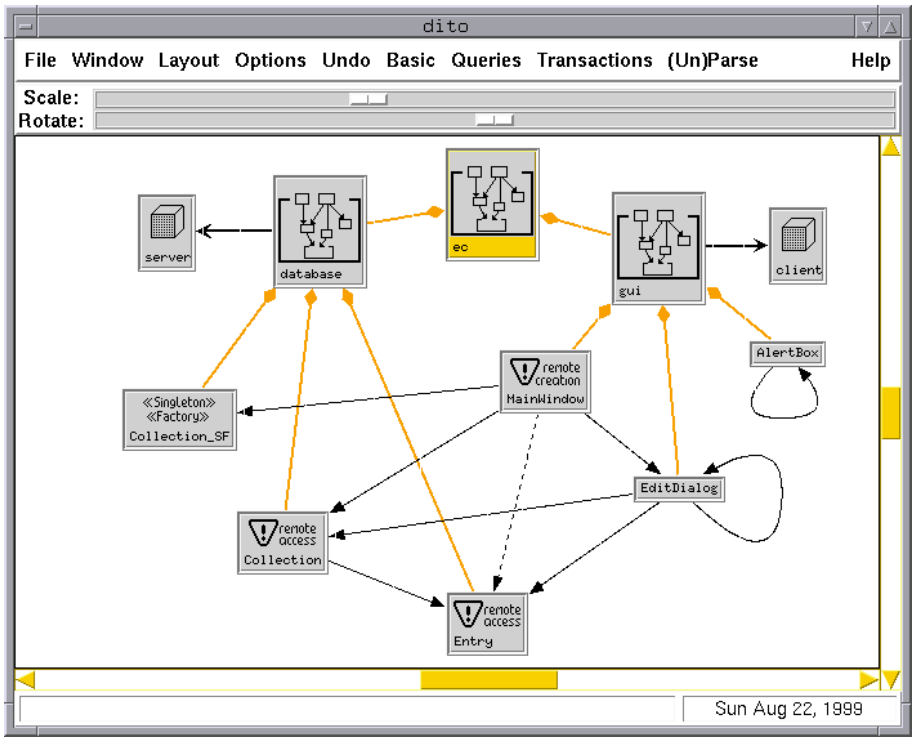

Figure 2: Prototype of a Distribution Tool

[1] A path expression behaves like an edge, but it is not materialized, it is computed.

[2] Further information about this tool can be found in [Rad00] and in http://ist.unibw-muenchen.de/People/ansgar/dito/.

2.1 Distribution Patterns

We will present the context and motivation of a necessary transformation using a schema known from *design patterns* [GHJV95]. Design patterns describe a problem together with a solution that has proven to be useful with respect to a certain goal. The approach described here does not handle all types of patterns, it is well-suited for those that propose a certain static structure which can be described by means of a class diagram.

The schema is outlined in the following:

Context and Problem This section describes a problematic architectural structure (in a given context, for instance distribution). Besides this informal description, a graph test provides a means to *detect* this structure.

Solution and Structure The paragraph outlines the general idea of a pattern and shortly sketches its advantages and disadvantages. The structure of the pattern is usually shown in form of a class diagram and a graph test[3].

Getting There This paragraph is not found in the standard literature dealing with patterns: If a developer starts writing a program from scratch, it is relatively easy to conform to the structure proposed by a pattern. This task is certainly non-trivial, if a program already exists. We describe the necessary transformation towards conformance to the pattern by means of a graph rewrite rule.

2.2 Direct Instantiation

2.2.1 Context and Problem

Object-oriented programming languages provide a primitive to instantiate a class, often called new. Of course, the language primitive only creates instances in the local address space (and the class has to be available in the partition). In a distributed system, it is often necessary to create objects residing in other address spaces. In these cases, the instantion via new has to be replaced by an additional indirection.

