Computer Standards & Interfaces 000 (2017) 1–13

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

White-box modernization of legacy applications: The oracle forms case study ☆

Kelly Garcés [a,1,*], Rubby Casallas [a,1], Camilo Álvarez [a,1], Edgar Sandoval [a,1], Alejandro Salamanca [b,2], Fredy Viera [b,2], Fabián Melo [b,2], Juan Manuel Soto [b,2]

[a] Department of Systems and Computing Engineering, School of Engineering, Universidad de los Andes, Bogota D.C., Colombia
[b] Asesoftware Ltda., Bogota D.C., Colombia

ARTICLE INFO

Keywords:
Industrial case study
Model-driven engineering (MDE)
Configuration
Quality attributes
Oracle forms
Java
.Net

ABSTRACT

Software modernization consists of transforming legacy applications into modern technologies, mainly to minimize maintenance costs. This transformation often produces a new application that is a poor copy of the legacy due to the degradation of quality attributes, for example. This paper presents a white-box transformation approach that changes the application architecture and the technological stack without losing business value and quality attributes. This approach obtains a technology agnostic model from the original sources, such a model facilitates the architecture configuration before performing the actual transformation of the application into the new technology. The architecture for the new application can be configured considering aspects such as data access, quality attributes, and process. We evaluate our approach through an industrial case study, the gist of which is the transformation of Oracle Forms applications—where the presentation layer is highly coupled to the data access layer—to multitiered applications.

1. Introduction

Software is constantly evolving; this evolution is motivated by different reasons such as the obsolescence of a technology, the pressure of users, or the need to build a single coherent information system when companies merge [2]. Our research lies in the field of *software modernization*, a kind of evolution, that refers to the understanding and evolving of existing software assets to maintain a large part of their business value [3].

In cooperation with industry partners, we have carried out projects to tackle different modernization challenges: (1) Migration from *Oracle Forms* to *Java* [1,4]; (2) Restructuring of *Java Enterprise Edition* (JEE) applications from monolithic architectures to microservices [5]; and (3) Maintenance of *Ruby on Rails (RoR)* applications developed by Agile practitioners [6]. These projects used a variety of source technologies (i.e., Oracle Forms, JEE, RoR), which led us to generalize the two main steps involved in any modernization: (1) *Understanding* and (2) *transforming*. The generalization of the first step is described in [7]. In that work, we propose an approach to generate architectural views that help to understand the original application and delimit the modernization scope. In turn, this paper presents the abstraction of the second step (see Section 2), that is, how to configure the as-is to obtain a to-be that is valid in the target technology, and generate the corresponding code.

In particular, this work deals with the transformation task when manually developed. The problem arise as a result of rewriting excerpts of legacy code from scratch —or even worst, of copying and pasting existing excerpts from one place to another— despite their similarity. Also, developers have to switch from the modern IDE to the legacy IDE in order to solve their questions related to the legacy implementation details. Therefore, rewriting and switching are time consuming and error-prone.

We have studied commercial tools and research works that deal with the translation problem (Section 7), and, therefore, are able to conclude that their output code is difficult to maintain and evolve because they

☆ This paper is an expanded and revised version of the document entitled "White-box Modernization of Legacy Applications" [1] presented in the International Conference on Model and Data Engineering (MEDI) 2016 in Almería (Spain), held between September 21, and 23 2016. The paper extends the version that has appeared in the Conference as follows: (1) Shows the studied problem and solution in their generic form; (2) Details each step of the modernization process, whilst former version mostly elaborates on the configuration step; (3) Presents more results of the experimental evaluation; (4) Discusses how our approach is extensible to other type of applications; and (5) Includes a deep review of the state-of-the-art.

* Corresponding author.
E-mail address: kj.garces971@uniandes.edu.co (K. Garcés).

[1] http://www.uniandes.edu.co/.

[2] http://www.asesoftware.com/.

https://doi.org/10.1016/j.csi.2017.10.004
Received 14 December 2016; Received in revised form 5 October 2017; Accepted 16 October 2017
Available online xxx
0920-5489/

mostly apply a black-box transformation approach [8]. The following are the drawbacks of black-box transformation: (1) Lack of information, which may hamper the proper execution of the application on a modern platform; (2) Analyses to discover dead code that the developer would want to avoid in the new application are rarely performed; (3) The ability to modify the user interface appears very late in the transformation life cycle (i.e., once the code has already been generated); (4) Means to see the transformation progress are uncommon in most of the related work.

These drawbacks have motivated our proposal, which is a white-box transformation process that focuses on the understanding of the legacy application, but put in the terms of a *technology agnostic model*. Our work targets stakeholder developers, so that they can use this model to configure the target architecture in the early steps of the process, and indicate the transformation progress. Even though it is likely that developers modify the modern code in our approach in order to complete the implementation of functionalities, the approach reduces the modifications to be made at a code level because they can configure many aspects of the architecture at a model level. The latter favors maintainability because most of the generated code follows default design patterns and standards. We count the following among the concerns that can be configured: *data access, quality attributes (maintainability, usability, and security), and configuration progress*. Our approach is based on Model-Driven Engineering (MDE) techniques that have been proven to improve productivity and quality in migration processes [9–11].

