RESEARCH ARTICLE OPEN ACCESS

Agile Data: Automating database refactorings

Bruno Xavier*, Guilherme Lacerda*, Vinicius Ribeiro*+, Emerson Ribeiro*, André da Silveira*, Sidnei Silveira**, Jorge Zabadal***, Fábio Gonçalves ****
*(Department of Computer Science, UniRitter, rua Orfanotrófio, 555, Porto Alegre, CEP 90.840-000 - RS – BRAZIL)
** (Department of Information Systems, UFSM Frederico Westphalen. Linha 7 de Setembro, s/n - BR 386 Km 40 – CEP 98400-000 - Frederico Westphalen – RS - BRAZIL)
*** (Department of Mechanical Engineering, Federal University of Rio Grande do Sul – UFRGS – Rua Sarmento Leite , 425, - CEP 90.050-170 - Porto Alegre – RS – BRAZIL)
**** (Department of Design, Federal University of Rio Grande do Sul – UFRGS Avenida Osvaldo Aranha, 99 - 6º andar- sala 607 - CEP 90035-190 - Porto Alegre, RS - , BRAZIL)
+ (Department of International Relations, ESPM, R. Guilherme Schell, 350 – CEP 90640-040, Porto Alegre, RS, BRASIL)

ABSTRACT
This paper discusses an automated approach to database change management throughout the companies' development workflow. By using automated tools, companies can avoid common issues related to manual database deployments. This work was motivated by analyzing usual problems within organizations, mostly originated from manual interventions that may result in systems disruptions and production incidents. In addition to practices of continuous integration and continuous delivery, the current paper describes a case study in which a suggested pipeline is implemented in order to reduce the deployment times and decrease incidents due to ineffective data controlling.

Keywords – Information Systems Design; Agile Methods, Automating Processes; Refactoring

I. INTRODUCTION

In system development processes, controlling and updating data across the delivery workflow are critical for the wealth of projects. In software organizations, the isolation of database activities and the lack of automation processes that involve data are quite common. Simple yet manual jobs that demand DBAs' and/or Operations teams' intervention may generate bottlenecks to the process. Both Sadalage [7] and Humble and Farley [6] strongly recommend the use of continuous integration for database management so that teams cannot only integrate code but also data. To adapt database control to a new model is not an easy job, since unlike code, data is greatly increased during the product life cycle and, therefore, changes and migration tasks must be carefully carried out. According to Humble and Farley [6], as a system evolves, changes are inevitable, thus, mechanisms that allow the smooth execution of such modifications in favor of process reliability are necessary.

In order to speed up the development process, this paper intends to introduce agile concepts relevant to data management into the environments that compound the delivery pipeline. This work does not aim to teach the usage of the mentioned tools, but to offer an adaptive model to distinct realities. With the support of such resources, it is possible to manage data along with continuous integration practices by treating data as code through version control and automated deployments.

The paper is structured as follows: the first chapters introduce concepts as guidelines to be used in the evolution process of development. Ideas of database refactoring and data management in continuous delivery are described, since such notions underlie the approach further presented in the case study. Furthermore, problems related to the lack of integration processes and how their effects can be harmful to team productivity are explored. Finally, the article outlines a case study in a real organization, in which an automated procedure is applied to improve data management along with software delivery. The benefits of this method are shown through metrics collected before and after its implementation, which served as the basis for a discussion of the observed results.

II. CONTINUOUS INTEGRATION

Continuous Integration (C.I) was primarily described by Beck and Andres [3] and vastly mentioned by Fowler and Foemmel [5] as an effective way to speed up the delivery process, minimizing software errors. Under the hood, C.I aims to keep products functional in a constant manner by frequently committing code to a baseline, whereby it is integrated and tested. According to Humble and

Farley [6], without continuous integration, software is broken until someone proves it works, either in test or integration stages. This practice helps to reduce the impact caused by small modifications, once many developers can share the same project. To accomplish that, version control and automated build tools are necessary. Figure 1, depicted by Ashley [2], shows a diagram of a classic continuous integration environment.