Fig. 3 shows a PROGRES test that detects a potential instantiation of a remote class. The rule contains three nodes representing model elements of the underlying class diagram and pathes representing relationships between these.

The test matches a situation in which a class (‘2) is instantiated by another class (‘1) and a partition (‘3) to which the former but not the latter (denoted by the “crossed out” path attached) is attached. If it matches, it returns the node ‘1.

2.2.2 Solution and Structure – Abstract/Generic Factory

The *abstract factory* [GHJV95] pattern enables the transparent use of different implementation variants which implement a common interface. The decision which of the variants is instantiated is encapsulated inside the factory. The factory is a normal object that offers the service to instantiate certain classes via a method, called for example createInstance().

[3] Class diagram and graph test are almost equivalent.

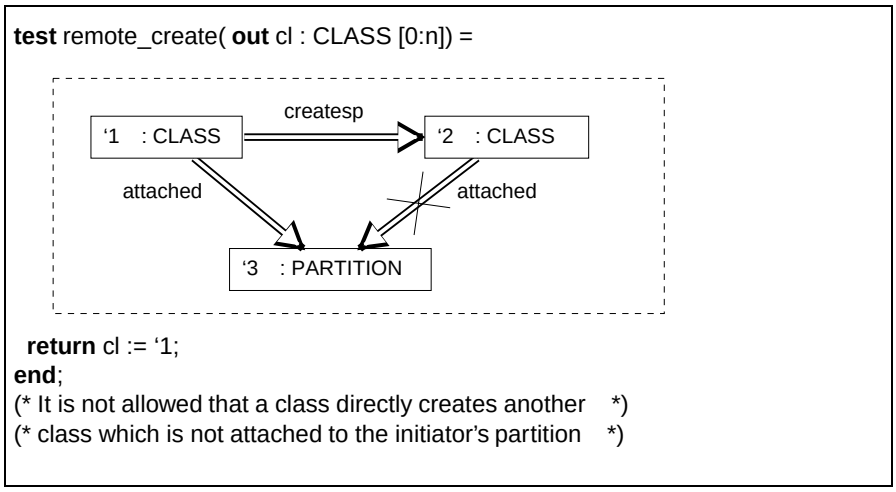

Figure 3: PROGRES Test: Find a Direct Instantiation of a Remote Class

In the context of distributed systems two problems are addressed: (1) The factory lives in the address space, in which the desired object should be instantiated. Thus, objects in other partitions can trigger the instantiation of a remote object by invoking the createInstance method of the factory. (2) It remains transparent for the callers, whether the result of an instantiation is a stub or an original implementation.

In order to invoke an operation of the factory object, callers have to get a handle or object reference. In our pattern, we require that the factory is a singleton, i.e. there is exactly one instance of this class. The *singleton* pattern which is discussed in [GHJV95], provides a common way to access a unique instance.

2.2.3 Getting There

After the recognition via the query in Fig. 3, two tasks have to be done: (1) create a suitable factory, (2) change all applied occurrences of the direct instantiation via new. The first task is trivial and not shown here. The second requires the replacement of all new operations (except the one in the factory) by an invocation of the factory. The latter is shown in Fig. 4: A create relation to a class is replaced by a call relationship[4] to the factory. The folding statement allows that factory and created class are identical. This situation can occur if an object supplies its own, static factory method.

3 Related Work

The transition from an object based towards a distributed application comprises several different aspects. An ideal tool (or a set of integrated tools) has to support all these aspects.

Roundtrip engineering This task comprises the reverse engineering of an architecture from an existing source, performing changes on the architecture and propagating these changes back to the source again. This task is supported by most UML tools, for example Rational Rose [Rat99].

[4]We use the convention to draw relationship nodes as squares.

