A True Story of Refactoring a Large Oracle PL/SQL Banking System

Csaba Nagy
University of Szeged
Department of Software Engineering
ncsaba@inf.u-szeged.hu

Rudolf Ferenc
University of Szeged
Department of Software Engineering
ferenc@inf.u-szeged.hu

Tibor Bakota
FrontEndART Software Ltd.
Hungary
bakotat@frontendart.com

ABSTRACT

It is common that due to the pressure of business, banking systems evolve and grow fast and even the slightest wrong decision may result in losing control over the codebase in long term. Once it happens, the business will not drive developments any more, but will be constrained by maintenance preoccupations. As easy is to lose control, as hard is to regain it again. Software comprehension and refactoring are the proper means for reestablishing governance over the system, but they require sophisticated tools and methods that help analyzing, understanding and refactoring the codebase. This paper tells a true story about how control has been lost and regained again in case of a real banking system written in PL/SQL programming language.

Keywords

Oracle PL/SQL, Refactoring, Software Quality Assurance

1. INTRODUCTION

Banking systems are critical systems from many different aspects. A simple rounding error may have a catastrophic influence on the reputation of the financial company, so there is a high pressure on developers to work precisely and test their code as much as possible. However, the business departments often urge the company to react for changes and implement new features rapidly. Developers emphasize reusing already working and tested solutions with fast, minor modifications, instead of properly designing solutions keeping in mind the quality and the maintainability of their code. This usually results in a fast evolution and growth of the system based on uncertain and ad-hoc decisions, which may end-up in losing control over the codebase in long term.

In this paper we present a true story of refactoring a large banking system mostly written in Oracle PL/SQL. The system under question is being developed by one of our industrial partners from the financial sector whose name is not published with regards to confidentiality agreement. After many years of development, they realized that their system's development was heading in wrong direction and they asked for help to take necessary steps against their serious software maintainability problems. Here we present both our analysis of their problems and our assessment.

The main contributions of this paper are:

- a case study – performed in a real industrial environment – of analyzing quality attributes and reconstructing the architecture of a large PL/SQL banking system;
- working solutions for emerging maintainability problems during the development of a large PL/SQL system.

The paper is organized as follows. First, we introduce the background story of the analyzed system in Section 2. Then, in Section 3 we present our first assessment of the system under question. Afterwards, we present our solutions in Section 4. Finally, we elaborate on related work in Section 5 and we conclude our paper in Section 6.

2. THE STORY BEGINS

The story began when our partner bought a boxed financial software from India. The programming language of the software was Oracle PL/SQL and it was designed to be easily extendible with additional functionalities. The only drawback of the software was that some parts of the core system contained wrapped stored procedures and packages, therefore it was not possible to modify the core functionality, but at that time this did not seem to be necessary.

Later the company started to extend their system with new features. The system evolved rapidly, and soon it became too large, so the small development team of the company could not maintain it alone anymore. Instead of hiring new programmers, they decided to outsource the development of certain modules to professional companies. The companies had to take responsibility for their own code so this decision seemed reasonable. However, vendors started to work hard and the system started to grow again rapidly. It's architecture became very complex soon. The company realized that maintaining the system and implementing new features became more and more expensive, so they had to stop and take serious steps to handle this situation. An illustration of this status can be seen in Figure 1.

Some of their critical problems were the following:

- the system was too complex,

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

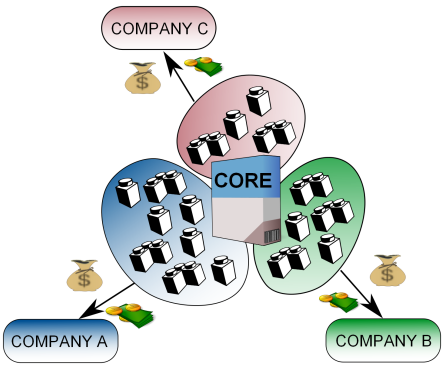

Figure 1: Outsourcing of the development of larger components.

- only a few experienced developers were aware of the full architecture,
- modifications were extremely expensive,
- nobody could estimate the cost of a modification,
- poor code quality,
- maintenance was very expensive,
- testing was very expensive.