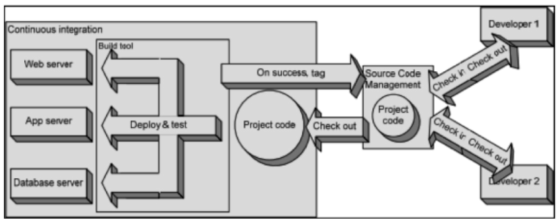

Fig. 1 Continuous Integration Setup (Ashley 2011)

As Ashley [2] illustrates, an integration environment works very similarly to an individual box, except for its version control and visibility to team members, since a trigger is launched every time code is committed to the baseline, as described below:

- The project repository is cloned from the version control system;
- The deployment is done along with unit tests;
- In case of success of previous steps, an incremental tag is created on the version control as a build number
- In case of failure, the execution is marked as an error and the server notifies the stakeholders.

III. DATABASE REFACTORING

Fowler and Beck [4] defines refactoring as refining code without changing either behavior or logic. To Ambler [1], database refactoring represents small schema modifications in order to improve its design. Although it seems to be a simple task, data modifications can be complex and take a certain time; hence, discipline and control are keywords.

According to Ambler [1], in opposite to code refactoring, which concerns only with behavioral semantics preservation, database refactoring tends to be more complex, since it must deal with information semantics besides behavior. In other words, the occasional change on a column value must not affect the final user. For instance, the value, which represents a certain telephone number, either commercial or personal, could be improved to be, in addition to a new personal telephone column, just commercial. This change implies code modifications to handle behavioral semantics and database migration scripts to keep information semantics.

Ambler [1] yet enumerates five categories of database refactoring, the last three as subcategories of structural refactoring:

Quality: This category focus on data quality. For instance, a restriction applied to an attribute in order to avoid null values.

Structural: Represents the schema modifications such as attribute name changes, attribute removal, table splitting and so forth.

Architectural: A kind of structural refactoring, albeit closer to the application. It assumes changes in the database encapsulation, for example, a view created over two tables or a procedure with business logic migrated from the database to the application.

Performance: One of the most recurring tasks among operation DBAs, it includes activities as index inserts in favor of performance.

Referential Integrity: Corresponds to schema modifications with regard to integrity on tables relations – example: cascading deletion.

In present, automation tools such as *dbdeploy* and *liquibase* have support to the foregoing categories of database refactoring, along with rollback mechanisms to most of them. To model such tasks, collaboration between DBAs and developers is important, due to the need of specialists' visions of both database and applications coupling.

IV. DATA MANAGEMENT WITH CONTINUOUS DELIVERY

According to Humble and Farley [6], continuous delivery is a way of absorbing the business needs without environment disruptions. In fact, the data management differs from other parts of the system, since unlike other aspects of the software, once in production; a product increases its data and adds value. Therefore, in most cases, this data cannot be reconstructed on every release, but can be migrated in a reliable way to ensure the information consistency. As said by Humble and Farley [6], continuous delivery demands that each approved release can be deployed in production, which implies in the preservation of data state. To achieve that, Sadalage [7] suggests database continuous integration along with code as a flexible method to handle the control of data in the product life cycle.

In favor of that, version control and automated builds are critical to refactoring management. A technique first described by Schuh [8] and, today, used as foundation to automated tools, presents a partitioning of database changes in cohesive scripts, each of which representing an operation. These operations are defined as change logs and stand for the variations since the last release. The change logs are segmented for the sake of traceability. Listing 1 represents a small *DBDeploy* script.

Listing 1. DBDeploy Changelog Example

```
CREATE TABLE cliente {
ID BIGINT GENERATED BY DEFAULT AS
IDENTITY ( START WITH 1 )
PRIMARY KEY,
primeiro_nome VARCHAR (255),
segundo_nome VARCHAR(255)
);
--//@ UNDO
DROP TABLE IF EXISTS cliente;
--//
```

very script must use a name pattern for the execution sequence and change tracking.

Ambler [1] discusses three nomenclature strategies; in addition, Sadalage [7] includes the release method, which appends the release number along with the scripts.

For example, release-1.0.1.sql, release-1.1.2.sql and so forth. Sadalage’s [7] method is effective with small changes, even though the tracking down for modifications done between releases could be missed. All the others are described below:

- BuildNumber: Requires the creation of a new script on every build even without changes. Although it is a good practice either it comes to modifications recovery of specific builds, in case of small changes or frequent builds, it could be an overkill due to the amount of artifacts.

- TimeStamp: With the approach of timestamp, the scripts are named and consolidated based on the date in which the modifications were done (e.g., *20120212.sql*, *200120215.sql*). Such technique may implies issues with concurrency, since two people can commit on the same day, generating version control conflicts.

- UniqueIdentifier: This strategy offers a sequential nomenclature for scripts.

