volume 32 number 6 june 1997

BB-Clark

ACM SIGPLAN notices

A Monthly Publication of the Special Interest Group on Programming Languages

Activities
The SIG Discretionary Fund: Role and Process *by Lori Pollock* 1
Conference Corner
Calendar 2
Language Tips
Implementation of a Database Factory *by Asokan R. Selvaraj and Debasish Ghosh* 14
Forth Report
Forth as a Robotics Language *by Paul Frenger* 19
Garbage In/Garbage Out
COMFY -- A Comfortable Set of Control Primitives for Machine Language Programming *by Henry G. Baker* 23
Curricular Patterns
The 'Java in the Computing Curriculum Conference' *by Fintan Culwin* 28
Technical Correspondence
Considerations in Developing a Formally-Based Visual Programming Language Reference Manual: A Case Study on the SLAM II Language *by Debbie K. Carter and Albert D. Baker* 34
Computational Steering Annotated Bibliography *by J. S. Vetter* 40
A Functional Approach to Type Constraints of Generic Definitions *by Myung Ho Kim* 45
Garment: A Mechanism for Abstraction and Encapsulation of Languages *by Zhang Naixiao, Zheng Hongjun, and Qui Zongyan* 53
Consummating Virtuality to Support More Polymorphism in C++ *by Wen-Ke Chen, Jia-Su Sun, and Zhi-Min Tang* 61
The F Programming Language *by Ralph Frisbie, Richard Hendrickson, and Michael Metcalf* 69
How to Achieve Modularity in Distributed Object Allocation *by Franco Zambonelli* 75

ACM SIGPLAN Notices

Table of Contents

Editor: *A. Michael Berman, Academic Computing, Rowan University, Glassboro, NJ 08028 USA*; *berman@rowan.edu*

Activities

The SIG Discretionary Fund: Role and Process *by Lori Pollock* 1

Conference Corner

Calendar 2

Language Tips

Implementation of a Database Factory *by Asokan R. Selvaraj and Debasish Ghosh* 14

Forth Report

Forth as a Robotics Language *by Paul Frenger* 19

Garbage In/Garbage Out

COMFY — A Comfortable Set of Control Primitives for Machine Language Programming *by Henry G. Baker* 23

Curricular Patterns

The 'Java in the Computing Curriculum Conference' *by Fintan Culwin* 28

Technical Correspondence

Considerations in Developing a Formally-Based Visual Programming Language Reference Manual: A Case Study on the SLAM II Language *Debbie K. Carter and Albert D. Baker* 34

Computational Steering Annotated Bibliography *J. S. Vetter* 40

A Functional Approach to Type Constraints of Generic Definitions *Myung Ho Kim* 45

Garment: A Mechanism for Abstraction and Encapsulation of Languages *Zhang Naixiao, Zheng Hongjun, and Qiu Zongyan* 53

Consummating Virtuality to Support More Polymorphism in C++ *Wen-Ke Chen, Jia-Su Sun, and Zhi-Min Tang* 61

The F Programming Language *Ralph Frisbie, Richard Hendrickson, and Michael Metcalf* 69

How to Achieve Modularity in Distributed Object Allocation *Franco Zambonelli* 75

ACM SIGPLAN Notices
A monthly publication of ACM SIGPLAN

Publications Office
ACM, 1515 Broadway,
New York, New York 10036-5701
USA
1-212-869-7440 FAX +1-212-302-9618

***Editor:** A. Michael Berman*

Contributing Editors:
Ron K. Cytron, Preston Briggs, G. Bowden Wise, Barbara G. Ryder, Seth Bergmann, Dan Yellin, Henry G. Baker, and Philip Wadler

ACM SIGPLAN Notices is an informal monthly publication of the Special Interest Group on Programming Languages (SIGPLAN) of ACM.

Membership in SIGPLAN is open to ACM Members or associate members for $30.00 per year and to ACM Student Members for $10.00 per year. Non-ACM members may join for $60.00 per year. All SIGPLAN members receive ACM SIGPLAN Notices, are given discounts at SIGPLAN-sponsored meetings and may vote in the Group's biennial elections. ACM members of SIGPLAN may serve as officers of the group.

Institutional or Library subscriptions to ACM SIGPLAN Notices are available for $57 per year, and the regular back issues of the Notices may be purchased for $7 per copy from ACM Headquarters.

Requests for reprints, copies of reports, or references should be sent directly to authors.

change of address: acmcoa@acm.org
Members Services Information:
acmhelp@acm.org or +1-212-626-0500

ACM SIGPLAN Notices (ISSN 0362-1340) is published monthly by ACM, 1515 Broadway, NY, NY 10036. The basic annual subscription price is $30.00 for ACM Members. Periodicals postage paid at NY, NY 10001 and at additional mailing offices.
POSTMASTER: Send change of address to ACM SIGPLAN NOTICES, ACM, 1515 Broadway, NY, NY 10036.

ACM SIGPLAN Language Tips

C++ Toolbox

Editor: *G. Bowden Wise, Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180; wiseb@cs.rpi.edu*

Implementation of a Database Factory

Asokan R. Selvaraj and Debasish Ghosh

Abstract

Object oriented software systems that utilize relational databases for data-store have to deal with the problem of interfacing to the relational data. This aspect of the software is more relevant to the solution domain rather than the problem domain i.e., the business requirements of an application do not dictate that there be a mechanism that allows the application to store and retrieve relational data. Hence, de-coupling the application from the database and its interface is relevant from the perspective of portability of the application to other kinds of databases. For example, it is conceivable the application may be required to work with relational databases from different vendors. This article shows an adaptation of the Factory Method Pattern and the Abstract Factory Pattern [1] as a generic solution to the problem of de-coupling application code from the underlying database and its associated interface mechanisms.