This was the point in time when we were involved.

3. ANALYSIS AND ARCHITECTURE RECONSTRUCTION

Our first look at the system showed us that it was written mainly in Oracle PL/SQL with some additional subsystems (e.g. web clients) in Java. Our main attention was on the PL/SQL code because the full business logic was implemented there together with the data-management tier which laid in the PL/SQL codebase.

Our first assessment was to create an architecture map of the system in order to understand the system and show interrelations among higher level components. Such a map can be a useful tool to estimate the impact of a change in one component to the other ones. We created a map from two main sources of information: first, we performed a detailed low-level static analysis of the PL/SQL codebase, and second, we conducted interviews with the developers.

3.1 Low-Level Static Analysis

The static analysis was performed on PL/SQL dumps using the PL/SQL front-end of Columbus [1]. Our purpose was to identify low-level database objects (tables, views, triggers, packages, standalone routines) and relations among them (call relations, CRUD relations, etc.). The system turned out to be larger than we first expected. We analyzed 4.1M lines of PL/SQL and SQL code (full dump w/o data) which had 8,225 data objects, out of which 2,544 objects were packages. The total number of stored routines were more than 30,000 with more than 1.8 MLOC (million lines of code) in total (see Table 1).

3.2 Interviews

We interviewed the developers to identify higher level logical components of the system. PL/SQL was not designed to support higher level modularization so it was necessary to obtain this information from the developers instead of the codebase itself. We identified 26 logical components (e.g. Accounting, Security, ...). Developers also told us that they kept strict naming conventions, hence the corresponding component of a data object could be identified from its name easily (e.g. `PKAC_*` is a package of the Accounting component). Unfortunately, later we found that their naming convention was not that much strict, so many objects remained uncategorized.

Table 1: Overview of the system.

total size of the full dump (w/o data)	4.1 MLOC
total size of stored procedures	1.8 MLOC
number of PL/SQL objects (tables, views, triggers, packages, routines)	8,225
number of packages	2,544
number of stored routines (including routines in packages)	>30,000

3.3 The Architecture Map

Based on the naming conventions we grouped low-level data objects into components and we lifted up relations among them to the higher, component level. The final result was a dependency graph where nodes were the components (identified via interviews) and directed edges showed the dependencies among them (identified via static analysis). The graph showed more than 200 dependency edges among the 26 components meaning that every object depended on almost every other (see Figure 2).

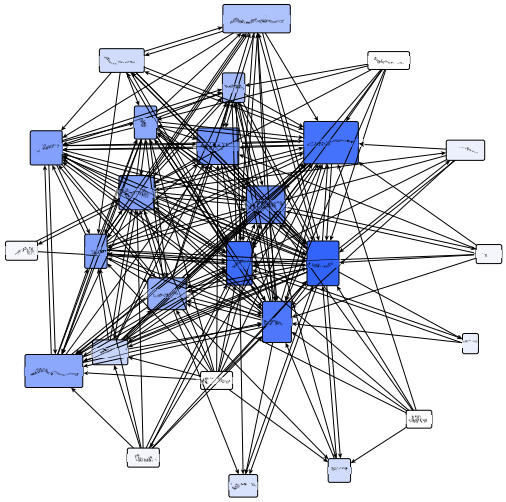

Figure 2: Relations between components (names distorted).

The results of the architecture reconstruction task showed that the system design was very complex. Even a simple change in a component may have an impact on almost all other components.

3.4 Code Quality

We investigated the quality of the source code as well. We identified many extremely large (more than 3,000 LOC) and complex (McCabe's complexity larger than 1,000) stored

routines in the system. Besides, we measured more than 20% clone coverage (copy&paste source code fragments) and we found 5 almost identical copies of a package with more than 5,000 LOC.

Apart from the most critical outlier objects of the system, the overall source code quality also showed a great negative influence on the maintainability of the system. We found thousands of coding rule violations and dangerous error-prone constructs in the codebase.

4. SOLUTIONS FOR MAINTAINABILITY PROBLEMS

The first assessment showed us that the company had a good reason to ask for help. We suggested them three primary solutions which we discuss in detail in this section.

