A GENERATIVE APPROACH TO IMPROVE THE ABSTRACTION LEVEL TO BUILD APPLICATIONS BASED ON THE NOTIFICATION OF CHANGES IN DATABASES

J. R. Coz, R. Heradio Gil, J. A. Cerrada Somolinos and J. C. López Ruiz
Departamento de Ingeniería de Software y Sistemas Informáticos, Universidad Nacional de Educación a Distancia
Ciudad Universitaria, Juan del Rosal 16, E-28040. Madrid, Spain

Keywords: Generative Programming, Software Product Line (SPL), SQL Procedural Extension Language.

Abstract: This paper highlights the benefits, in terms of quality, productivity and time-to-market, of applying a generative approach to increase the abstraction level to build applications based on the notification of changes in databases. Most of the databases maintain meta-tables with information about all stored tables; this information is used in an automatic process to define the software product line (SPL) variability. The remaining variability can be specified by means of domain specific languages. Code generators can automatically query the meta-tables, analyze the input specifications and configure the current product. The paper also introduces the Exemplar Driven Development process to incrementally develop code generators and the Exemplar Flexibilization Language that supports the process implementation.

1 INTRODUCTION

Significant research lines advocating for the increase in the productivity of the software development are the *Generative Programming* (GP) (Czarnecki, 2000), the *Model Driven Development* (MDD) and its representative the *Model Driven Architecture* (MDA) of OMG (Object Management Group) (Frankel, 2003), *Frameworks* (Gamma, 94) and *Software Product Lines* (SPL) engineering (Clements, 2002), (Verlage, 2005).

GP and MDD propose to raise the level of abstraction of programming languages through specifications or models. According to (Weis, 2003), a key factor for success with these paradigms is the automatic translation from the model to executable code. To make this possible, the domain should be restricted enough so that the overlaps between systems that integrate a family overcome discrepancies (Cleaveland, 2001).

Frameworks are considered as gray box abstractions (Greenfield, 2004) and as a result their reutilization demands a remarkable learning effort and the evolution of the products have a dependency on the framework implementation.

Early case studies have exhibited significant barriers to adopt an SPL approach. For instance, in the successful Diesel Engine SPL, Cummins stopped all product deployments for six months (Krueger, 2002) As C. Krueger argues (Clements, 2002), many organizations cannot afford to slow or stop production for six months, even if the potential *Return On Investment* (ROI) is huge.

The approach proposed in this paper is the construction of a *SPL* using an approach based on *GP* (Czarnecki, 2000). To build the *SPL* an adaptation of the *Exemplar Driven Development* (EDD) and *Exemplar Flexibilization Language* (EFL) (Heradio, 2007) are used. The *SPL* created is about the *notification of changes in databases*.

The purpose of the *notification of changes in databases* is to provide a range of services to users to make them aware of the changes that are being produced in a database. For the construction of such applications programmers should know certain SQL *procedural extension languages* and, in some cases, certain libraries that provide the database products. These kinds of applications must be manually built and the cost of development is high.

The development of the *SPL* presented in this work offers a high productivity and profitability as it will be discussed later in this paper, allowing the automatic construction of all products of the SPL, from an exemplar, or product modified with the necessary flexibility, and a *Domain Specific*

Language (DSL) for this domain.

This paper is structured as follows: section 2 presents the domain. Section 3 describes the analysis of this domain. Section 4 summarizes EDD. Section 5 presents EFL that supports the flexibilization of any software artifact. The section 6 gives an overview of the profitability of the solution provided in several study cases. Finally, the section 7 summarizes the presented work.

2 DOMAIN OVERVIEW

The research problem is to find an economic way for implementing the change notification service in databases. This service is responsible for communicating the changes that happen in the database to the subscribed users. Users can be interested only in specific events. For example, users may need to be reported about: insertions, deletions, updates, login, logouts, start ups, shutdowns and others. To implement this kind of features nowadays databases offer different mechanisms such as Advanced Queue, Pipes or Alert / Signals technologies, *procedural extension languages* such as PL / SQL and specialized libraries that extend these languages such as AQ, Pipe and Alert /Signals Libraries.

Although these utilities facilitate the developments, products must be programmed manually and the cost of development is high. The development of specific products for this domain depends not only on the specific requirements established (priorities, time management, subscribers, searches, granularity of the solution, visibility, navigation between messages and so on), but the internal structure of the database (tables, keys, users and others).

3 DOMAIN ANALYSIS

The main requirements of this domain have been analyzed after developing several products. Feature-Oriented Domain Analysis (FODA) has been used in order to specify the domain features. The FODA notation followed (Czarnecki, 2004) uses the idea of cardinality to solve the difficulties suffered by other notations (Czarnecki, 2004), as suggested in (Heradio, 2007).

