Preliminary Case Study on Software Reuse with Object Persistency Framework

Hiroshi Suganuma  Norikazu Kijima  Takeshi Nii
Hitachi Software Engineering Co., Ltd.
Yokohama 231-8475, Japan
{suga, kijima, nii}@aqu.hitachi-sk.co.jp

Abstract

Software reuse is considered one of the most effective ways of achieving higher productivity improvement. In this paper we propose an object persistency reuse framework to overcome the issues such as database connection handling problems, time consuming unit testing for database access modules, and larger maintenance cost due to the frequent changes in the target applications specifications.

The OPRF consists of (1)SQL execution component, which provides the database connection and data conversion, (2)database connection component, which provides a mechanism to attach and detach to/from a web application server without source code changes, and (3)table access source code generator, which generates the table access components.

We applied our technique to the several actual web application development projects and kept the records on their work time. The study showed that 20 percent, 17 percent, and 30 percent cost reductions for implementation, unit testing/debugging, and maintenance respectively.

Keywords: reuse, framework, component, database, object, persistency

1 Introduction

Reusing existing software components at any level are considered effective in software productivity improvement and quality assurance [Barnes91]. Many practices at industries are reported successful [Morisio00][Barbier01].

From another perspective, reuse has become more important at industries due to the rapid expanding and emerging new technologies especially in Internet applications. Single engineer is no longer able to catch up with all these technologies during their own tight scheduled customer application development. Engineers should be able to facilitate other engineers practices, instead of following the same ad-hoc experiments. For more effective technical transfer, reusing components, which already contain the well-examined practices with easy-to-use interfaces, would be one of such best candidates.

In industrial environment we should always consider the balance of investment between providing components and reusing components. Providing components requires more resources than just developing working software. Thus, if the provided components were not reused more than certain amount of time, the resources for creation and maintenance are considered wasted.

Furthermore, software industries are likely to develop varieties of software such as operating systems for a number of mainframe computers, user application systems such as banking, security, personnel, inventory, accounting, insurance, etc., and embedded systems such as for PDAs and cellular phones. Due to these large scale variations, the chances for reusing components may be small unless the components are generic. But the generic components are less likely effective. And also, even if the components are domain specific and effective for some applications, due to the difference between the customer companies business models, there may be too many portions in the components that are affected by these differences.

2 Issues

To achieve our practical reuse, we started from the generic function library, and we are now in the second stage, which focuses on the object persistency. Nowadays, almost all of user applications use database for storing and retrieving data, and the ways to access are basically the same, yet object design for this part is also basically the same. Thus, it has larger coverage in many application domains.

**Redundant code for database connection handling.** Depending on the skills of programmers, database handling code such as initialization of the connections could be multiply coded with many different parameters, and/or datatype conversions between database and Java could be inconsistent.

**Frequent changes in specifications.** Due to the customers strong needs for time to market services the development cycle time becomes shorter and shorter, yet the requirements change frequently even at the later phases of development. Thus, if the transformation from the table design to the source code is straightforward, it is reasonable to provide a tool and have it done the transformation, allowing the engineers to concentrate on more important parts.

**Web application server handling.** If the target application runs on web application servers for Internet and/or intranet access, the situation becomes one more step complex due to the connection pooling mechanism that web application servers provide. Although this mechanism provides better performance for database connections, it requires careful han-
dling in its initializations. Also, since the connection mechanism is tightly coupled with the web application server, unit testing and debugging require certain amount of code modifications to run stand alone, which takes nontrivial time.

Solving these issues should virtually raise the skills of the programmers and prevent from conversion problems between datatypes, initialization problems, and time consuming test and debug problems.

3 Design architecture and implementation

To overcome the issues, we designed an Object Persistence Reuse Framework (OPRF) architecture with components and a code generator, as shown in Figure 1. Followings are the details of components and how these components solve the issues:

**SQL execution component:** Instead of having programmers to write code directly, we designed a component that accesses the database using the appropriate SQL code. SQL execution component prevents the programmers from writing the source code for connecting database and converting data types directly, which caused using uninitialized connections and/or undesirable data type matching.

**Database connection component:** It is time consuming task during the test execution to force programmers/testers to manipulate web application server, which should not be necessary to just test the unit level code. Database connection component enables the programmers and testers to debug and test the database access layer without any source code changes due to the web application server constraints, by only turning the switches on/off in the configuration file.

**Table access source code generator:** Except some of the conditions for searching, the most of the database access are likely to be the same. Thus, it is not difficult to separate the affected point, typically the “where” clause from other parts. The “where” clause may need human interactions, but the other the fewer. Table access source code generator prevents programmers from writing the source code for storing, retrieving, and searching the data directly every time the table design is changed.

**Interactions between components:** The generated table access component(c) represents the data model stored in the database as classes and objects. It receives requests from the business logic layer to perform a part of certain functionality, and sends the appropriate messages to the SQL execution component(d) to activate corresponding operations.

The SQL execution component(d), then obtains the connection object from the database connection component(f), executes the query, and then returns the result to (c). The database connection component(f) reads the configuration file at the start up and decides the behavior whether it is on the web application server or stand-alone environment.