The Problem

In designing object oriented software systems, there is a need to separate application logic (hereafter referred to as the application) from functionality that interfaces the application to the database. From an architectural standpoint it is necessary to have this de-coupling to minimize/eliminate the impact of changes in the database products or the implementation of the database interface.

A typical application may require a family of utility classes that allow it to interface to the database e.g., `DbTransaction`, `DbQuery`, `DbReport`, etc. Once this set of classes and their responsibilities are established, these classes can be specialized differently to work with different database products. There are two factors to consider in organizing such classes in hierarchies.

1. Different implementations of these utility classes may be required for different database products.
2. Parts of the implementation of these classes may be common to most database products and can be factored out as base utility classes (default implementation – see Figure 1). Different versions of the utility classes may then be inherited from the bases to different levels of specialization.

Given the above organization of hierarchies, the application would have to be aware of which database product (ORACLE, INFORMIX, etc.) it is working with, and for that product, whether the utility class it requires has been sub-typed or not. As an example, consider the following two scenarios (see Figure 1):

1. To support ORACLE, the utility class `DbTransaction` has been sub-typed as `DbOracleTransaction`, while the standard implementation of `DbQuery` is sufficient.
2. To support INFORMIX, the utility class `DbQuery` has been sub-typed as `DbInfQuery`, while the standard implementation of `DbTransaction` is sufficient.

Listing 1 shows how the application must determine which sub-type of the utility classes must be instantiated based on which database engine (ORACLE or INFORMIX) is desired. Note that this is resolved at compile-time through the use of conditional compilation switches.

The most noticeable deficiency in the above approach is the strong physical coupling between the application code and the database interface. Arising out of this coupling, are problems like a change in one sub-system requiring the other to be recompiled and re-tested. In a typical application, code utilizing the database classes will be distributed over many parts of the system and the consequent ripple effect could be considerable.

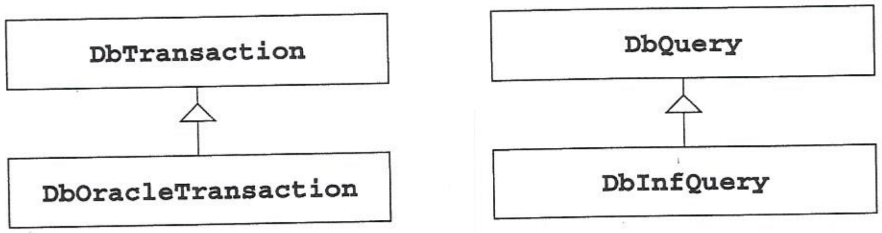

Figure 1: Sub-typing the utility classes for Oracle and Informix.

The Factory Approach

The Database Factory design is based on the Factory Method and the Abstract Factory patterns [1]. In this approach, described below, the Database Factory subsystem assumes the responsibility of resolving which family of utility classes to use (i.e., ORACLE or INFORMIX), identifying the correct sub-types of the required objects, and creating the objects for its clients.

For simplicity, the application code that uses the Database Factory sub-system and the database utility classes is indicated as class `Application` in Figure 2. The `Application` code uses classes `DbFactory` and `DbFactoryUtils` to obtain database utility objects of the appropriate sub-type. These two classes serve as the interface to the Database Factory subsystem.

Figure 2 shows the main components in the Database Factory subsystem. The `static` method

```
DbFactory*
    DbFactoryUtils::getFactory()
```

is the *factory method* (as in the Factory Method pattern). Assuming the system is configured to use INFORMIX, this function will simply create a new `DbInfFactory` object and return it to the `Application`. The `Application` is unaware of the exact sub-type of the `DbFactory` object that it received (nor does it care) and manipulates the returned object only through the `DbFactory` abstract interface. This mechanism reduces any coupling between the `Application` and the kind of database used, to a link time dependency. Linking in a different implementation of the `getFactory()` method is all that is required to change the database the `Application` works with. This is a significant saving in comparison to the solution that uses conditional compilation based on compile-time defined flags (see Listing 1). Of course, the price we pay for this solution is the function call overhead at run-time for every call to the `getFactory()` method. (Note that an attempt to eliminate this function call overhead by inlining the `getFactory()` method will defeat the savings in recompilation when the system is to be regenerated to use a different database. This is because, when inlined, the code for the `getFactory()` method would be physically inserted in the `Application` wherever the `getFactory()` method is invoked, and would now have to be replaced).

The `DbFactory` class (implemented as an abstract interface – see Listing 2), through another level of indirection, eliminates in the `Application` code, knowledge of the various levels to which the database utility classes have been specialized for different databases. This was not possible with the earlier approach. Consequently, in the earlier design, the `Application` code would have to be modified, compiled and tested if it is decided to specialize `DbTransaction` to `DbInfTransaction` in the INFORMIX implementation (or `DbQuery` to `DbOracleQuery` in the ORACLE implementation).

In the Abstract Factory pattern based approach, the `DbFactory` class supports pure virtual methods of the `makeObject` nature for every database utility class supported by the factory mechanism. For example, to support the code fragment above, the `DbFactory` will provide methods (interfaces):

```
DbQuery* DbFactory::makeDbQuery()
```

and

```
DbTransaction*
  DbFactory::makeDbTransaction()
```

The `DbDefaultFactory` class inherits these interfaces from the `DbFactory` class and provides the standard implementations of the methods `makeDbQuery()` and `makeDbTransaction()`, which return the standard versions of the classes `DbQuery` and `DbTransaction` respectively (see Listing 3).

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.