Likewise the software build number, generated through a continuous integration system, the sequence of script numbers dictates the build number of modifications in database (e.g., *001-Client.sql*, *002-Account.sql*, *003-AccountType.sql*). It is important, still, that a link exists between the application build number and the data scripts.

Automation tools implement the above-mentioned strategies with the support of checksum tables so that tracking down changes can be idempotent, hence, the same operation will just be applied once.

V. METHOD

In favor of delivery process optimization and to minimize operation incidents, the aforementioned techniques were implemented as an approach to address the control of data modifications within a software company. In spite of well scoped environments and a controlled software pipeline, the release process constantly presented flaws caused by an inefficient management over database changes. The deployment was manual and strongly dependent on a well-written documentation, which, in general, was not the case. In addition, consistency problems brought by the isolation of database deployment processes were common during the delivery stages. To manage this implementation, the collaboration between the development and operations teams was necessary in order to establish a reliable flow for the ongoing projects.

Table 1 shows metrics collected from the production environment, in the period of 5 months, before the improvement plan. The table depicts the average deployment times and the incident numbers related to databases.

Table 1. Deployments X Incidents

	Jan	Jul	Aug	Sep	Oct
Deployments	10	6	10	13	8
Average Time of Deployment (min)	40	35	30	35	40
Incidents	6	3	6	5	4

At the company, the delivery flow is composed of four environments with the following scopes:

- Integration: Intends to promote continuous integration among teams with regular commitments and unit tests.
- Quality: Environment used for acceptance tests.
- Staging: Similar to production, it is responsible for integration around systems and concerns user acceptance tests.
- Production: The last environment in the process, which is made available to the clients.

The project builds were all handled by Jenkins with Maven , whereas the version control was done through Git . Additionally, the build artifacts were stored inside Apache Archiva. For the project purpose, the Liquibase tool was included within the stack. Supporting more than 30 operations, Liquibase is focused on database refactoring and used for data migration tasks.

V.I CHANGE LOGS DESIGN

The development teams were instructed to write a single change log on every release. It is built manually and retains the differential code from the last version. Once done, the change log is committed to the version control and added to the previous scripts. As for the tracking, the change logs are appended to the current release version. To illustrate, the following tree depicts the base structure of a generic project:

project
pom.xml
project-db
src
main
assembly
liquibase.xml
changelog
data
1http://jenkins-ci.org
2http://maven.apache.org
3git-scm.com
4apache.archiva.org
5http://www.liquibase.org
1.0.0.xml
1.1.0.xml schema
1.0.0.xml
master.xml
project-ear
project-ejb
project-web

The execution is controlled by a master descriptor (master.xml), which handles the sequential execution of the scripts, as shown in listing 2.

Listing 2. Main Descriptor
```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
xmlns="http://www.liquibase.org/xml/ns/
dbchangelog"xmlns:xsi="http://www.w3.org/200
1/XMLSchemainstance"
xmlns:ext="http://www.liquibase.org/xml/ns/
dbchangelog-ext"
xsi:schemaLocation="http://www.liquibase.org
/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/
dbchangelog/dbchangelog-2.0.xsd
http://www.liquibase.org/xml/
ns/dbchangelog ext
http://www.liquibase.org/xml/ns/
dbchangelog/dbchangelog-ext.xsd">
<include
file="src/main/database/changelog/data/1.0.0
.xml" />
<include
file="src/main/database/changelog/data/1.0.1
.xml" />
<include
file="src/main/database/changelog/schema/1.0
.0.xml"
/>
</databaseChangeLog>
```

While the data folder stores the data insertion scripts, the schema directory retains the structural scripts. The Assembly directory keeps the achievement descriptor used by Maven during the release stage, whereby the change logs are deployed to Archiva to be further executed against the staging and production environments.

Following the Liquibase definitions, each change log is grouped by a sequence of change sets, representing individual operations as exemplified in listing 3.

Listing 3. Changeset sample
```
<databaseChangeLog
xmlns="http://www.liquibase.org/xml/ns/dbcha
ngelog"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://www.liquibase.org
/xml/ns/
dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/
dbchangelog
-2.0.xsd">
<changeSet id="1" author="silva"
context="integration, qa,
staging, production">
<createTable tableName="cliente">
<column name="id" type="int">
<constraints primaryKey="true"
nullable="false"/
>
</column>
<column name="name" type="varchar(50)">
<constraints nullable="false"/>
</column>
</createTable>
</changeSet>
</databaseChangeLog>
```

It is important observe that the context property, in some cases, because of environment peculiarities, can be used to link some operations to the right environments. This property is further set inline on Jenkins.