4 Case study and evaluations

4.1 Cost reductions at design and implementation

With this technique, at the design phase, as soon as the table design is fixed, the corresponding application interface can be derived by generating the table dependent component. Based on our statistical record, we see this is equivalent to 5% cost reduction.

At the implementation phase, we observed that approximately 25% of the cost could be reduced, assuming that the cost is linearly related with the size of the project. Table 1 shows the effects of our technique toward the case study projects. PL denotes the presentation layer, BL the business logic layer, and DBA the database access layer. All the unit, except the reduction, is KLOC. \( M \) denotes the estimated original size of the DBA before OPRF application, which was calculated based on the OPRF-applied size. Note that this adjustment is necessary to see if the sizes of the generated component is adequate and is not unnecessarily large. Otherwise the reduction effect would be over-estimated.

4.2 Cost reduction at test phase

The connection component provides the automatic initialization facility and the framework for the unit test activity. Before applying this technique the test engineers needed to test the database access modules by incorporating other modules in other layers, namely that in the business logic layer and the presentation layer. Due to the increased number of unstable elements during the test, it was inadequately time consuming to just clarify if the target database layer has the problem or not. Moreover, the test execution could not be performed without using the web application server, involv-
ing deployment, restarting the web application server, GUI manipulations, and the log data analysis that the web application server recorded.

Table 2 shows the measured time for each activity during the test execution. We observed 58% cost reduction on the project A.

<table>
<thead>
<tr>
<th>no.</th>
<th>activity</th>
<th>before(min.)</th>
<th>after(min.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Find errors</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>1.1</td>
<td>clarify if in DB layer</td>
<td>(11)</td>
<td>(0)</td>
</tr>
<tr>
<td>1.2</td>
<td>investigate DB layer</td>
<td>(5)</td>
<td>(5)</td>
</tr>
<tr>
<td>2</td>
<td>Change code</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Prepare for test exec.</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3.1</td>
<td>compile</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>3.2</td>
<td>deploy</td>
<td>(1)</td>
<td>(0)</td>
</tr>
<tr>
<td>3.3</td>
<td>restart AP server</td>
<td>(2)</td>
<td>(0)</td>
</tr>
<tr>
<td>4</td>
<td>Execute test</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>4.1</td>
<td>GUI manipulation</td>
<td>(3)</td>
<td>(0)</td>
</tr>
<tr>
<td>4.2</td>
<td>input test data</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>4.3</td>
<td>check test result</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>4.4</td>
<td>analyze log data</td>
<td>(5)</td>
<td>(0)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>38</td>
<td>16</td>
</tr>
</tbody>
</table>

### 4.3 Cost reduction at maintenance

By the word “maintenance” we mean that the projects received certain specification changes during the development, including the table design. If the table design is changed, the most affected part of the source code will be the statements that actually specify any of the columns of the tables. In here, we evaluated how many columns, which are actually directly written in the source code, can be reduced.

We estimated the number of columns to be affected before and after applying OPRF technique, which is shown in Table 5. This is based on what types of SQLs can be automated and the number of columns actually written in the source code. Table 3 shows the types of SQLs and for each one of them if it is automated or not by this technique, and the mean to measure the number. Table 4 shows the results from counting and calculating the number of columns that are actually written in the source code.

We achieved 90 percent reduction in A and C projects, and 30 percent in B project. The reason the number for B project is significantly different from A and C is the number of SQLs that are complex was rather larger than the other two projects. We see that the typical web applications will be closer to the B project rather than A or C. Thus, considering 30 percent reduction for maintenance of web application would be appropriate.

### 5 Related works

Enterprise JavaBeans [Suna] deals with object persistency by mapping one table to one class, which is less scalable in terms of execution performance, whereas OPRF provides more flexible mapping. JDO [Sunb] is another emerging technology but it is not yet available for industrial critical systems, and the mapping between a table and a class is the same as EJB. As a framework, [Saimi00] presented a similar approach, but it focuses on the presentation layer of web application systems.

### 6 Conclusions and discussions

We developed a reuse framework for database access layer, and applied to the actual web application development projects. The framework consists of (1) SQL execution component, (2) database connection component, and (3) table access source code generator.

The case study showed that the framework successfully achieved 20 percent cost reduction at the implementation phase, 5 percent at design phase, 17 percent at test phase, and 30 percent at maintenance phase.

The framework only addresses the database access layer. We need to extend the CBD to other layers, particularly to user application domains such as banking, security, personnel, inventory, accounting, etc. However, the engineers in the development projects virtually no time to consider creating reusable components due to the tight schedules. By employing the technique such as proposed in [Suganuma98], to provide a set of reusable components, it is necessary to have collaborations between Engineering dept. and Development dept. Engineering dept. has extensive knowledge in the involving domain.
As we proceed to accumulating more and more components, it will be important to categorize them for finding the appropriate components without time consuming searching and trying activity. The components themselves evolve as they are reused, such as new functionality added, corrective maintenance, providing new hotspots. It will be important to have the system for management and to provide certain organizations dedicated for evolutions and management.

The cost estimation schemes for software development using reusable components would also be important so that the project manager can be convinced that reusing components is really cost effective than creating from scratch, despite the risk of underestimating the quality and overestimating the functionality of the components.

References