A *new DSL, called Notification Change Service Language (NCSL)* has been developed to gather the domain variability. In order to derive new products, the *application engineer* writes NCSL specifications, from the user requirements, that are completed with information automatically gathered from the database as tables, users, fields, keys, schemas and others.

Some elements of this NSCL describe variability related to the internal database information (tables, keys, schemas and others) whereas other elements describe the events priorities, times, subscribers, type of visibility, events to be notified by the service, permissions and so on. Users, through a program implemented for this purpose, specify their needs against this NSCL.

4 EDD

EDD is a *SPL* methodology which takes advantage of the similarities among domain products to build them by analogy (Heradio, 2007).

The EDD starting point is whatever domain product built using conventional software engineering. The product is called *exemplar*. It is assumed that this *exemplar* implements implicitly the intersection of all the domain product requirements. To satisfy the domain variable requirements that are out of the intersection, EDD uses the concept of *exemplar Flexibilization.* The *Flexibilization* is the mechanism that allows establishing an analogy relation between the *exemplar* and the new product, so the new products can be derived automatically from the *exemplar*. The tool that performs the flexibilization is a *domain specific compiler* (DSC), which is used to derive automatically new products.

An adaptation of EDD has been developed, where a NSCL is built specifying the user features and using the necessary information from the database. This database information is contained in metatables and it is obtained automatically. Once the *domain specific language* exist (in this case, the NSCL), the DSC for this language is implemented. A summary of the EDD adaptation is illustrated in the next figure:

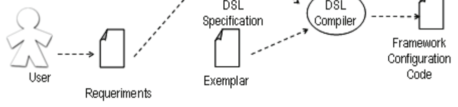

Database Management System MetaTables
DSL Specification
DSL Compiler
User
Requeriments
Exemplar
Framework Configuration Code

Figure 1: Generative model.

5 EFL

EFL is an external flexibilization technique that supports non invasive exemplar transformations and crosscutting flexibilizations. It is applicable to whatever kind of software artifact and provides an efficient generative variant construction [Heradio, 2007].

EFL is used to build the *DSC* that deal with the specification variability and also with the implementation variability in our domain case study. EFL is currently implemented as a library of the Ruby object oriented language (Thomas, 2004). This implementation is distributed as Lesser GNU General Public License (LGPL) and it is available in different repositories as Ruby Forge [Heradio, 2008]

The most important part is that *generators* are responsible for analyzing the *exemplar* and adapt it in order to generate the new product according to the given specification. Generators are also responsible for detecting dependencies and inconsistencies in the configuration model. This capability, in the *SPL* presented in this paper, is considered essential because the user might have selected wrong requirement or the requirements could contain incompatibilities among them, as combinations not allowed. This could drive to an invalid product for the SPL. In case of misconfigurations *generators* provide a detailed report about the incompatible features. The user can use this report to review the selected features.

Finally, *generators* can analyse the internal elements of the database to obtain all the necessary information of the domain. Figure 2 illustrates how the generators work: analyzing the NSCL configuration consistency, updating information from the database, generating the required reports and finally getting all the products of the *SPL* from the NSCL and the *exemplar.*

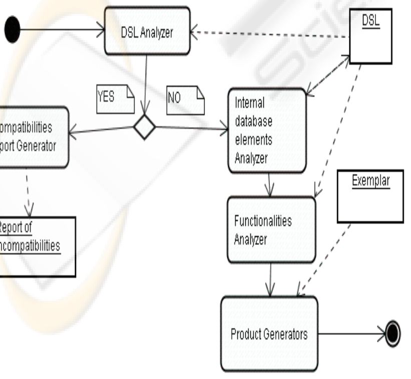

DSL Analyzer
DSL
YES
NO
Incompatibilities Report Generator
Internal database elements Analyzer
Report of Incompatibilities
Functionalities Analyzer
Exemplar
Product Generators

Figure 2: How Generators work.

6 STUDY CASES

The presented SPL has been employed for conducting several case studies with different databases: a database supporting a university that offers courses, a control system for air navigation that contains electronic controls that inform pilots of changes in different aspects of navigation and others.

Oracle database has been used in all these case studies. Different implementation mechanisms provided by Oracle are applied for implementing the products. For example, it has been selected the PL / SQL language and several libraries incorporated in Oracle.

In all of these studies the *SPL* generates the 100% of the new products, covering all the features specified. The study of investment profitability is summarised below:

1. Generative properties and profitability. The number of products that can be obtained with the proposed *SPL* is measured. This number corresponds to the combinations of features that have sense, that is to say, that do not maintain dependencies or constraints among them.