To identify the already executed change sets, Liquibase itself generates checksums to every operation and, along with the author, date and identifier attributes, stores it in a table called DATABASECHANGELOG. This table is checked before each execution to ensure that only the new change sets will be applied against the database. However, the tool provides mechanisms to treat executions within other use cases, as the recurrent execution of the same change set.

V.II DEPLOY STAGES

The development pipeline was preserved in the integration and quality environments. On each build, the scripts are cloned from Git and ran against the databases. On the other hand, to ensure the final environments reliability, the artefacts are released before the distribution to staging. Therefore, the same scripts package is deployed to production.

The release, staging and production phases run based on the success of previous tasks. Henceforth, the triggers become manual, carried out by the quality team as soon as the version is approved. Figure 2 illustrates the job sequence on Jenkins.

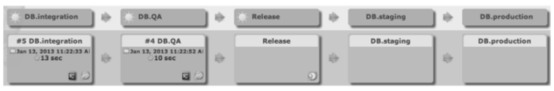

Fig. 2 Build pipeline

V.II.I DATABASE ACCESS

In order to automate the data migration tasks, the Liquibase Maven plugin must be configured as follows:

Listing 4. Liquibase Plugin

```
<?xml version="1.0"?>
<plugin>
<groupId>org.liquibase</groupId>
<artifactId>liquibase-maven-
plugin</artifactId>
<version>2.0.1</version>
<dependencies>
<dependency>
<groupId>com.microsoft.sqlserver</groupId>
<artifactId>sqljdbc4</artifactId>
<version>4.0</version>
</dependency>
</dependencies>
<configuration>
<driver>com.microsoft.sqlserver.jdbc.SQLServ
erDriver</driver>
<changeLogFile>src/main/database/changelog/m
aster.xml</changeLogFile>
<url>jdbc:sqlserver://${database.host}:${dat
abase.port};
DatabaseName=${database.name}</url>
<username>${database.username}</username>
<password>${database.password}</password>
<promptOnNonLocalDatabase>false</promptOnNon
LocalDatabase>
</configuration>
</plugin>
```

Regarding security, the credentials are stored as properties inside individual environment profiles along with the database address and port. Nevertheless, since the database name is a common property, it can be included on the project main descriptor.

Listing 5 demonstrates a suggested profile configuration.

Listing 5. Environment Profiles

```
<?xml version="1.0"?>
<profile>
<id>production</id>
<properties>
<database.host>10.2.20.1</database.host>
<database.port>1040</database.port>
<database.username>project</database.usernam
e>
<database.password>drowssap</database.passwo
rd>
</properties>
</profile>
```

V.II.II REMOTE REPOSITORY

To store change logs, a repository was created in Archiva. This space is used throughout the release and deployment tasks in the staging and production environments. The Maven configuration is presented below.

Listing 6. Remote Repository Configuration

```
<distributionManagement>
<repository>
<id>release.repo</id>
<name>Release Repository</name>
<url>http://archiva.compania.com/archiva/rep
ository/release.
repo/</url>
</repository>
<snapshotRepository>
<id>archiva.snapshots</id>
<name>Internal Snapshot Repository</name>
<url>http://archiva.compania.com/archiva/rep
ository/
snapshots/</url>
</snapshotRepository>
</distributionManagement>
```

V.II.III INTEGRATION AND TESTS

The integration and quality deployment tasks share a common profile, configured inside the data sub module Maven descriptor. The profile performs the scripts execution by running the Liquibase update goal during the process-resources phase of Maven, as exemplified in listing 7.

Listing 7. Update Profile

```
<?xml version="1.0"?>
<profile>
<id>update-db</id>
<build>
<plugins>
<plugin>
<groupId>org.liquibase</groupId>
<artifactId>liquibase-maven-
plugin</artifactId>
<version>2.0.1</version>
<executions>
<execution>
<phase>process-resources</phase>
<goals>
<goal>update</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
```

```
</build>
</profile>
```

All profiles aforementioned, as well as the context and Maven phases, are configured as parameters on Jenkins inline configuration:

```
-Pintegration -Pupdate-db -
Dliquibase.contexts=integration process-
resources
```

V.II.IV RELEASING

The release stage uses the Maven Release plugin, which increments the descriptors to the next development version and archives the scripts. This step depends on two attributes set on Jenkins:

- releaseVersion: Identification for Archiva artifacts and tags on version control.
- developmentVersion: Represents the next release to be worked by development team. The value is updated during the release stage as mentioned before.