Furthermore, this article reports an instantiation of the approach in the Oracle Forms technology (see Section 3), which is our most advanced case study since it covered a pilot study developed in conjunction with a Colombian company called Asesoftware. This real application allowed us to identify the parts of the code that can be modernized in a fully automated way and the elements that have to be manually changed by developers.

We demonstrate the applicability of our approach by transforming Oracle Forms applications into a multi-tier architecture (see Section 4). The evaluation (see Section 5) covers a proof of concept and the pilot study. In particular, the pilot study demonstrates that developers that use the white-box method have an increased productivity and produce higher quality code than developers that use a manual method. It is worth noting that these developers are not Oracle Forms experts. We have built an editor on top of the technology agnostic model to ease the configuration. Section 6 discusses how our approach has been extended to different Oracle Forms versions and to an additional target technology (i.e., .Net), and elaborates on the necessary effort to extended it to further technologies. Finally, Section 8 concludes the paper and outlines future work.

2. Generic white-box transformation approach

As mentioned in the introduction, this section presents the generalization of the transformation process (see Fig. 1). In the first step a technology specific model is obtained from the legacy code. The second step comprises transforming this model into a technology agnostic model. In contrast to the former model —which is verbose and technology specific—, the technology agnostic model conforms to a metamodel, the concepts of which matter in the target architecture. Steps three to six can be performed in an iterative manner. In each iteration, the developer can configure how to transform a set of legacy artifacts (third step), and generate, complete, and test the corresponding new code (steps 4 to 6), until satisfying the scope of a given modernization project. As we proposed in our previous work [4], the decision of which artifacts are included in each iteration is made in the understanding phase and depends on aspects such as artifact dependencies and client requests. In the remaining sections, we describe an instantiation of the approach in the Oracle Forms technology.

3. Background of oracle forms case study

The case study comes from the “Forms Modernization” project. *Oracle Forms* appeared towards the end of the 1980s and provides a rapid application development environment plus a run-time environment.

A basic Forms application consists of forms accessed via a menu and, in most cases, these forms have items that can be mapped to database tables. Most of the database behavior (read/write/update/delete) is predefined in Oracle Forms, thus avoiding the need to write too many lines of code.

Oracle Forms enables programmers to develop database-centric functionality, thus minimizing the need to program common functionality like the CRUD (Create/Read/Update/Delete) operations that manage records of database tables. Oracle Forms also makes it possible for developers to include PL/SQL code in the applications in order to enrich their functionality beyond CRUD standard logic. PL/SQL is a procedural programming language integrated with SQL. The PL/SQL code can be executed by the database management system or directly by the Oracle Forms environment (as code excerpts embedded in the form trigger events or Oracle Forms Properties). This section is structured as follows: first, it presents the architecture of Oracle Forms applications, and then the modernization scope and target architecture defined in the project.

3.1. Source architecture

Fig. 2 illustrates the architecture of Oracle Forms applications. It is client/server where the application is highly coupled to the database. Events associated to components of the application (i.e., Form, Block, Item) fire triggers. The components that appear in the Figure are described below:

- *Form*: A Form is a collection of objects and code, which includes windows, items, triggers, etc.
- *Blocks*: Represent logical containers for grouping related items into a function unit to store, display and manipulate records of database tables. Programmers configure blocks depending on the number of tables from which they want to manipulate the form: (1) The way to display a single database table in a form is to create a block. This results in a *master form*. (2) The way to display two tables that share a master-detail relationship (i.e., “One to Many” relationship) is through two blocks. Oracle Forms guarantees that the detail block will display only records that are associated with the current record in the master block. This results in a *master/detail form*.
- *Item*: Items display information to users and enable them to interact with the application. Item objects include the following types: button, check box, display item, among others.
- *Trigger*: A trigger object is associated to an event. It represents a named PL/SQL function or procedure that is written in a form, block or item.
- *Menu*: Is displayed as a collection of menu names appearing horizontally under the application window title. There is a drop-down list of commands under each menu name. Each command can represent a submenu or an action.

3.2. Modernization scope

In the context of the Forms Modernization project, we automate the transformation of master and master/detail forms. Given a form, we are able to generate: (1) The corresponding graphical interface (except the layout); (2) The CRUD logic; (3) The scaffolding code that calls PL/SQL logic embedded in triggers. The transformation of PL/SQL code to modern language is the responsibility of the developer. We decided to leave the layout out of our modernization project for two reasons: (1) In Oracle Forms, the layout is implicit, therefore, it is necessary to implement complex algorithms to discover the concrete interface [12]; and (2) For us, the development of such algorithms might not be worth the effort