4.1 Stop Deterioration of Code Quality

Complex system architecture and bad code quality implied that the first and most important step that they should take was to stop the deterioration of their software. The huge amount of incoming source code from different vendors resulted in different coding styles at different quality standards.

The company realized in time that they need to set up a system to continuously monitor their code. The SourceInventory framework provides solutions for them to analyze their codebase. It periodically measures metrics, checks and reports on potential errors and on procedures where important metrics increased above a specified baseline.

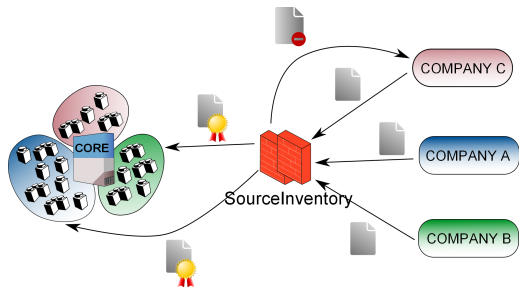

Figure 3: SourceInventory as a delivery-acceptance certificating system.

The framework serves as a delivery-acceptance certificating system. If an incoming code from a vendor does not conform to the quality requirements of the company, it will not be committed into the codebase (see Figure 3).

4.2 Enhance Test Coverage

Enhancing test coverage is another solution to improve the quality of the code. A test procedure is more effective if it has a better coverage, but first of all, we need to measure it. The company had a qualified testing team, they built up a regression testing framework with automated tests, they created and ran series of test cases for different usage scenarios, but they did not have a ready solution to measure the coverage. The testing team had no information on the set of covered stored procedures for different test cases. Without this information, after a change in the source code they always needed to re-run all the test cases and they were still not certain whether they tested the changed part of the code or not.

Oracle provides support for profiling (`DBMS_PROFILER`). Using Oracle's profiler, executed stored procedures can be easily obtained by querying virtual tables of the database manager. By using a toolset, we connected the profiler's data with the test cases and prepared coverage reports for test runs. The gathered information showed us that the test cases covered only about 20% of the stored procedures.

4.3 Elimination of Unused Objects

We followed the life-cycle of the system during a half-year period. It turned out that during this period the total number of database objects increased by about 25% (see Table 2). Developers told us that during this period they added some new features into the code which may explain the huge number of new objects, especially the number of new tables (see Table 3). Another reason is that they usually use working (temporary) tables which often remain in the database even after the final phases of the development.

Table 2: Growth of the total number of database objects in the system during a half year period.

Date	Total number of objects
2010.04	8,255
2010.09	9,582
2010.11	10,681

Table 3: Detailed growth of the system during a half year period.

Date	Table	View	Trigger	Routine	Package
2010.04	3,943	1,350	337	51	2,544
2010.09	4,868	1,459	346	102	2,807
2010.11	5,865	1,462	355	143	2,856

They also informed us that they re-implemented the largest component of their system in Java and they functionally cut-off this component from the rest of the PL/SQL codebase. Hence, a huge number of data objects remained in the code unused. Additionally, large data tables also remained in the database, but became useless after the reorganization. All the unused stored procedures and packages increase the complexity of the system. Furthermore, large and useless data tables affect hardware maintenance costs too. Note, that the required table space could be measured in TBytes for these data tables.

Elimination of the obsolete component and unused data objects became important particularly because of hardware maintenance costs.

Removing unused data objects requires careful work for such a complex system. If an unhandled reference remains in the code, its consequences may be undetermined. Although it would be a catastrophic error, it would still be a better case when the system fails with an 'object does not exist' error, compared to miscalculating the account balance of a customer without any signs of error. Direct references to objects can be identified via the database manager or by static analysis, but dynamic references may still remain hidden.

All in all, we identified a number of challenges in eliminating a single unused component from the system:

- identifying tables/procedures of the component that became obsolete,

- identifying references to tables/procedures of the obsolete component,
- validating the correct removal of the elements (e.g. make sure that no dead code remained after removing them).

4.3.1 Identification of Objects of Obsolete Components

We could identify data objects of the obsolete component by using our previous categorization based on the naming conventions of the company. However, it was not enough as some of the developers did not keep the naming conventions and many objects remained uncategorized.