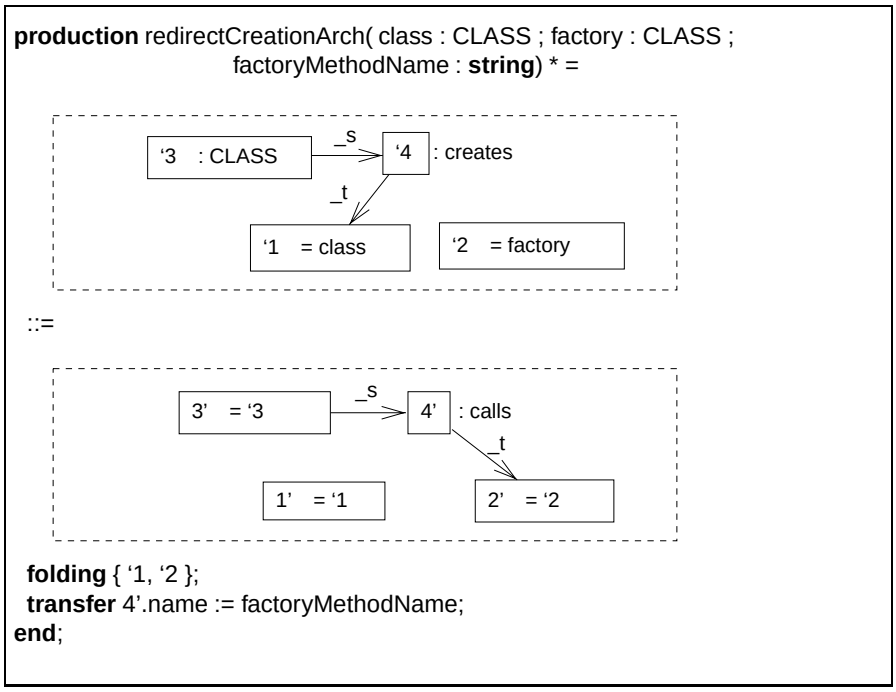

Figure 4: PROGRES Production: Instantion ⇒ Invocation of the getInstance Method

Domain specific analysis (reverse engineering) In order to support the transition towards a distributed application, it is necessary to detect a violation of distribution restrictions. This is not supported by most commercial OO tools. EBERT et. al have done work in the area of graph based analysis of programs [EKW96]. ZÜNDORF and JAHNKE [JZ97] use a fuzzy reasoning net to detect "bad code" in existing programs. DEAN and CORDY [DC95] use an architecture like language to represent system structures. They are able to *characterize* systems according to their conformance with a given *architectural style*, e.g. a layered architecture or a pipe-filter. DEVANBU developed a source code analysis framework [Dev99]: an analyzer can be generated from a compact analysis specification.

Semantic preserving transformations (re-engineering) UML CASE tools allow for basic changes, for instance to change the name of a method. Usually, they do not automatically preserve a consistent state, e.g. automatically change all places in the source code which invoke the renamed method.

This task is typically supported by re-engineering tools. TXL, developed by CORDY [CCH95], is a powerful transformation tool, yet it is not possible to determine for example the return type of a method whose class is not defined within the current source file – an analysis information needed by some transformations.

Recently, FOWLER [Fow99] uses the notion *refactoring* to describe the small transformations that restructure a (micro) architecture. MÉTAYER [Mét96] also uses graph grammars to describes the *evolution* of an application by graph rewrite rules. However, there is no tool supporting MÉTAYER'S approach.

Pattern oriented transformations A pattern oriented transformation tool has been developed by FLORIJN et al. [FMW97] at the university of Utrecht. Their system allows

to bind existing classes to a role in a pattern or to create new classes by instantiating a pattern. It operates in three ways: (1) instantiate a pattern, i.e. generate program skeletons (2) bind classes to roles in a pattern, and (3) check conformance to a pattern.

4 Summary

The transition from a monolithic to a distributed application on top a a standardized middleware can be tackled by a sequence of transformations. The necessary transformations are selected interactively in a preparation phase and applied automatically in a generation phase. Text and graph transformations are coupled: they are executed simultaneously on the architecture graph and on the source code.