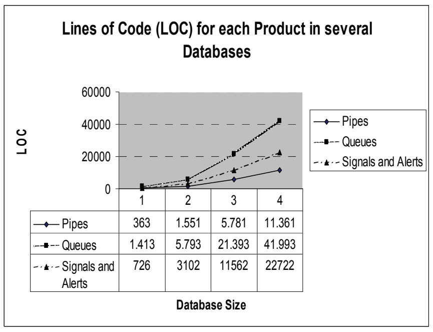
The number of products obtained in some of these case studies carried out is tens of thousands. This estimation is calculated using all the valid requirements combinations.

For example, the Oracle Advanced Queue mechanism is used in case of air navigation control system. The number of valid feature combinations is illustrated below, in table 1.

Table 1: Number of products that can be generated.

Type	Combinations
Time Management	**12**
Subscriptions and Priority	**4**
Aggregations and Visibility	**6**
Waits and Granularity	**9**
Navigation and Searches	**6**
Operations	**32**
Total	**497.664**
Not Valid	**124.416**
Number of Products	**373.248**

2. Profitability Depending on the Database size. For larger databases the code to be generated is bigger than for small ones. The database size, in a simplified form, depends on the number of tables with requirements of notification changes, the number of users of the database who subscribe to the notification changes service and the number of fields in each table. This study shows that the profitability increases with the size of the database, that is to say, more code is automatically generated. Figure 3 illustrates a study of four databases with different sizes, and the average number of code lines generated for each product.



	1	2	3	4
Pipes	363	1.551	5.781	11.361
Queues	1.413	5.793	21.393	41.993
Signals and Alerts	726	3102	11562	22722

Figure 3: Code Lines vs. Database Size.

In this study we have considered different implementation technologies and databases. The first database has the smallest size, with only 8 tables that contains changes notifications features and with 10 subscribers to *the notifications changes service*. The second one contains 50 tables and 10 subscribers. The third one contains 200 tables and 30 subscribers and in the last one there are 400 tables and 50 subscribers. In all cases there is an average size of the fields for each table.

3. Effort to Build the SPL. To make this estimation, measured in lines of code (LOC), it is taken into account: the development of the test products developed, the exemplar, the generators and the entire generation of the NSCL. For example, in a database of average size (50 tables and 10 subscribers), this effort is equivalent to the development of a dozen products.

7 CONCLUSIONS

This paper has showed the construction of a *SPL* using a *generative programming* approach. A new DSL, called NCSL*,* has been developed to gather the domain variability. EDD and EFL [Heradio, 2007] have been used as supporting tools to build the product line.

The *SPL* presented has been applied to solve different study cases related with *change notifications service in databases*. In all of these studies the *SPL* was able to generate the 100% of the new products, covering all the requirements specified.

The profitability analysis shows great benefits of applying this SPL. This profitability increases with the database size.

REFERENCES

Czarnecki, K.; Eisenecker, U. W. *Generative Programming. Methods Tools and Applications*. Addison-Wesley, 2000.

Frankel, D. John Wiley and Sons, 2003. *Model Driven Architecture: Applying MDA to enterprise Computing.*

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley. 1994.

Clements, P.; Northrop, L. *SOFTWARE PRODUCT LINEs: Practices and Patterns.* Boston, MA: Addison-Wesley, 2002.

Verlage, M.; Kiesgen, T. *Five years of product line engineering in a small company.* Proceedings of 27th International Conference on Software Engineering, 2005 (ICSE 2005), pp. 534-543.

Weis, T.; Ulbrich, A.; Geihs, K. *Mode metamorphosis. Software, IEEE.* Volume 20, Issue 5, Sept.-Oct. 2003 Page(s):46 – 51.

Cleaveland, J. C. *Program Generators with XML and JAVA*. Prentice Hall, 2001.

Greenfield, J.; Short, K. Software *Factories: assembling Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

Krueger, C. *Eliminating the adoption barrier.* IEEE Software, Volume 19, Issue 4, 2002, pp. 29-31.

Heradio, R. *Metodología de desarrollo de software basada en el paradigma generativo. Realización mediante la transformación de ejemplares.* Ph. D. Thesis, Ingeniería de SW y Sistemas Informáticos de la UNED, España. 2007.

Czarnecki, K.; Helsen, S.; Eisenecker, U. Staged *Configuration Using Feature Models. Software Product Lines Conference (SPLC).* Boston, MA, USA, August 30-September 2, 2004, pp. 266-283.

Czarnecki, K.; Bednasch, T.; Unger, P.; Eisenecker, U.W. *Generative programming for embedded software: An industrial experience report. ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering* GPCE'02), Pittsburgh, October 6–8, 2002, LNCS 2487, Springer-Verlag (2002), pp. 156–172

Thomas, D.; Fowler C.; Hunt, A.; *Programming Ruby: The Pragmatic Programmers' Guide.* Pragmatic Bookshelf; 2nd edition (October 1, 2004).

Heradio, 2008, *A Ruby implementation of EFL in RubyForge:* http://rubyforge.org/projects/efl/