The release step is linked to a profile (archive-db), which controls the scripts distribution.

The archiving is configured inside the data sub module and calls the Assembly plugin that comprises the information needed to the package creation. Listings 8 and 9 show the Release Plugin configuration and the archive-db profile respectively.

Listing 8. Release Plugin Configuration
```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-release-
plugin</artifactId>
<version>2.2.2</version>
<configuration>
<tagNameFormat>@{version}</tagNameFormat>
<scmCommentPrefix>Project -
</scmCommentPrefix>
<tag>${env.releaseVersion}</tag>
<releaseVersion>${env.releaseVersion}</relea
seVersion>
<developmentVersion>${env.developmentVersion
}</
developmentVersion>
<checkModificationExcludes>
<checkModificationExclude>build-number.txt</
checkModificationExclude>
</checkModificationExcludes>
<arguments>-Parchive-db</arguments>
</configuration>
</plugin>
```

Listing 9. Assembly Plugin Configuration
```
<profile>
<id>archive-db</id>
<build>
<plugins>
<plugin>
<artifactId>maven-assembly-
plugin</artifactId>
<configuration>
<descriptors>
<descriptor>src/main/assembly/liquibase.xml<
/
descriptor>
</descriptors>
</configuration>
<executions>
<execution>
<id>archive</id>
<phase>package</phase>
<goals>
<goal>single</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
</profile>
```

The assembly descriptor contains the compression format together with the change logs location, as exemplified in listing 10.

Listing 10. Assembly Descriptor
```
<assembly>
<id>liquibase</id>
<formats>
<format>zip</format>
</formats>
<includeBaseDirectory>false</includeBaseDire
ctory>
<fileSets>
<fileSet>
<useDefaultExcludes>true</useDefaultExcludes
>
<directory>src/main/changelog/</directory>
</fileSet>
<fileSet>
<useDefaultExcludes>true</useDefaultExcludes
>
<directory>src/main/changelog/</directory>
<includes>
<include>liquibase.xml</include>
</includes>
</fileSet>
</fileSets>
</assembly>
```

V.II.IV STAGING AND PRODUCTION

With the scripts stored into the repository, the subsequent tasks of staging and production can be executed. A generic project to such job is created on Git, whereby another Maven descriptor is used to download and run the scripts. This project has two stages: first, download and decompression and, second, the scripts execution is performed by Liquibase plugin as described in listing 11:

Listing 11. Generic project descriptor
```
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-dependency-
plugin</artifactId>
<configuration>
<artifactItems>
<artifactItem>
<groupId>${env.groupId}</groupId>
```

```
<artifactId>${env.artifactId}</artifactId>
<version>${env.version}</version>
<type>zip</type>
<outputDirectory>${project.build.testOutputD
irectory}<
/outputDirectory>
</artifactItem>
</artifactItems>
</configuration>
<executions>
<execution>
<phase>validate</phase>
<goals>
<goal>unpack</goal>
</goals>
</execution>
</executions>
</plugin>
<plugin>
<groupId>org.liquibase</groupId>
<artifactId>liquibase-maven-
plugin</artifactId>
<version>2.0.1</version>
<dependencies>
<dependency>
<groupId>com.microsoft.sqlserver</groupId>
<artifactId>sqljdbc4</artifactId>
<version>4.0</version>
</dependency>
</dependencies>
<configuration>
<driver>com.microsoft.sqlserver.jdbc.SQLServ
erDriver</driver>
<changeLogFile>src/main/database/changelog/m
aster.xml</changeLogFile>
<url>jdbc:sqlserver://${database.host}:${dat
abase.port};
DatabaseName=${database.name}</url>
<username>${database.username}</username>
<password>${database.password}</password>
<promptOnNonLocalDatabase>false</promptOnNon
LocalDatabase>
</configuration>
<executions>
<execution>
<phase>integration-test</phase>
<goals>
<goal>update</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
```

As for the remote repository, its information is inserted as a mirror into Maven settings, as in listing 12.

Listing 12. Release repository

```
<mirror>
<id>release.repo</id>
<url>http://archiva.company.com/archiva/repo
sitory/release.repo
/</url>
<mirrorOf>*</mirrorOf>
</mirror>
```

As seen in listing 11, the tasks created on Jenkins require four attributes although only the version number is inserted manually on every release. The parameters are described below:

- groupId: Project group identifier;
- artifactId: Artifact identifier;
- version: Artifact version to be deployed;
- databaseName: Database name where change logs will be executed on.