Graph techniques allow for a suitable specification of analysis (graph tests) and transformation rules. A graph replacement rule combines both anti pattern (as introduced by Brown et al. [BMIM98]) and pattern on left and right-hand side, respectively.

The use of graph specifications within the PROGRES environment has two advantages: (1) they provide an intuitive, yet well defined means to specify architectural transformations, and (2) it is simple to adapt the rules to new requirements and generate a new prototype that can execute the graph rules.

References

[AM97] Mehmet Akşit and Satoshi Matsuoka, editors. *ECOOP'97, 11th European Conference on Object Oriented Programming, Jyväskylä, Finland*. Springer, Heidelberg, June 1997.

[Arn93] Robert S. Arnold, editor. *Software Reengineering*. IEEE Computer Society Press, 1993.

[BMIM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns*. John Wiley & Sons, 1998.

[CC90] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomoy. In Arnold [Arn93], chapter Software Reengineering: Context and Definitions, pages 54–58.

[CCH95] J.R. Cordy, I.H. Carmichael, and R. Halliday. *The TXL Programming Language (Version 8)*. Legasys Corp., Kingston, 1995.

[Cre99] Katja Cremer. *Werkzeuge zum Reengineering*. PhD thesis, RWTH Aachen, Department of Computer Science III, March 1999.

[DC95] Thomas R. Dean and James R. Cordy. A Syntactic Theory of Software Architecture. *IEEE Transactions on Software Engineering*, 21(4):302–313, 1995.

[Dev99] Premkumar T. Devanbu. GENOA - A Customizable, front-end Retargetable-Source Code Analysis Framework. *Transactions on Software Engineering and Methodology*, 8(2):177–212, April 1999.

[Dow98] T.B. Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, February 1998.

[EKW96] J. Ebert, M. Kamp, and A. Winter. Generic Support for Understanding Heterogeneous Software. Fachbericht Informatik, Universität Koblenz-Landau, Fachbereich Informatik, March 1996.

[FMW97] G. Florijn, M. Meijers, and P. Winsen. Tool Support for Object-Oriented Pattern. In Akşit and Matsuoka [AM97], pages 472–495.

[Fow99] Martin Fowler, editor. *Refactoring – Improving the Design of Existing Code*. Addison Wesley, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, Addison-Wesley, 1995.

[JZ97] Jens Jahnke and Albert Zündorf. Rewriting poor Design Patterns by good Design Patterns. In S. Demeyer and H. Gall, editors, *Proceedings ESEC/FSE'97, Workshop on Object-Oriented Reengineering*. University of Vienna, Technical Report TUV-1841-97-10, September 1997.

[Mét96] Daniel Le Métayer. Software architecture styles as graph grammars. In David Garlan, editor, *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–23, San Francisco, California, October 1996.

[OMG98] OMG (Object Management Group). The CORBA/IIOP 2.2 Specification. *OMG Document formal/98-02-01*, 1998.

[Rad00] Ansgar Radermacher. *Tool Support for the Distribution of Object-Based Applications*. PhD thesis, RWTH Aachen, Department of Computer Science III, March 2000. to appear.

[Rat99] Rational Software. Rational Rose. http://www.rational.com/products/rose, 1999.

[RO+99] Rational Software, OMG, et al. Unified Modelling Language, V 1.3 alpha 2. http://www.rational.com/uml, 1999.

[Roz99] Grzegorz Rozenberg, editor. *Handbook on Graph Grammars and Computing by Graph Transformation: Applications*, volume 2. World Scientific, Singapore, 1999.

[Ses97] Roger Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, New York, 1997.

[SWZ99] Andy Schürr, Andreas J. Winter, and Albert Zündorf. *The PROGRES Approach: Language and Enviroment*, chapter 13, pages 487–550. Volume 2 of Rozenberg [Roz99], 1999.