We defined five elimination sets and calculated them via static impact analysis:

SET_1 : elements of the obsolete components that match the naming conventions;

SET_2 : uncategorized elements in (direct or transitive) relation only and only with objects from SET_1 or SET_2;

SET_3 : uncategorized elements in (direct or transitive) relation with objects from SET_1 and SET_4;

SET_4 : categorized elements in (direct or transitive) relation with objects from SET_1;

SET_5 : elements that have no (direct or transitive) relation with SET_1.

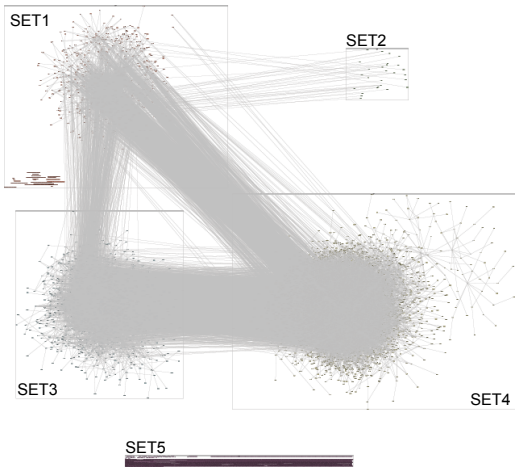

Figure 4: Elimination sets showing many un-cut relations between the obsolete component and others.

The connections between SET_1 and SET_3 or SET_4 (see Figure 4) showed us that the component had not been functionally cut from other components even though developers told us that they had done this before.

4.3.2 Identifying References to Objects of the Component and Validating their Correct Removal

We identified references to objects via the database manager's internal functions (`DBA_DEPENDENCIES`) and via static analysis extended with algorithms to analyze embedded dynamic SQLs (e.g. `EXECUTE IMMEDIATE` statements) where it was possible.

Our partner asked us to focus on the removal of large, unused data tables. After identifying these tables, developers cut off identified references and we recommended to rerun the automatic test cases with auditing the usage of these tables both in the test databases and in the live system. If the table has no more accesses after a certain period, it can be dropped safely, but it is still needed to check its triggers whether triggers use additional stored procedures that could be dropped too.

This technique can be extended to other data objects too.

5. USEFUL ADDITIONAL TOOLS

In this section we introduce some additional useful tools for analyzing Oracle PL/SQL code that may help companies in similar situations. Quest's TOAD[1] is a tool to perform complex analysis of a PL/SQL system. It is able to measure metrics, coding errors and performance issues in the database. Oracle SQL Developer[2] is also able to prepare Quality Assurance reports that identify conditions that are technically not errors, but that usually indicate flaws in the database design. SD CloneDR[3] has also a PL/SQL front-end as a tool to find exact or near-miss duplicated code.

6. CONCLUSIONS

As an industrial paper, we briefly introduced our techniques that we used during the preparation of our case study. Instead of technical details and research questions we kept our focus on the industrial context and highlighted the practical benefits. We note that for PL/SQL systems such a complex methodology against software deterioration is novel, to our best knowledge. However, most of the applied techniques has a huge amount of related work for other languages (e.g. dead code elimination, test coverage measurement and test selection, software quality analysis, etc.).

Our story began with serious maintainability problems at our industrial partner from the financial sector. We have analyzed their problems, set up a delivery-acceptance certification system, a test coverage measurement toolset and we assisted them in eliminating unused data objects in order to simplify their system's architecture. We believe that our complex methodology to stop software deterioration and solve maintenance issues helped them in their daily problems. Some of the previously mentioned techniques are so novel to the company that we cannot report on objective measures comparing maintenance costs or quality attributes before or after our suggestions. However, it is obvious that the company had a great need for ready solutions and they were eager to have them as soon as possible. After all, we are certain that nowadays they pay a lot more attention on the code quality and overall complexity of their system.

7. REFERENCES

[1] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, Oct. 2002.

[1] `http://www.quest.com/toad/`
[2] `http://www.oracle.com/technology/products/database/sql_developer/index.html`
[3] `http://www.semanticdesigns.com/Products/Clone`