In addition, the command line below, appended to the task, calls the environment profile and the integration-tests phase, chosen for the Liquibase update as mentioned before:

```
-       Production integration-tests
```

VI. RESULTS

Past six months, there was a significant improvement to the collected metrics within the production environment. Two of them were considered to evaluate the efficiency of the applied practices. Figure 3 corresponds to the average time on the scripts execution since the project started, as follows:

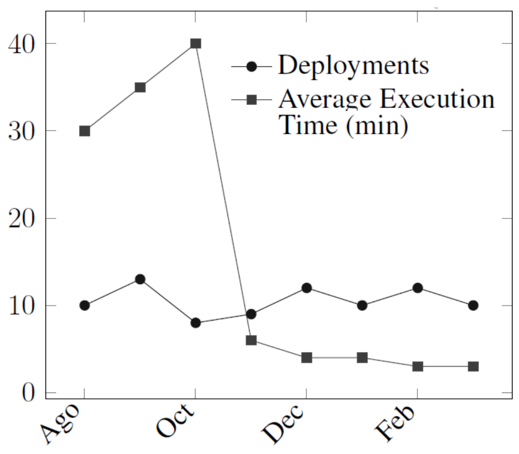

Fig. 3 Deployment Execution Time

By using the first release as an example, a manually executed project with approximately 100 scripts used to take between forty minutes and one hour to be finished.

Today, just five minutes are necessary to run and validate the same amount of data inside change logs.

Figure 4 shows the incidents generated within the same period. Notably, the incidents owed to manual deployments, which were one of the biggest problems faced by the company, drastically decreased from an average of 10 to 2.

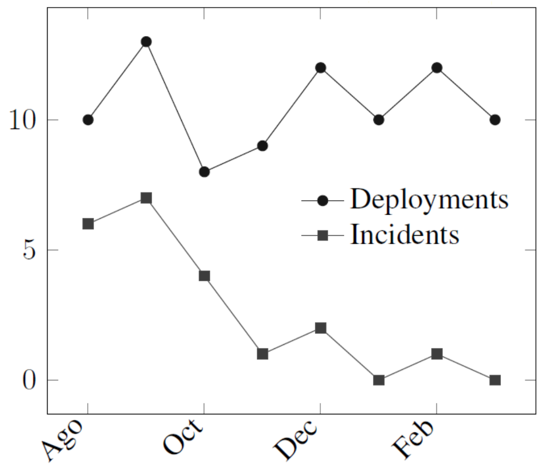

Fig. 4 Deployment Incidents

VII. CONCLUSION

At the project conclusion, it can be said that the biggest constraint still is to deal with changes resistance. It is hard to change individual mindsets as well as organizational cultures. However, despite the necessary efforts to implement the process, the results were significantly positive and the initiative achieved its goal.

With continuous improvement in mind, further projects for the pipeline optimization could be implemented, such as the release step, which still implies some manual intervention. Such task could be fully automated with incremental version numbers for the tracking, not forgetting to mention that code builds should follow the same practice.

In addition, although the work did not introduce automated tests, those could be easily inserted along with the migration tasks.

ACKNOWLEDGEMENTS

Authors Lacerda and Ribeiro would like to thank UniRitter for financial support.

REFERENCES

[1] S. Ambler, *Agile database techniques: effective strategies for the agile software developer*. New York, John Wiley & Sons, 2003.
[2] N. Ashley, *Taking control of your database development*. Available in http://dbdeploy.com/wpcontent/uploads/2007/05/taking-control-of-your-database-development.pdf.
[3] K. Beck and C. Andres, *Extreme programming explained: embrace change*. Boston, Addison-Wesley Professional, 2004.
[4] M. Fowler and K. Beck, *Refactoring: improving the design of existing code.* Boston, Addison-Wesley Professional, 1999.
[5] M. Fowler and M. Foemmel, *Continuous integration*. Available in http://www. thoughtworks. com/Continuous Integration. pdf.
[6] J. Humble and D. Farley, *Continuous Delivery*. Boston, Addison-Wesley, 2010.
[7] P. Sadalage, *Recipes for Continuous Database Integration*. Boston, Addison-Wesley, 2007.
[8] P. Schuh, *Integrating agile development in the real world.* Stamford, Cengage Learning, 2005.